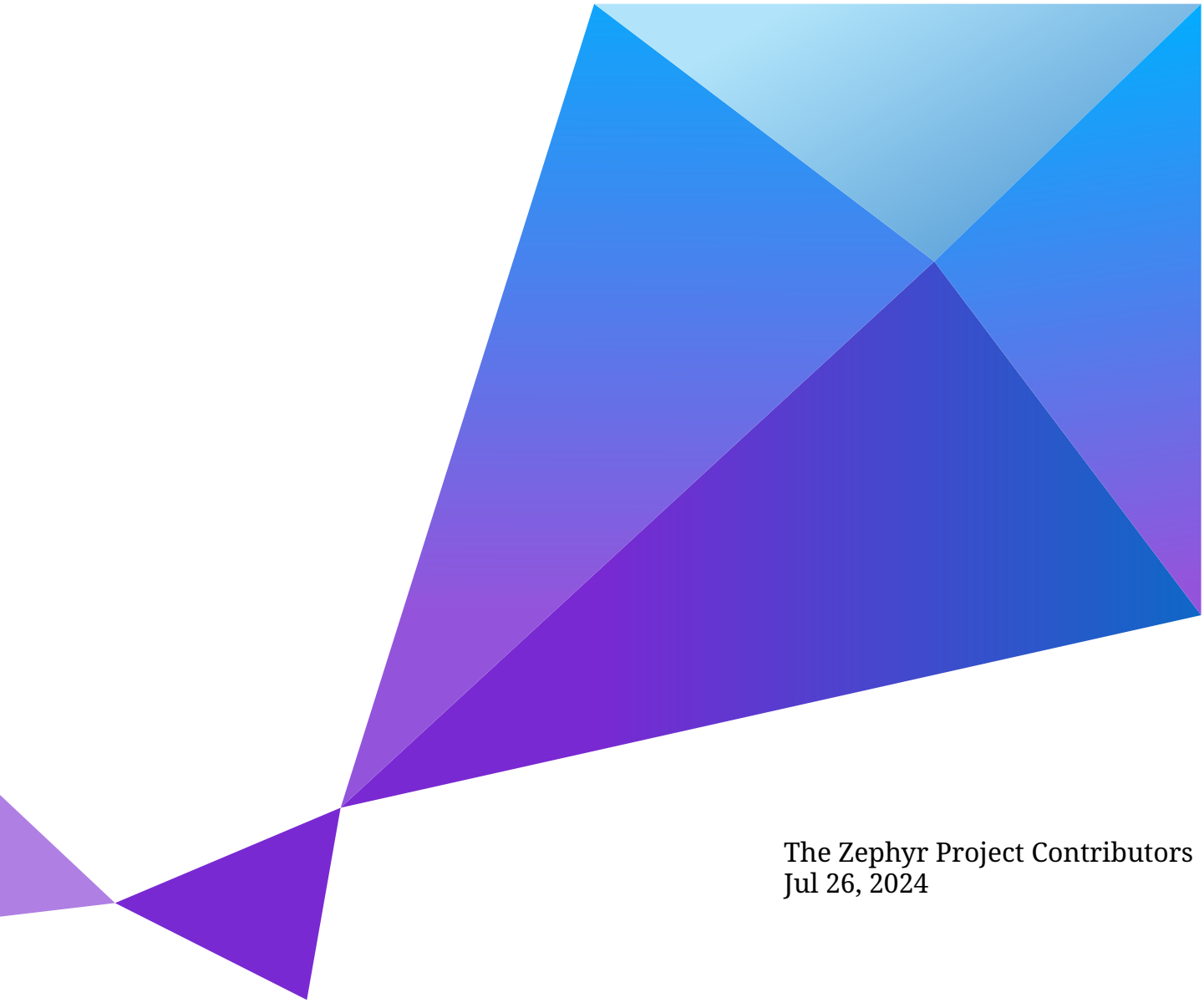
Zephyr Project Documentation

Release 3.7.0



The Zephyr Project Contributors

Jul 26, 2024

Table of contents

1 Introduction **1**
- 1.1 Licensing . . . 1
- 1.2 Distinguishing Features . . . 1
- 1.3 Community Support . . . 4
- 1.4 Resources . . . 4
  - 1.4.1 Getting Started . . . 4
  - 1.4.2 Code and Development . . . 4
  - 1.4.3 Community and Support . . . 4
  - 1.4.4 Issue Tracking and Security . . . 4
  - 1.4.5 Additional Resources . . . 4
- 1.5 Fundamental Terms and Concepts . . . 4

2 Developing with Zephyr **5**
- 2.1 Getting Started Guide . . . 5
  - 2.1.1 Select and Update OS . . . 5
  - 2.1.2 Install dependencies . . . 5
  - 2.1.3 Get Zephyr and install Python dependencies . . . 7
  - 2.1.4 Install the Zephyr SDK . . . 11
  - 2.1.5 Build the Blinky Sample . . . 13
  - 2.1.6 Flash the Sample . . . 14
  - 2.1.7 Next Steps . . . 14
  - 2.1.8 Troubleshooting Installation . . . 14
  - 2.1.9 Asking for Help . . . 15
- 2.2 Beyond the Getting Started Guide . . . 16
  - 2.2.1 Python and pip . . . 16
  - 2.2.2 Advanced Platform Setup . . . 16
  - 2.2.3 Install a Toolchain . . . 21
  - 2.2.4 Updating the Zephyr SDK toolchain . . . 22
  - 2.2.5 Cloning the Zephyr Repositories . . . 22
  - 2.2.6 Export Zephyr CMake package . . . 23
  - 2.2.7 Board Aliases . . . 23
  - 2.2.8 Build and Run an Application . . . 23
- 2.3 Environment Variables . . . 25
  - 2.3.1 Setting Variables . . . 25
  - 2.3.2 Zephyr Environment Scripts . . . 27
  - 2.3.3 Important Environment Variables . . . 27
- 2.4 Application Development . . . 28
  - 2.4.1 Overview . . . 28
  - 2.4.2 Application types . . . 30
  - 2.4.3 Creating an Application . . . 31
  - 2.4.4 Important Build System Variables . . . 33
  - 2.4.5 Application CMakeLists.txt . . . 34
  - 2.4.6 CMakeCache.txt . . . 36
  - 2.4.7 Application Configuration . . . 36
  - 2.4.8 Application-Specific Code . . . 38
  - 2.4.9 Building an Application . . . 39

2.4.10 Run an Application . . . 42
2.4.11 Custom Board, Devicetree and SOC Definitions . . . 43
2.5 Debugging . . . 47
2.5.1 Application Debugging . . . 47
2.5.2 Debug with Eclipse . . . 49
2.5.3 Debugging I2C communication . . . 51
2.6 API Status and Guidelines . . . 52
2.6.1 API Overview . . . 52
2.6.2 API Lifecycle . . . 65
2.6.3 API Design Guidelines . . . 69
2.6.4 API Terminology . . . 71
2.7 Language Support . . . 73
2.7.1 C Language Support . . . 73
2.7.2 C++ Language Support . . . 84
2.8 Optimizations . . . 88
2.8.1 Optimizing for Footprint . . . 88
2.8.2 Optimization Tools . . . 89
2.9 Flashing and Hardware Debugging . . . 93
2.9.1 Flash & Debug Host Tools . . . 93
2.9.2 Debug Probes . . . 99
2.10 Modules (External projects) . . . 106
2.10.1 Modules vs west projects . . . 107
2.10.2 Module Repositories . . . 107
2.10.3 Contributing to Zephyr modules . . . 109
2.10.4 Licensing requirements and policies . . . 110
2.10.5 Documentation requirements . . . 111
2.10.6 Testing requirements . . . 111
2.10.7 Deprecating and removing modules . . . 112
2.10.8 Integrate modules in Zephyr build system . . . 112
2.10.9 Module yaml file description . . . 113
2.10.10 Submitting changes to modules . . . 121
2.11 West (Zephyr's meta-tool) . . . 122
2.11.1 Installing west . . . 123
2.11.2 West Release Notes . . . 124
2.11.3 Troubleshooting West . . . 136
2.11.4 Basics . . . 139
2.11.5 Built-in commands . . . 142
2.11.6 Workspaces . . . 145
2.11.7 West Manifests . . . 150
2.11.8 Configuration . . . 179
2.11.9 Extensions . . . 182
2.11.10 Building, Flashing and Debugging . . . 185
2.11.11 Signing Binaries . . . 201
2.11.12 Additional Zephyr extension commands . . . 204
2.11.13 History and Motivation . . . 207
2.11.14 Moving to West . . . 209
2.11.15 Using Zephyr without west . . . 209
2.12 Testing . . . 211
2.12.1 Test Framework . . . 211
2.12.2 Test Runner (Twister) . . . 240
2.12.3 Twister blackbox tests . . . 261
2.12.4 Integration with pytest test framework . . . 264
2.12.5 Generating coverage reports . . . 270
2.12.6 BabbleSim . . . 272
2.12.7 ZTest Deprecated APIs . . . 275
2.13 Static Code Analysis (SCA) . . . 281
2.13.1 SCA Tool infrastructure . . . 281
2.13.2 Native SCA Tool support . . . 282

2.14 Toolchains . . . 284
2.14.1 Zephyr SDK . . . 284
2.14.2 Arm Compiler 6 . . . 288
2.14.3 Cadence Tensilica Xtensa C/C++ Compiler (XCC) . . . 288
2.14.4 DesignWare ARC MetaWare Development Toolkit (MWDT) . . . 289
2.14.5 GNU Arm Embedded . . . 290
2.14.6 Intel oneAPI Toolkit . . . 291
2.14.7 Crosstool-NG (Deprecated) . . . 291
2.14.8 Host Toolchains . . . 292
2.14.9 Other Cross Compilers . . . 292
2.14.10 Custom CMake Toolchains . . . 293
2.15 Tools and IDEs . . . 294
2.15.1 CLion . . . 294
2.15.2 Coccinelle . . . 299
2.15.3 Visual Studio Code . . . 307

3 Kernel **309**
3.1 Kernel Services . . . 309
3.1.1 Scheduling, Interrupts, and Synchronization . . . 309
3.1.2 Data Passing . . . 426
3.1.3 Memory Management . . . 469
3.1.4 Timing . . . 469
3.1.5 Other . . . 490
3.2 Device Driver Model . . . 511
3.2.1 Introduction . . . 511
3.2.2 Standard Drivers . . . 512
3.2.3 Synchronous Calls . . . 512
3.2.4 Driver APIs . . . 512
3.2.5 Driver Data Structures . . . 512
3.2.6 Subsystems and API Structures . . . 513
3.2.7 Device-Specific API Extensions . . . 514
3.2.8 Single Driver, Multiple Instances . . . 515
3.2.9 Initialization Levels . . . 516
3.2.10 Deferred initialization . . . 517
3.2.11 System Drivers . . . 517
3.2.12 Inspecting the initialization sequence . . . 517
3.2.13 Error handling . . . 517
3.2.14 Memory Mapping . . . 518
3.2.15 API Reference . . . 521
3.3 User Mode . . . 531
3.3.1 Overview . . . 531
3.3.2 Memory Protection Design . . . 533
3.3.3 Kernel Objects . . . 542
3.3.4 System Calls . . . 549
3.3.5 MPU Stack Objects . . . 558
3.3.6 MPU Backed Userspace . . . 559
3.4 Memory Management . . . 559
3.4.1 Memory Heaps . . . 559
3.4.2 Shared Multi Heap . . . 576
3.4.3 Memory Slabs . . . 580
3.4.4 Memory Blocks Allocator . . . 585
3.4.5 Demand Paging . . . 593
3.4.6 Virtual Memory . . . 601
3.5 Data Structures . . . 606
3.5.1 Single-linked List . . . 606
3.5.2 Double-linked List . . . 619
3.5.3 Multi Producer Single Consumer Packet Buffer . . . 627
3.5.4 Single Producer Single Consumer Packet Buffer . . . 629

3.5.5 Balanced Red/Black Tree . . . 630
3.5.6 Ring Buffers . . . 634
3.5.7 Multi Producer Single Consumer Lock Free Queue . . . 647
3.5.8 Single Producer Single Consumer Lock Free Queue . . . 648
3.6 Executing Time Functions . . . 650
3.6.1 Configuration . . . 650
3.6.2 Usage . . . 651
3.6.3 API documentation . . . 651
3.7 Object Cores . . . 661
3.7.1 Object Core Concepts . . . 662
3.7.2 Object Core Statistics Concepts . . . 662
3.7.3 Implementation . . . 663
3.7.4 Configuration Options . . . 665
3.7.5 API Reference . . . 665
3.8 Time Utilities . . . 671
3.8.1 Overview . . . 671
3.8.2 Time Utility APIs . . . 672
3.8.3 Concepts Underlying Time Support in Zephyr . . . 677
3.9 Utilities . . . 679
3.10 Iterable Sections . . . 699
3.10.1 Usage . . . 699
3.10.2 API Reference . . . 700
3.11 Code And Data Relocation . . . 705
3.11.1 Overview . . . 705
3.11.2 Details . . . 706

4 OS Services 709
4.1 Binary Descriptors . . . 709
4.1.1 Internals . . . 709
4.1.2 Usage . . . 710
4.1.3 API Reference . . . 711
4.2 Console . . . 713
4.3 Cryptography . . . 714
4.3.1 PSA Crypto . . . 714
4.3.2 Random Number Generation . . . 717
4.3.3 Crypto APIs . . . 720
4.4 Debugging . . . 727
4.4.1 Thread analyzer . . . 727
4.4.2 Core Dump . . . 729
4.4.3 GDB stub . . . 737
4.4.4 Cortex-M Debug Monitor . . . 742
4.4.5 MIPI STP Decoder . . . 743
4.4.6 Symbol Table (Symtab) . . . 746
4.5 Device Management . . . 747
4.5.1 MCUmgr . . . 747
4.5.2 MCUmgr handlers . . . 755
4.5.3 MCUmgr Callbacks . . . 763
4.5.4 Fixing and backporting fixes to Zephyr v2.7 MCUmgr . . . 773
4.5.5 SMP Protocol Specification . . . 776
4.5.6 SMP Transport Specification . . . 814
4.5.7 Device Firmware Upgrade . . . 819
4.5.8 Over-the-Air Update . . . 825
4.5.9 EC Host Command . . . 826
4.5.10 SMP Groups . . . 839
4.6 Digital Signal Processing (DSP) . . . 840
4.6.1 Using zDSP . . . 840
4.6.2 Optimizing for your architecture . . . 841
4.6.3 API Reference . . . 841

4.7 File Systems . . . 842
4.7.1 Samples . . . 842
4.7.2 API Reference . . . 843
4.8 Formatted Output . . . 859
4.8.1 Cbprintf Packaging . . . 860
4.8.2 API Reference . . . 862
4.9 Input . . . 872
4.9.1 Input Events . . . 872
4.9.2 Input Devices . . . 873
4.9.3 Application API . . . 873
4.9.4 HID code mapping . . . 873
4.9.5 Kscan Compatibility . . . 873
4.9.6 Driver Documentation . . . 874
4.9.7 API Reference . . . 882
4.9.8 Input Event Definitions . . . 885
4.9.9 Analog Axis API Reference . . . 899
4.10 Interprocessor Communication (IPC) . . . 900
4.10.1 IPC service . . . 900
4.11 Linkable Loadable Extensions (LLEXT) . . . 917
4.11.1 Configuration . . . 917
4.11.2 Building extensions . . . 919
4.11.3 Loading extensions . . . 922
4.11.4 Troubleshooting . . . 923
4.11.5 API Reference . . . 923
4.12 Logging . . . 931
4.12.1 Global Kconfig Options . . . 933
4.12.2 Usage . . . 935
4.12.3 Logging panic . . . 937
4.12.4 Printk . . . 937
4.12.5 Architecture . . . 937
4.12.6 Recommendations . . . 943
4.12.7 Benchmark . . . 944
4.12.8 Stack usage . . . 944
4.12.9 API Reference . . . 944
4.13 Tracing . . . 965
4.13.1 Overview . . . 965
4.13.2 Serialization Formats . . . 966
4.13.3 Transport Backends . . . 968
4.13.4 Using Tracing . . . 968
4.13.5 Visualisation Tools . . . 969
4.13.6 Future LTTng Inspiration . . . 969
4.13.7 Object tracking . . . 970
4.13.8 API . . . 971
4.14 Resource Management . . . 1001
4.14.1 On-Off Manager . . . 1001
4.15 Memory Attributes . . . 1011
4.15.1 Migration guide from *zephyr,memory-region-mpu* . . . 1012
4.15.2 Memory Attributes Heap Allocator . . . 1012
4.15.3 API Reference . . . 1014
4.16 Modbus . . . 1017
4.16.1 Samples . . . 1017
4.16.2 API Reference . . . 1017
4.17 Modem modules . . . 1028
4.17.1 Modem pipe . . . 1028
4.17.2 Modem PPP . . . 1031
4.17.3 Modem CMUX . . . 1032
4.17.4 Modem pipelink . . . 1035
4.18 Asynchronous Notifications . . . 1038

4.18.1 API Reference . . . 1038
4.19 Power Management . . . 1041
4.19.1 Overview . . . 1041
4.19.2 System Power Management . . . 1042
4.19.3 Device Power Management . . . 1044
4.19.4 Device Runtime Power Management . . . 1050
4.19.5 Power Domain . . . 1055
4.19.6 Power Management APIs . . . 1058
4.20 OS Abstraction . . . 1082
4.20.1 POSIX . . . 1082
4.20.2 CMSIS RTOS v1 . . . 1110
4.20.3 CMSIS RTOS v2 . . . 1110
4.21 Power off . . . 1111
4.22 Shell . . . 1111
4.22.1 Overview . . . 1112
4.22.2 Backends . . . 1113
4.22.3 Commands . . . 1114
4.22.4 Tab Feature . . . 1121
4.22.5 History Feature . . . 1121
4.22.6 Wildcards Feature . . . 1122
4.22.7 Meta Keys Feature . . . 1122
4.22.8 Getopt Feature . . . 1123
4.22.9 Obscured Input Feature . . . 1123
4.22.10 Shell Logger Backend Feature . . . 1124
4.22.11 RTT Backend Channel Selection . . . 1124
4.22.12 Usage . . . 1124
4.22.13 API Reference . . . 1126
4.23 Serialization . . . 1147
4.23.1 Nanopb . . . 1147
4.24 Settings . . . 1148
4.24.1 Handlers . . . 1148
4.24.2 Backends . . . 1149
4.24.3 Zephyr Storage Backends . . . 1149
4.24.4 Storage Location . . . 1149
4.24.5 Loading data from persisted storage . . . 1149
4.24.6 Storing data to persistent storage . . . 1150
4.24.7 Secure domain settings . . . 1150
4.24.8 Example: Device Configuration . . . 1150
4.24.9 Example: Persist Runtime State . . . 1151
4.24.10 Example: Custom Backend Implementation . . . 1152
4.24.11 API Reference . . . 1153
4.25 State Machine Framework . . . 1163
4.25.1 Overview . . . 1163
4.25.2 State Creation . . . 1163
4.25.3 State Machine Creation . . . 1163
4.25.4 State Machine Execution . . . 1164
4.25.5 Preventing Parent Run Actions . . . 1164
4.25.6 State Machine Termination . . . 1164
4.25.7 UML State Machines . . . 1165
4.25.8 State Machine Examples . . . 1165
4.25.9 API Reference . . . 1172
4.26 Storage . . . 1175
4.26.1 Non-Volatile Storage (NVS) . . . 1175
4.26.2 Disk Access . . . 1180
4.26.3 Flash map . . . 1187
4.26.4 Flash Circular Buffer (FCB) . . . 1194
4.26.5 Stream Flash . . . 1200
4.27 Sensing Subsystem . . . 1203

4.27.1 Overview . . . 1204
4.27.2 Configurability . . . 1204
4.27.3 Main Features . . . 1205
4.27.4 Major Flows . . . 1206
4.27.5 Sensor Types And Instance . . . 1207
4.27.6 Sensor Instance Handler . . . 1207
4.27.7 Sensor Sample Value . . . 1208
4.27.8 Device Tree Configuration . . . 1209
4.27.9 API Reference . . . 1209
4.28 Task Watchdog . . . 1219
4.28.1 Overview . . . 1219
4.28.2 Configuration Options . . . 1220
4.28.3 API Reference . . . 1220
4.29 Trusted Firmware-M . . . 1221
4.29.1 Trusted Firmware-M Overview . . . 1221
4.29.2 TF-M Requirements . . . 1225
4.29.3 TF-M Build System . . . 1227
4.29.4 Trusted Firmware-M Integration . . . 1230
4.29.5 Test Suites . . . 1231
4.30 Virtualization . . . 1231
4.30.1 Inter-VM Shared Memory . . . 1232
4.31 Retention System . . . 1234
4.31.1 Devicetree setup . . . 1235
4.31.2 Mutex protection . . . 1236
4.31.3 Boot mode . . . 1236
4.31.4 Retention system modules . . . 1237
4.31.5 API Reference . . . 1240
4.32 Real Time I/O (RTIO) . . . 1243
4.32.1 Problem . . . 1243
4.32.2 Inspiration, introducing io_uring . . . 1244
4.32.3 Submission Queue . . . 1244
4.32.4 Completion Queue . . . 1245
4.32.5 Executor . . . 1245
4.32.6 IO Device . . . 1245
4.32.7 Cancellation . . . 1245
4.32.8 Memory pools . . . 1245
4.32.9 When to Use . . . 1246
4.32.10 API Reference . . . 1246
4.33 Zephyr bus (zbus) . . . 1257
4.33.1 Concepts . . . 1258
4.33.2 Usage . . . 1266
4.33.3 Samples . . . 1272
4.33.4 Suggested Uses . . . 1273
4.33.5 Configuration Options . . . 1273
4.33.6 API Reference . . . 1274
4.34 Miscellaneous . . . 1288
4.34.1 Checksum APIs . . . 1288
4.34.2 Structured Data APIs . . . 1294

5 Build and Configuration Systems 1307
5.1 Build System (CMake) . . . 1307
5.1.1 Build and Configuration Phases . . . 1307
5.1.2 Supporting Scripts and Tools . . . 1313
5.2 Devicetree . . . 1317
5.2.1 Devicetree Guide . . . 1317
5.2.2 Devicetree Reference . . . 1378
5.3 Configuration System (Kconfig) . . . 1576
5.3.1 Interactive Kconfig interfaces . . . 1576

5.3.2 Setting Kconfig configuration values . . . 1581
5.3.3 Kconfig - Tips and Best Practices . . . 1586
5.3.4 Custom Kconfig Preprocessor Functions . . . 1601
5.3.5 Kconfig extensions . . . 1603
5.4 Snippets . . . 1604
5.4.1 Using Snippets . . . 1605
5.4.2 Built-in snippets . . . 1605
5.4.3 Writing Snippets . . . 1608
5.4.4 Snippets Design . . . 1611
5.5 Zephyr CMake Package . . . 1612
5.5.1 Zephyr CMake package export (west) . . . 1613
5.5.2 Zephyr CMake package export (without west) . . . 1613
5.5.3 Zephyr Base Environment Setting . . . 1613
5.5.4 Zephyr CMake Package Search Order . . . 1613
5.5.5 Zephyr CMake Package Version . . . 1614
5.5.6 Multiple Zephyr Installations (Zephyr workspace) . . . 1615
5.5.7 Zephyr Build Configuration CMake packages . . . 1616
5.5.8 Zephyr CMake package source code . . . 1618
5.6 Sysbuild (System build) . . . 1618
5.6.1 Definitions . . . 1619
5.6.2 Architectural Overview . . . 1619
5.6.3 Building with sysbuild . . . 1620
5.6.4 Configuration namespacing . . . 1620
5.6.5 Sysbuild flashing using `west flash` . . . 1622
5.6.6 Sysbuild debugging using `west debug` . . . 1622
5.6.7 Building a sample with MCUboot . . . 1622
5.6.8 Sysbuild Kconfig file . . . 1623
5.6.9 Sysbuild targets . . . 1624
5.6.10 Dedicated image build targets . . . 1624
5.6.11 Adding Zephyr applications to sysbuild . . . 1624
5.6.12 Adding non-Zephyr applications to sysbuild . . . 1628
5.6.13 Extending sysbuild . . . 1628
5.7 Application version management . . . 1629
5.7.1 VERSION file . . . 1629
5.7.2 Use in code . . . 1630
5.7.3 Use in Kconfig . . . 1630
5.7.4 Use in CMake . . . 1631
5.7.5 Use in MCUboot-supported applications . . . 1632
5.8 Flashing . . . 1632
5.8.1 Flashing configuration . . . 1632

6 Connectivity **1635**
6.1 Bluetooth . . . 1635
6.1.1 Supported features . . . 1635
6.1.2 Qualification . . . 1637
6.1.3 Stack Architecture . . . 1657
6.1.4 LE Host . . . 1662
6.1.5 LE Audio Stack . . . 1666
6.1.6 LE Audio resources . . . 1686
6.1.7 LE Controller . . . 1687
6.1.8 Application Development . . . 1701
6.1.9 API . . . 1705
6.1.10 Tools . . . 2429
6.1.11 Shell . . . 2434
6.2 Controller Area Network (CAN) Bus Protocols . . . 2442
6.2.1 ISO-TP Transport Protocol . . . 2442
6.3 Networking . . . 2448
6.3.1 Overview . . . 2449

6.3.2 Network Stack Architecture . . . 2451
6.3.3 Network Configuration Guide . . . 2457
6.3.4 Networking with the host system . . . 2461
6.3.5 Monitor Network Traffic . . . 2475
6.3.6 Networking APIs . . . 2479
6.3.7 Connection Manager . . . 2969
6.4 LoRa and LoRaWAN . . . 2995
6.4.1 Overview . . . 2995
6.4.2 Configuration Options . . . 2995
6.4.3 API Reference . . . 2996
6.5 USB . . . 3009
6.5.1 USB device support . . . 3009
6.5.2 USB device support APIs . . . 3019
6.5.3 New USB device support . . . 3036
6.5.4 New USB device support APIs . . . 3039
6.5.5 USB host support APIs . . . 3068
6.5.6 USB-C device stack . . . 3077
6.5.7 Human Interface Devices (HID) . . . 3096

7 Hardware Support 3111
7.1 Architecture-related Guides . . . 3111
7.1.1 Zephyr support status on ARC processors . . . 3111
7.1.2 Arm Cortex-M Developer Guide . . . 3112
7.1.3 Zephyr support status on RISC-V processors . . . 3122
7.1.4 Semihosting Guide . . . 3123
7.1.5 Additional Functionality . . . 3124
7.1.6 x86 Developer Guide . . . 3128
7.1.7 Xtensa Developer Guide . . . 3129
7.2 Barriers API . . . 3130
7.3 Cache Interface . . . 3131
7.3.1 Cache API . . . 3132
7.4 Zephyr's device emulators/simulators . . . 3138
7.4.1 Overview . . . 3138
7.4.2 Available Emulators . . . 3139
7.5 External Bus and Bus Connected Peripherals Emulators . . . 3140
7.5.1 Overview . . . 3140
7.5.2 Concept . . . 3140
7.5.3 Creating a Device Driver Emulator . . . 3141
7.5.4 Available Emulators . . . 3143
7.5.5 Samples . . . 3143
7.6 Peripherals . . . 3147
7.6.1 1-Wire Bus . . . 3147
7.6.2 Analog-to-Digital Converter (ADC) . . . 3157
7.6.3 Auxiliary Display (auxdisplay) . . . 3172
7.6.4 Audio . . . 3181
7.6.5 Battery Backed RAM (BBRAM) . . . 3211
7.6.6 BC1.2 Devices (Experimental) . . . 3215
7.6.7 Clock Control . . . 3219
7.6.8 Controller Area Network (CAN) . . . 3223
7.6.9 Chargers . . . 3258
7.6.10 Coredump Device . . . 3265
7.6.11 Counter . . . 3267
7.6.12 Digital-to-Analog Converter (DAC) . . . 3276
7.6.13 Direct Memory Access (DMA) . . . 3277
7.6.14 Display Interface . . . 3290
7.6.15 Electrically Erasable Programmable Read-Only Memory (EEPROM) . . . 3306
7.6.16 Enhanced Serial Peripheral Interface (eSPI) Bus . . . 3308
7.6.17 Entropy . . . 3327

7.6.18 Error Detection And Correction (EDAC) . . . 3329
7.6.19 Flash . . . 3334
7.6.20 Fuel Gauge . . . 3342
7.6.21 GNSS (Global Navigation Satellite System) . . . 3350
7.6.22 General-Purpose Input/Output (GPIO) . . . 3361
7.6.23 Hardware Information . . . 3384
7.6.24 I2C EEPROM Target . . . 3387
7.6.25 Improved Inter-Integrated Circuit (I3C) Bus . . . 3388
7.6.26 Inter-Integrated Circuit (I2C) Bus . . . 3448
7.6.27 Inter-Processor Mailbox (IPM) . . . 3468
7.6.28 Keyboard Scan . . . 3471
7.6.29 Light-Emitting Diode (LED) . . . 3473
7.6.30 Management Data Input/Output (MDIO) . . . 3481
7.6.31 MIPI Display Bus Interface (DBI) . . . 3483
7.6.32 MIPI Display Serial Interface (DSI) . . . 3488
7.6.33 Multi-bit SPI Bus . . . 3493
7.6.34 Multi-Channel Inter-Processor Mailbox (MBOX) . . . 3501
7.6.35 Peripheral Component Interconnect express Bus (PCIe) . . . 3507
7.6.36 Platform Environment Control Interface (PECI) . . . 3515
7.6.37 PS/2 . . . 3523
7.6.38 Pulse Width Modulation (PWM) . . . 3525
7.6.39 Real-Time Clock (RTC) . . . 3540
7.6.40 Regulators . . . 3549
7.6.41 Reset Controller . . . 3554
7.6.42 Retained Memory . . . 3559
7.6.43 Secure Digital High Capacity (SDHC) . . . 3562
7.6.44 Sensors . . . 3572
7.6.45 Serial Peripheral Interface (SPI) Bus . . . 3606
7.6.46 System Management Bus (SMBus) . . . 3621
7.6.47 Universal Asynchronous Receiver-Transmitter (UART) . . . 3633
7.6.48 USB-C VBUS . . . 3653
7.6.49 USB Type-C Port Controller (TCPC) . . . 3654
7.6.50 Time-aware General-Purpose Input/Output (TGPIO) . . . 3692
7.6.51 Video . . . 3694
7.6.52 Watchdog . . . 3703
7.7 Pin Control . . . 3706
7.7.1 Introduction . . . 3707
7.7.2 State model . . . 3708
7.7.3 Dynamic pin control . . . 3709
7.7.4 Devicetree representation . . . 3710
7.7.5 Implementation guidelines . . . 3712
7.7.6 Pin Control API . . . 3714
7.7.7 Other reference material . . . 3719
7.8 Porting . . . 3719
7.8.1 Architecture Porting Guide . . . 3719
7.8.2 SoC Porting Guide . . . 3754
7.8.3 Board Porting Guide . . . 3759
7.8.4 Shields . . . 3772

8 Contributing to Zephyr 3777
8.1 General Guidelines . . . 3777
8.1.1 Contribution Guidelines . . . 3777
8.1.2 Coding Guidelines . . . 3791
8.1.3 Proposals and RFCs . . . 3807
8.1.4 Contributor Expectations . . . 3807
8.2 Documentation . . . 3812
8.2.1 Documentation Guidelines . . . 3812
8.2.2 Documentation Generation . . . 3823

8.3 Dealing with external components . . . 3827
8.3.1 Contributing External Components . . . 3828
8.3.2 Binary Blobs . . . 3832
8.4 Zephyr Contributor Badge . . . 3835
8.5 Need help along the way? . . . 3836

9 Project and Governance 3837
9.1 Technical Steering Committee (TSC) . . . 3837
9.1.1 TSC Member Role . . . 3837
9.2 TSC Project Roles . . . 3840
9.2.1 Project Roles . . . 3840
9.2.2 Role Retirement . . . 3842
9.2.3 Teams and Supporting Activities . . . 3843
9.2.4 MAINTAINERS File . . . 3845
9.2.5 Release Activity . . . 3845
9.3 TSC Working Groups . . . 3847
9.3.1 Overview . . . 3847
9.3.2 Membership . . . 3847
9.3.3 Advisory role . . . 3848
9.3.4 TSC Working Group Lifecycle . . . 3849
9.4 Release Process . . . 3849
9.4.1 Development Phase . . . 3850
9.4.2 Stabilization Phase . . . 3850
9.4.3 Release Criteria . . . 3851
9.4.4 Release Milestones . . . 3852
9.4.5 Releases . . . 3852
9.4.6 Hardware Support Tiers . . . 3855
9.4.7 Release Procedure . . . 3856
9.5 Feature Tracking . . . 3857
9.5.1 Roadmap and Release Plans . . . 3858
9.6 Code Flow and Branches . . . 3859
9.6.1 Introduction . . . 3859
9.6.2 Roles and Responsibilities . . . 3859
9.7 Modifying Contributions made by other developers . . . 3860
9.7.1 Scenarios . . . 3860
9.7.2 Accepted policies . . . 3860
9.8 Development Environment and Tools . . . 3861
9.8.1 Code Review . . . 3861
9.8.2 Continuous Integration . . . 3865
9.8.3 Labeling issues and pull requests in GitHub . . . 3865
9.9 Bug Reporting . . . 3867
9.9.1 Reporting a regression issue . . . 3867
9.10 Communication and Collaboration . . . 3868
9.11 Code Documentation . . . 3868
9.11.1 API Documentation . . . 3868
9.11.2 Reference to Requirements . . . 3868
9.11.3 Test Documentation . . . 3868
9.11.4 Documentation Guidelines . . . 3869
9.12 Terminology . . . 3870

10 Security 3871
10.1 Zephyr Security Overview . . . 3871
10.1.1 Introduction . . . 3871
10.1.2 Current Security Definition . . . 3872
10.1.3 Secure Development Process . . . 3875
10.1.4 Secure Design . . . 3878
10.1.5 Security Certification . . . 3880
10.2 Security Vulnerability Reporting . . . 3881

10.2.1 Introduction . . . 3881
10.2.2 Security Issue Management . . . 3881
10.2.3 Vulnerability Notification . . . 3883
10.2.4 Backporting of Security Vulnerabilities . . . 3884
10.2.5 Need to Know . . . 3884
10.3 Secure Coding . . . 3884
10.3.1 Introduction and Scope . . . 3885
10.3.2 Secure Coding . . . 3885
10.3.3 Secure development knowledge . . . 3886
10.3.4 Code Review . . . 3887
10.3.5 Issues and Bug Tracking . . . 3887
10.3.6 Modifications to This Document . . . 3888
10.4 Sensor Device Threat Model . . . 3888
10.4.1 Assets . . . 3888
10.4.2 Communication . . . 3890
10.4.3 Other Considerations . . . 3892
10.4.4 Threats . . . 3892
10.4.5 Notes . . . 3892
10.5 Hardening Tool . . . 3892
10.5.1 Usage . . . 3893
10.6 Vulnerabilities . . . 3893
10.6.1 CVE-2017 . . . 3893
10.6.2 CVE-2019 . . . 3894
10.6.3 CVE-2020 . . . 3895
10.6.4 CVE-2021 . . . 3903
10.6.5 CVE-2022 . . . 3908
10.6.6 CVE-2023 . . . 3909
10.6.7 CVE-2024 . . . 3914
10.7 Security standards and Zephyr . . . 3916
10.7.1 ETSI 303-645 . . . 3916

11 Safety **3927**
11.1 Zephyr Safety Overview . . . 3927
11.1.1 Introduction . . . 3927
11.1.2 Overview . . . 3927
11.1.3 Safety Document update . . . 3927
11.1.4 General safety scope . . . 3928
11.1.5 Quality . . . 3928
11.1.6 Processes and workflow . . . 3930
11.2 Safety Requirements . . . 3932
11.2.1 Introduction . . . 3932
11.2.2 Guidelines . . . 3932

12 Glossary of Terms **3935**

Bibliography **3939**

Python Module Index **3941**

Index **3943**

Chapter 1

Introduction

The Zephyr OS is based on a small-footprint kernel designed for use on resource-constrained and embedded systems: from simple embedded environmental sensors and LED wearables to sophisticated embedded controllers, smart watches, and IoT wireless applications.

The Zephyr kernel supports multiple architectures, including:

- ARCv2 (EM and HS) and ARCv3 (HS6X)
- ARMv6-M, ARMv7-M, and ARMv8-M (Cortex-M)
- ARMv7-A and ARMv8-A (Cortex-A, 32- and 64-bit)
- ARMv7-R, ARMv8-R (Cortex-R, 32- and 64-bit)
- Intel x86 (32- and 64-bit)
- MIPS (MIPS32 Release 1 specification)
- NIOS II Gen 2
- RISC-V (32- and 64-bit)
- SPARC V8
- Tensilica Xtensa

The full list of supported boards based on these architectures can be found here.

In the context of the Zephyr OS, a *subsystem* refers to a logically distinct part of the operating system that handles specific functionality or provides certain services. Subsystems can include components such as networking, file systems, device driver classes, power management, and communication protocols, among others. Each subsystem is designed to be modular and can be configured, customized, and extended to meet the requirements of different embedded applications.

1.1 Licensing

Zephyr is permissively licensed using the Apache 2.0 license (as found in the `LICENSE` file in the project's GitHub repo). There are some imported or reused components of the Zephyr project that use other licensing, as described in *Licensing of Zephyr Project components*.

1.2 Distinguishing Features

Zephyr offers a large and ever growing number of features including:

Extensive suite of Kernel services
Zephyr offers a number of familiar services for development:

- *Multi-threading Services* for cooperative, priority-based, non-preemptive, and preemptive threads with optional round robin time-slicing. Includes POSIX pthreads compatible API support.
- *Interrupt Services* for compile-time registration of interrupt handlers.
- *Memory Allocation Services* for dynamic allocation and freeing of fixed-size or variable-size memory blocks.
- *Inter-thread Synchronization Services* for binary semaphores, counting semaphores, and mutex semaphores.
- *Inter-thread Data Passing Services* for basic message queues, enhanced message queues, and byte streams.
- *Power Management Services* such as overarching, application or policy-defined, System Power Management and fine-grained, driver-defined, Device Power Management.

Multiple Scheduling Algorithms
Zephyr provides a comprehensive set of thread scheduling choices:

- Cooperative and Preemptive Scheduling
- Earliest Deadline First (EDF)
- Meta IRQ scheduling implementing "interrupt bottom half" or "tasklet" behavior
- Timeslicing: Enables time slicing between preemptible threads of equal priority
- Multiple queuing strategies:
  - Simple linked-list ready queue
  - Red/black tree ready queue
  - Traditional multi-queue ready queue

Highly configurable / Modular for flexibility
Allows an application to incorporate *only* the capabilities it needs as it needs them, and to specify their quantity and size.

Cross Architecture
Supports a wide variety of supported boards with different CPU architectures and developer tools. Contributions have added support for an increasing number of SoCs, platforms, and drivers.

Memory Protection
Implements configurable architecture-specific stack-overflow protection, kernel object and device driver permission tracking, and thread isolation with thread-level memory protection on x86, ARC, and ARM architectures, userspace, and memory domains.

For platforms without MMU/MPU and memory constrained devices, supports combining application-specific code with a custom kernel to create a monolithic image that gets loaded and executed on a system's hardware. Both the application code and kernel code execute in a single shared address space.

Compile-time resource definition
Allows system resources to be defined at compile-time, which reduces code size and increases performance for resource-limited systems.

Optimized Device Driver Model
Provides a consistent device model for configuring the drivers that are part of the platform/system and a consistent model for initializing all the drivers configured into the system and allows the reuse of drivers across platforms that have common devices/IP blocks.

Devicetree Support
Use of *devicetree* to describe hardware. Information from devicetree is used to create the application image.

Native Networking Stack supporting multiple protocols
Networking support is fully featured and optimized, including LwM2M and BSD sockets compatible support. OpenThread support (on Nordic chipsets) is also provided - a mesh network designed to securely and reliably connect hundreds of products around the home.

Bluetooth Low Energy 5.0 support
Bluetooth 5.0 compliant (ESR10) and Bluetooth Low Energy Controller support (LE Link Layer). Includes Bluetooth Mesh and a Bluetooth qualification-ready Bluetooth controller.

- Generic Access Profile (GAP) with all possible LE roles
- Generic Attribute Profile (GATT)
- Pairing support, including the Secure Connections feature from Bluetooth 4.2
- Clean HCI driver abstraction
- Raw HCI interface to run Zephyr as a Controller instead of a full Host stack
- Verified with multiple popular controllers
- Highly configurable

Mesh Support:

- Relay, Friend Node, Low-Power Node (LPN) and GATT Proxy features
- Both Provisioning bearers supported (PB-ADV & PB-GATT)
- Highly configurable, fitting in devices with at least 16k RAM

Native Linux, macOS, and Windows Development
A command-line CMake build environment runs on popular developer OS systems. A native port (native_sim) lets you build and run Zephyr as a native application on Linux, aiding development and testing.

Virtual File System Interface with ext2, FatFs, and LittleFS Support
ext2, LittleFS and FatFS support; FCB (Flash Circular Buffer) for memory constrained applications.

Powerful multi-backend logging Framework
Support for log filtering, object dumping, panic mode, multiple backends (memory, networking, filesystem, console, ...) and integration with the shell subsystem.

User friendly and full-featured Shell interface
A multi-instance shell subsystem with user-friendly features such as autocompletion, wildcards, coloring, metakeys (arrows, backspace, ctrl+u, etc.) and history. Support for static commands and dynamic sub-commands.

Settings on non-volatile storage
The settings subsystem gives modules a way to store persistent per-device configuration and runtime state. Settings items are stored as key-value pair strings.

Non-volatile storage (NVS)
NVS allows storage of binary blobs, strings, integers, longs, and any combination of these.

Native port
Native sim allows running Zephyr as a Linux application with support for various subsystems and networking.

1.3 Community Support

Community support is provided via mailing lists and Discord; see the Resources below for details.

1.4 Resources

Here's a quick summary of resources to help you find your way around:

1.4.1 Getting Started

- Zephyr Documentation
- Getting Started Guide
- Tips when asking for help
- Code samples

1.4.2 Code and Development

- Source Code Repository
- Releases
- Contribution Guide

1.4.3 Community and Support

- Discord Server for real-time community discussions
- User mailing list (users@lists.zephyrproject.org)
- Developer mailing list (devel@lists.zephyrproject.org)
- Other project mailing lists
- Project Wiki

1.4.4 Issue Tracking and Security

- GitHub Issues
- Security documentation
- Security Advisories Repository
- Report security vulnerabilities at vulnerabilities@zephyrproject.org

1.4.5 Additional Resources

- Zephyr Project Website
- Zephyr Tech Talks

1.5 Fundamental Terms and Concepts

See *Glossary of Terms*

Chapter 2

Developing with Zephyr

2.1 Getting Started Guide

Follow this guide to:

- Set up a command-line Zephyr development environment on Ubuntu, macOS, or Windows (instructions for other Linux distributions are discussed in *Install Linux Host Dependencies*)
- Get the source code
- Build, flash, and run a sample application

2.1.1 Select and Update OS

Click the operating system you are using.

Ubuntu

This guide covers Ubuntu version 20.04 LTS and later.

```
sudo apt update
sudo apt upgrade
```

macOS

On macOS Mojave or later, select *System Preferences* > *Software Update*. Click *Update Now* if necessary.

On other versions, see this Apple support topic.

Windows

Select *Start* > *Settings* > *Update & Security* > *Windows Update*. Click *Check for updates* and install any that are available.

2.1.2 Install dependencies

Next, you'll install some host dependencies using your package manager.

The current minimum required version for the main dependencies are:

Tool	Min. Version
CMake	3.20.5
Python	3.10
Devicetree compiler	1.4.6

Ubuntu

1. If using an Ubuntu version older than 22.04, it is necessary to add extra repositories to meet the minimum required versions for the main dependencies listed above. In that case, download, inspect and execute the Kitware archive script to add the Kitware APT repository to your sources list. A detailed explanation of `kitware-archive.sh` can be found here kitware third-party apt repository:

```
wget https://apt.kitware.com/kitware-archive.sh
sudo bash kitware-archive.sh
```

2. Use `apt` to install the required dependencies:

```
sudo apt install --no-install-recommends git cmake ninja-build gperf \
  ccache dfu-util device-tree-compiler wget \
  python3-dev python3-pip python3-setuptools python3-tk python3-wheel xz-utils file \
  make gcc gcc-multilib g++-multilib libsdl2-dev libmagic1
```

3. Verify the versions of the main dependencies installed on your system by entering:

```
cmake --version
python3 --version
dtc --version
```

   Check those against the versions in the table in the beginning of this section. Refer to the *Install Linux Host Dependencies* page for additional information on updating the dependencies manually.

macOS

1. Install Homebrew:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/
↪install.sh)"
```

2. After the Homebrew installation script completes, follow the on-screen instructions to add the Homebrew installation to the path.

   - On macOS running on Apple Silicon, this is achieved with:

```
(echo; echo 'eval "$(/opt/homebrew/bin/brew shellenv)"') >> ~/.zprofile
source ~/.zprofile
```

   - On macOS running on Intel, use the command for Apple Silicon, but replace `/opt/homebrew/` with `/usr/local/`.

3. Use `brew` to install the required dependencies:

```
brew install cmake ninja gperf python3 ccache qemu dtc libmagic wget openocd
```

4. Add the Homebrew Python folder to the path, in order to be able to execute `python` and `pip` as well `python3` and `pip3`.

```
(echo; echo 'export PATH="'$(brew --prefix)'/opt/python/libexec/bin:$PATH"')␣
↪>> ~/.zprofile
source ~/.zprofile
```

Windows

> **Note**
>
> Due to issues finding executables, the Zephyr Project doesn't currently support application flashing using the Windows Subsystem for Linux (WSL) (WSL).
>
> Therefore, we don't recommend using WSL when getting started.

These instructions must be run in a `cmd.exe` command prompt terminal window. In modern version of Windows (10 and later) it is recommended to install the Windows Terminal application from the Microsoft Store. The required commands differ on PowerShell.

These instructions rely on Chocolatey. If Chocolatey isn't an option, you can install dependencies from their respective websites and ensure the command line tools are on your *PATH environment variable*.

1. Install chocolatey.
2. Open a `cmd.exe` terminal window as **Administrator**. To do so, press the Windows key, type `cmd.exe`, right-click the *Command Prompt* search result, and choose *Run as Administrator*.
3. Disable global confirmation to avoid having to confirm the installation of individual programs:

   ```
   choco feature enable -n allowGlobalConfirmation
   ```

4. Use choco to install the required dependencies:

   ```
   choco install cmake --installargs 'ADD_CMAKE_TO_PATH=System'
   choco install ninja gperf python311 git dtc-msys2 wget 7zip
   ```

   > **Warning**
   >
   > As of November 2023, Python 3.12 is not recommended for Zephyr development on Windows, as some required Python dependencies may be difficult to install.

5. Close the terminal window.

2.1.3 Get Zephyr and install Python dependencies

Next, clone Zephyr and its *modules* into a new *west* workspace named `zephyrproject`. You'll also install Zephyr's additional Python dependencies.

> **Note**
>
> It is easy to run into Python package incompatibilities when installing dependencies at a system or user level. This situation can happen, for example, if working on multiple Zephyr versions or other projects using Python on the same machine.
>
> For this reason it is suggested to use Python virtual environments.

Ubuntu

Install within virtual environment

1. Use `apt` to install Python `venv` package:

```
sudo apt install python3-venv
```

2. Create a new virtual environment:

```
python3 -m venv ~/zephyrproject/.venv
```

3. Activate the virtual environment:

```
source ~/zephyrproject/.venv/bin/activate
```

Once activated your shell will be prefixed with `(.venv)`. The virtual environment can be deactivated at any time by running `deactivate`.

> **Note**
>
> Remember to activate the virtual environment every time you start working.

4. Install west:

```
pip install west
```

5. Get the Zephyr source code:

```
west init ~/zephyrproject
cd ~/zephyrproject
west update
```

6. Export a *Zephyr CMake package*. This allows CMake to automatically load boilerplate code required for building Zephyr applications.

```
west zephyr-export
```

7. Zephyr's `scripts/requirements.txt` file declares additional Python dependencies. Install them with `pip`.

```
pip install -r ~/zephyrproject/zephyr/scripts/requirements.txt
```

Install globally

1. Install west, and make sure `~/.local/bin` is on your *PATH environment variable*:

```
pip3 install --user -U west
echo 'export PATH=~/.local/bin:"$PATH"' >> ~/.bashrc
source ~/.bashrc
```

2. Get the Zephyr source code:

```
west init ~/zephyrproject
cd ~/zephyrproject
west update
```

3. Export a *Zephyr CMake package*. This allows CMake to automatically load boilerplate code required for building Zephyr applications.

```
west zephyr-export
```

4. Zephyr's `scripts/requirements.txt` file declares additional Python dependencies. Install them with `pip3`.

```
pip3 install --user -r ~/zephyrproject/zephyr/scripts/requirements.txt
```

macOS

Install within virtual environment

1. Create a new virtual environment:

   ```
   python3 -m venv ~/zephyrproject/.venv
   ```

2. Activate the virtual environment:

   ```
   source ~/zephyrproject/.venv/bin/activate
   ```

   Once activated your shell will be prefixed with `(.venv)`. The virtual environment can be deactivated at any time by running `deactivate`.

   Note

   Remember to activate the virtual environment every time you start working.

3. Install west:

   ```
   pip install west
   ```

4. Get the Zephyr source code:

   ```
   west init ~/zephyrproject
   cd ~/zephyrproject
   west update
   ```

5. Export a *Zephyr CMake package*. This allows CMake to automatically load boilerplate code required for building Zephyr applications.

   ```
   west zephyr-export
   ```

6. Zephyr's `scripts/requirements.txt` file declares additional Python dependencies. Install them with `pip`.

   ```
   pip install -r ~/zephyrproject/zephyr/scripts/requirements.txt
   ```

Install globally

1. Install west:

   ```
   pip3 install -U west
   ```

2. Get the Zephyr source code:

   ```
   west init ~/zephyrproject
   cd ~/zephyrproject
   west update
   ```

3. Export a *Zephyr CMake package*. This allows CMake to automatically load boilerplate code required for building Zephyr applications.

   ```
   west zephyr-export
   ```

4. Zephyr's `scripts/requirements.txt` file declares additional Python dependencies. Install them with `pip3`.

```
pip3 install -r ~/zephyrproject/zephyr/scripts/requirements.txt
```

Windows

Install within virtual environment

1. Open a `cmd.exe` terminal window **as a regular user**
2. Create a new virtual environment:

```
cd %HOMEPATH%
python -m venv zephyrproject\.venv
```

3. Activate the virtual environment:

```
zephyrproject\.venv\Scripts\activate.bat
```

Once activated your shell will be prefixed with `(.venv)`. The virtual environment can be deactivated at any time by running `deactivate`.

> **Note**
>
> Remember to activate the virtual environment every time you start working.

4. Install west:

```
pip install west
```

5. Get the Zephyr source code:

```
west init zephyrproject
cd zephyrproject
west update
```

6. Export a *Zephyr CMake package*. This allows CMake to automatically load boilerplate code required for building Zephyr applications.

```
west zephyr-export
```

7. Zephyr's `scripts\requirements.txt` file declares additional Python dependencies. Install them with `pip`.

```
pip install -r %HOMEPATH%\zephyrproject\zephyr\scripts\requirements.txt
```

Install globally

1. Open a `cmd.exe` terminal window **as a regular user**
2. Install west:

```
pip3 install -U west
```

3. Get the Zephyr source code:

```
cd %HOMEPATH%
west init zephyrproject
cd zephyrproject
west update
```

4. Export a *Zephyr CMake package*. This allows CMake to automatically load boilerplate code required for building Zephyr applications.

```
west zephyr-export
```

5. Zephyr's `scripts\requirements.txt` file declares additional Python dependencies. Install them with `pip3`.

```
pip3 install -r %HOMEPATH%\zephyrproject\zephyr\scripts\requirements.txt
```

2.1.4 Install the Zephyr SDK

The *Zephyr Software Development Kit (SDK)* contains toolchains for each of Zephyr's supported architectures, which include a compiler, assembler, linker and other programs required to build Zephyr applications.

It also contains additional host tools, such as custom QEMU and OpenOCD builds that are used to emulate, flash and debug Zephyr applications.

Note

You can change `0.16.8` to another version in the instructions below if needed; the Zephyr SDK Releases page contains all available SDK releases.

Note

If you want to uninstall the SDK, you may simply remove the directory where you installed it.

Ubuntu

1. Download and verify the Zephyr SDK bundle:

```
cd ~
wget https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.16.8/
↪zephyr-sdk-0.16.8_linux-x86_64.tar.xz
wget -O - https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.
↪16.8/sha256.sum | shasum --check --ignore-missing
```

   If your host architecture is 64-bit ARM (for example, Raspberry Pi), replace `x86_64` with `aarch64` in order to download the 64-bit ARM Linux SDK.

2. Extract the Zephyr SDK bundle archive:

```
tar xvf zephyr-sdk-0.16.8_linux-x86_64.tar.xz
```

Note

It is recommended to extract the Zephyr SDK bundle at one of the following locations:

- `$HOME`
- `$HOME/.local`
- `$HOME/.local/opt`
- `$HOME/bin`
- `/opt`
- `/usr/local`

The Zephyr SDK bundle archive contains the zephyr-sdk-<version> directory and, when extracted under $HOME, the resulting installation path will be $HOME/zephyr-sdk-<version>.

3. Run the Zephyr SDK bundle setup script:

```
cd zephyr-sdk-0.16.8
./setup.sh
```

Note

You only need to run the setup script once after extracting the Zephyr SDK bundle.

You must rerun the setup script if you relocate the Zephyr SDK bundle directory after the initial setup.

4. Install udev rules, which allow you to flash most Zephyr boards as a regular user:

```
sudo cp ~/zephyr-sdk-0.16.8/sysroots/x86_64-pokysdk-linux/usr/share/openocd/
↪contrib/60-openocd.rules /etc/udev/rules.d
sudo udevadm control --reload
```

macOS

1. Download and verify the Zephyr SDK bundle:

```
cd ~
curl -L -O https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.
↪16.8/zephyr-sdk-0.16.8_macos-x86_64.tar.xz
curl -L https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.16.
↪8/sha256.sum | shasum --check --ignore-missing
```

If your host architecture is 64-bit ARM (Apple Silicon), replace x86_64 with aarch64 in order to download the 64-bit ARM macOS SDK.

2. Extract the Zephyr SDK bundle archive:

```
tar xvf zephyr-sdk-0.16.8_macos-x86_64.tar.xz
```

Note

It is recommended to extract the Zephyr SDK bundle at one of the following locations:

- $HOME
- $HOME/.local
- $HOME/.local/opt
- $HOME/bin
- /opt
- /usr/local

The Zephyr SDK bundle archive contains the zephyr-sdk-<version> directory and, when extracted under $HOME, the resulting installation path will be $HOME/zephyr-sdk-<version>.

3. Run the Zephyr SDK bundle setup script:

```
cd zephyr-sdk-0.16.8
./setup.sh
```

> **Note**
>
> You only need to run the setup script once after extracting the Zephyr SDK bundle.
>
> You must rerun the setup script if you relocate the Zephyr SDK bundle directory after the initial setup.

Windows

1. Open a `cmd.exe` terminal window **as a regular user**
2. Download the Zephyr SDK bundle:

   ```
   cd %HOMEPATH%
   wget https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.16.8/
   ↪zephyr-sdk-0.16.8_windows-x86_64.7z
   ```

3. Extract the Zephyr SDK bundle archive:

   ```
   7z x zephyr-sdk-0.16.8_windows-x86_64.7z
   ```

   > **Note**
   >
   > It is recommended to extract the Zephyr SDK bundle at one of the following locations:
   >
   > - `%HOMEPATH%`
   > - `%PROGRAMFILES%`
   >
   > The Zephyr SDK bundle archive contains the `zephyr-sdk-<version>` directory and, when extracted under `%HOMEPATH%`, the resulting installation path will be `%HOMEPATH%\zephyr-sdk-<version>`.

4. Run the Zephyr SDK bundle setup script:

   ```
   cd zephyr-sdk-0.16.8
   setup.cmd
   ```

   > **Note**
   >
   > You only need to run the setup script once after extracting the Zephyr SDK bundle.
   >
   > You must rerun the setup script if you relocate the Zephyr SDK bundle directory after the initial setup.

2.1.5 Build the Blinky Sample

> **Note**
>
> blinky is compatible with most, but not all, boards. If your board does not meet Blinky's blinky-sample-requirements, then hello_world is a good alternative.
>
> If you are unsure what name west uses for your board, `west boards` can be used to obtain a list of all boards Zephyr supports.

Build the blinky with *west build*, changing `<your-board-name>` appropriately for your board:

Ubuntu

```
cd ~/zephyrproject/zephyr
west build -p always -b <your-board-name> samples/basic/blinky
```

macOS

```
cd ~/zephyrproject/zephyr
west build -p always -b <your-board-name> samples/basic/blinky
```

Windows

```
cd %HOMEPATH%\zephyrproject\zephyr
west build -p always -b <your-board-name> samples\basic\blinky
```

The `-p always` option forces a pristine build, and is recommended for new users. Users may also use the `-p auto` option, which will use heuristics to determine if a pristine build is required, such as when building another sample.

Note

A board may contain one or multiple SoCs, Also, each SoC may contain one or more CPU clusters. When building for such boards it is necessary to specify the SoC or CPU cluster for which the sample must be built. For example to build blinky for the cpuapp core on the nRF5340DK the board must be provided as: `nrf5340dk/nrf5340/cpuapp`. See also *Board terminology* for more details.

2.1.6 Flash the Sample

Connect your board, usually via USB, and turn it on if there's a power switch. If in doubt about what to do, check your board's page in boards.

Then flash the sample using *west flash*:

```
west flash
```

You may need to install additional *host tools* required by your board. The `west flash` command will print an error if any required dependencies are missing.

If you're using blinky, the LED will start to blink as shown in this figure:

2.1.7 Next Steps

Here are some next steps for exploring Zephyr:

- Try other samples-and-demos
- Learn about *Application Development* and the *west* tool
- Find out about west's *flashing and debugging* features, or more about *Flashing and Hardware Debugging* in general
- Check out *Beyond the Getting Started Guide* for additional setup alternatives and ideas
- Discover *Resources* for getting help from the Zephyr community

2.1.8 Troubleshooting Installation

Here are some tips for fixing some issues related to the installation process.

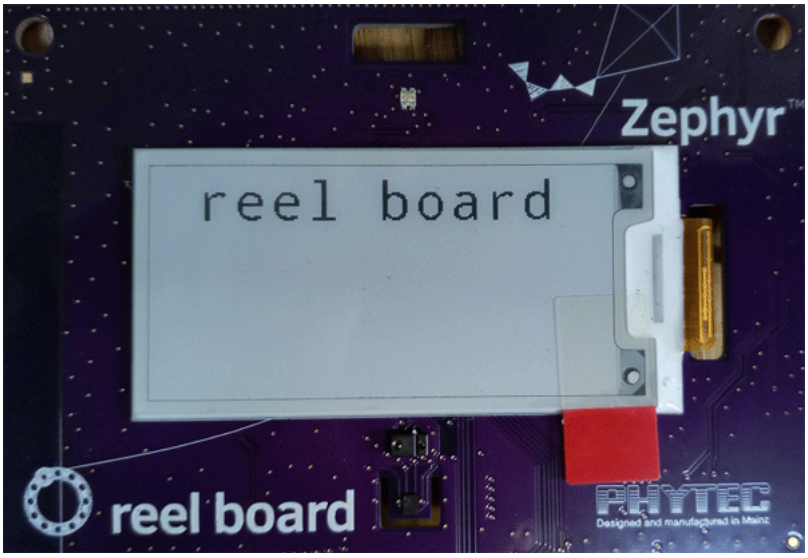


Fig. 1: Phytec reel_board running blinky

Double Check the Zephyr SDK Variables When Updating

When updating Zephyr SDK, check whether the *ZEPHYR_TOOLCHAIN_VARIANT* or *ZEPHYR_SDK_INSTALL_DIR* environment variables are already set. See *Updating the Zephyr SDK toolchain* for more information.

For more information about these environment variables in Zephyr, see *Important Environment Variables*.

2.1.9 Asking for Help

You can ask for help on a mailing list or on Discord. Please send bug reports and feature requests to GitHub.

- **Mailing Lists**: users@lists.zephyrproject.org is usually the right list to ask for help. Search archives and sign up here.
- **Discord**: You can join with this Discord invite.
- **GitHub**: Use GitHub issues for bugs and feature requests.

How to Ask

Important

Please search this documentation and the mailing list archives first. Your question may have an answer there.

Don't just say "this isn't working" or ask "is this working?". Include as much detail as you can about:

1. What you want to do
2. What you tried (commands you typed, etc.)
3. What happened (output of each command, etc.)

Use Copy/Paste

Please **copy/paste text** instead of taking a picture or a screenshot of it. Text includes source code, terminal commands, and their output.

Doing this makes it easier for people to help you, and also helps other users search the archives. Unnecessary screenshots exclude vision impaired developers; some are major Zephyr contributors. Accessibility has been recognized as a basic human right by the United Nations.

When copy/pasting more than 5 lines of computer text into Discord or Github, create a snippet using three backticks to delimit the snippet.

2.2 Beyond the Getting Started Guide

The *Getting Started Guide* gives a straight-forward path to set up your Linux, macOS, or Windows environment for Zephyr development. In this document, we delve deeper into Zephyr development setup issues and alternatives.

2.2.1 Python and pip

Python 3 and its package manager, pip[1], are used extensively by Zephyr to install and run scripts required to compile and run Zephyr applications, set up and maintain the Zephyr development environment, and build project documentation.

Depending on your operating system, you may need to provide the `--user` flag to the `pip3` command when installing new packages. This is documented throughout the instructions. See Installing Packages in the Python Packaging User Guide for more information about pip[1], including information on -\-user.

- On Linux, make sure `~/.local/bin` is at the front of your *PATH environment variable*, or programs installed with `--user` won't be found. Installing with `--user` avoids conflicts between pip and the system package manager, and is the default on Debian-based distributions.
- On macOS, Homebrew disables -\-user.
- On Windows, see the Installing Packages information on `--user` if you require using this option.

On all operating systems, pip's `-U` flag installs or updates the package if the package is already installed locally but a more recent version is available. It is good practice to use this flag if the latest version of a package is required. (Check the scripts/requirements.txt file to see if a specific Python package version is expected.)

2.2.2 Advanced Platform Setup

Here are some alternative instructions for more advanced platform setup configurations for supported development platforms:

[1] pip is Python's package installer. Its `install` command first tries to reuse packages and package dependencies already installed on your computer. If that is not possible, `pip install` downloads them from the Python Package Index (PyPI) on the Internet.
The package versions requested by Zephyr's `requirements.txt` may conflict with other requirements on your system, in which case you may want to set up a virtualenv for Zephyr development.

Install Linux Host Dependencies

Documentation is available for these Linux distributions:

- Ubuntu
- Fedora
- Clear Linux
- Arch Linux

For distributions that are not based on rolling releases, some of the requirements and dependencies may not be met by your package manager. In that case please follow the additional instructions that are provided to find software from sources other than the package manager.

> **Note**
>
> If you're working behind a corporate firewall, you'll likely need to configure a proxy for accessing the internet, if you haven't done so already. While some tools use the environment variables `http_proxy` and `https_proxy` to get their proxy settings, some use their own configuration files, most notably `apt` and `git`.

Update Your Operating System Ensure your host system is up to date.

Ubuntu

```
sudo apt-get update
sudo apt-get upgrade
```

Fedora

```
sudo dnf upgrade
```

Clear Linux

```
sudo swupd update
```

Arch Linux

```
sudo pacman -Syu
```

Install Requirements and Dependencies Note that both Ninja and Make are installed with these instructions; you only need one.

Ubuntu

```
sudo apt-get install --no-install-recommends git cmake ninja-build gperf \
  ccache dfu-util device-tree-compiler wget \
  python3-dev python3-pip python3-setuptools python3-tk python3-wheel xz-utils file \
  make gcc gcc-multilib g++-multilib libsdl2-dev libmagic1
```

Fedora

```
sudo dnf group install "Development Tools" "C Development Tools and Libraries"
sudo dnf install cmake ninja-build gperf dfu-util dtc wget which \
  python3-pip python3-tkinter xz file python3-devel SDL2-devel
```

Clear Linux

```
sudo swupd bundle-add c-basic dev-utils dfu-util dtc \
  os-core-dev python-basic python3-basic python3-tcl
```

The Clear Linux focus is on *native* performance and security and not cross-compilation. For that reason it uniquely exports by default to the *environment* of all users a list of compiler and linker flags. Zephyr's CMake build system will either warn or fail because of these. To clear the C/C++ flags among these and fix the Zephyr build, run the following command as root then log out and back in:

```
echo 'unset CFLAGS CXXFLAGS' >> /etc/profile.d/unset_cflags.sh
```

Note this command unsets the C/C++ flags for *all users on the system*. Each Linux distribution has a unique, relatively complex and potentially evolving sequence of bash initialization files sourcing each other and Clear Linux is no exception. If you need a more flexible solution, start by looking at the logic in `/usr/share/defaults/etc/profile`.

Arch Linux

```
sudo pacman -S git cmake ninja gperf ccache dfu-util dtc wget \
    python-pip python-setuptools python-wheel tk xz file make
```

CMake A *recent CMake version* is required. Check what version you have by using `cmake --version`. If you have an older version, there are several ways of obtaining a more recent one:

- On Ubuntu, you can follow the instructions for adding the kitware third-party apt repository to get an updated version of cmake using apt.
- Download and install a packaged cmake from the CMake project site. (Note this won't uninstall the previous version of cmake.)

  ```
  cd ~
  wget https://github.com/Kitware/CMake/releases/download/v3.21.1/cmake-3.21.1-Linux-x86_
  ↪64.sh
  chmod +x cmake-3.21.1-Linux-x86_64.sh
  sudo ./cmake-3.21.1-Linux-x86_64.sh --skip-license --prefix=/usr/local
  hash -r
  ```

  The `hash -r` command may be necessary if the installation script put cmake into a new location on your PATH.

- Download and install from the pre-built binaries provided by the CMake project itself in the CMake Downloads page. For example, to install version 3.21.1 in `~/bin/cmake`:

  ```
  mkdir $HOME/bin/cmake && cd $HOME/bin/cmake
  wget https://github.com/Kitware/CMake/releases/download/v3.21.1/cmake-3.21.1-Linux-x86_
  ↪64.sh
  yes | sh cmake-3.21.1-Linux-x86_64.sh | cat
  echo "export PATH=$PWD/cmake-3.21.1-Linux-x86_64/bin:\$PATH" >> $HOME/.zephyrrc
  ```

- Use `pip3`:

  ```
  pip3 install --user cmake
  ```

  Note this won't uninstall the previous version of cmake and will install the new cmake into your ~/.local/bin folder so you'll need to add ~/.local/bin to your PATH. (See *Python and pip* for details.)

- Check your distribution's beta or unstable release package library for an update.
- On Ubuntu you can also use snap to get the latest version available:

```
sudo snap install cmake
```

After updating cmake, verify that the newly installed cmake is found using `cmake --version`. You might also want to uninstall the CMake provided by your package manager to avoid conflicts. (Use `whereis cmake` to find other installed versions.)

DTC (Device Tree Compiler) A *recent DTC version* is required. Check what version you have by using `dtc --version`. If you have an older version, either install a more recent one by building from source, or use the one that is bundled in the *Zephyr SDK* by installing it.

Python A *modern Python 3 version* is required. Check what version you have by using `python3 --version`.

If you have an older version, you will need to install a more recent Python 3. You can build from source, or use a backport from your distribution's package manager channels if one is available. Isolating this Python in a virtual environment is recommended to avoid interfering with your system Python.

Install the Zephyr Software Development Kit (SDK) The Zephyr Software Development Kit (SDK) contains toolchains for each of Zephyr's supported architectures. It also includes additional host tools, such as custom QEMU and OpenOCD.

Use of the Zephyr SDK is highly recommended and may even be required under certain conditions (for example, running tests in QEMU for some architectures).

The Zephyr SDK supports the following target architectures:

- ARC (32-bit and 64-bit; ARCv1, ARCv2, ARCv3)
- ARM (32-bit and 64-bit; ARMv6, ARMv7, ARMv8; A/R/M Profiles)
- MIPS (32-bit and 64-bit)
- Nios II
- RISC-V (32-bit and 64-bit; RV32I, RV32E, RV64I)
- x86 (32-bit and 64-bit)
- Xtensa

Follow these steps to install the Zephyr SDK:

1. Download and verify the Zephyr SDK bundle:

   ```
   wget https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.16.8/
   ↪zephyr-sdk-0.16.8_linux-x86_64.tar.xz
   wget -O - https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.
   ↪16.8/sha256.sum | shasum --check --ignore-missing
   ```

   You can change `0.16.8` to another version if needed; the Zephyr SDK Releases page contains all available SDK releases.

   If your host architecture is 64-bit ARM (for example, Raspberry Pi), replace `x86_64` with `aarch64` in order to download the 64-bit ARM Linux SDK.

2. Extract the Zephyr SDK bundle archive:

   ```
   cd <sdk download directory>
   tar xvf zephyr-sdk-0.16.8_linux-x86_64.tar.xz
   ```

3. Run the Zephyr SDK bundle setup script:

```
cd zephyr-sdk-0.16.8
./setup.sh
```

If this fails, make sure Zephyr's dependencies were installed as described in *Install Requirements and Dependencies*.

If you want to uninstall the SDK, remove the directory where you installed it. If you relocate the SDK directory, you need to re-run the setup script.

Note

It is recommended to extract the Zephyr SDK bundle at one of the following locations:

- `$HOME`
- `$HOME/.local`
- `$HOME/.local/opt`
- `$HOME/bin`
- `/opt`
- `/usr/local`

The Zephyr SDK bundle archive contains the `zephyr-sdk-<version>` directory and, when extracted under `$HOME`, the resulting installation path will be `$HOME/zephyr-sdk-<version>`.

If you install the Zephyr SDK outside any of these locations, you must register the Zephyr SDK in the CMake package registry by running the setup script, or set *ZEPHYR_SDK_INSTALL_DIR* to point to the Zephyr SDK installation directory.

You can also use *ZEPHYR_SDK_INSTALL_DIR* for pointing to a directory containing multiple Zephyr SDKs, allowing for automatic toolchain selection. For example, `ZEPHYR_SDK_INSTALL_DIR=/company/tools`, where the `company/tools` folder contains the following subfolders:

- `/company/tools/zephyr-sdk-0.13.2`
- `/company/tools/zephyr-sdk-a.b.c`
- `/company/tools/zephyr-sdk-x.y.z`

This allows the Zephyr build system to choose the correct version of the SDK, while allowing multiple Zephyr SDKs to be grouped together at a specific path.

Building on Linux without the Zephyr SDK The Zephyr SDK is provided for convenience and ease of use. It provides toolchains for all Zephyr target architectures, and does not require any extra flags when building applications or running tests. In addition to cross-compilers, the Zephyr SDK also provides prebuilt host tools. It is, however, possible to build without the SDK's toolchain by using another toolchain as described in the *Toolchains* section.

As already noted above, the SDK also includes prebuilt host tools. To use the SDK's prebuilt host tools with a toolchain from another source, you must set the *ZEPHYR_SDK_INSTALL_DIR* environment variable to the Zephyr SDK installation directory. To build without the Zephyr SDK's prebuilt host tools, the *ZEPHYR_SDK_INSTALL_DIR* environment variable must be unset.

To make sure this variable is unset, run:

```
unset ZEPHYR_SDK_INSTALL_DIR
```

macOS alternative setup instructions

Important note about Gatekeeper Starting with macOS 10.15 Catalina, applications launched from the macOS Terminal application (or any other terminal emulator) are subject to the same system security policies that are applied to applications launched from the Dock. This means that if you download executable binaries using a web browser, macOS will not let you execute those from the Terminal by default. In order to get around this issue you can take two different approaches:

- Run `xattr -r -d com.apple.quarantine /path/to/folder` where `path/to/folder` is the path to the enclosing folder where the executables you want to run are located.
- Open *System Preferences* ‣ *Security and Privacy* ‣ *Privacy* and then scroll down to "Developer Tools". Then unlock the lock to be able to make changes and check the checkbox corresponding to your terminal emulator of choice. This will apply to any executable being launched from such terminal program.

Note that this section does **not** apply to executables installed with Homebrew, since those are automatically un-quarantined by `brew` itself. This is however relevant for most *Toolchains*.

Additional notes for MacPorts users While MacPorts is not officially supported in this guide, it is possible to use MacPorts instead of Homebrew to get all the required dependencies on macOS. Note also that you may need to install `rust` and `cargo` for the Python dependencies to install correctly.

Windows alternative setup instructions

Windows 10 WSL (Windows Subsystem for Linux) If you are running a recent version of Windows 10 you can make use of the built-in functionality to natively run Ubuntu binaries directly on a standard command-prompt. This allows you to use software such as the *Zephyr SDK* without setting up a virtual machine.

Warning

Windows 10 version 1803 has an issue that will cause CMake to not work properly and is fixed in version 1809 (and later). More information can be found in Zephyr Issue 10420.

1. Install the Windows Subsystem for Linux (WSL).

   Note

   For the Zephyr SDK to function properly you will need Windows 10 build 15002 or greater. You can check which Windows 10 build you are running in the "About your PC" section of the System Settings. If you are running an older Windows 10 build you might need to install the Creator's Update.

2. Follow the Ubuntu instructions in the *Install Linux Host Dependencies* document.

2.2.3 Install a Toolchain

Zephyr binaries are compiled and linked by a *toolchain* comprised of a cross-compiler and related tools which are different from the compiler and tools used for developing software that runs natively on your host operating system.

You can install the *Zephyr SDK* to get toolchains for all supported architectures, or install an *alternate toolchain* recommended by the SoC vendor or a specific board (check your specific board-level documentation).

You can configure the Zephyr build system to use a specific toolchain by setting *environment variables* such as *ZEPHYR_TOOLCHAIN_VARIANT* to a supported value, along with additional variable(s) specific to the toolchain variant.

2.2.4 Updating the Zephyr SDK toolchain

When updating Zephyr SDK, check whether the *ZEPHYR_TOOLCHAIN_VARIANT* or *ZEPHYR_SDK_INSTALL_DIR* environment variables are already set.

- If the variables are not set, the latest compatible version of Zephyr SDK will be selected by default. Proceed to next step without making any changes.
- If *ZEPHYR_TOOLCHAIN_VARIANT* is set, the corresponding toolchain will be selected at build time. Zephyr SDK is identified by the value `zephyr`. If the *ZEPHYR_TOOLCHAIN_VARIANT* environment variable is not `zephyr`, then either unset it or change its value to `zephyr` to make sure Zephyr SDK is selected.
- If the *ZEPHYR_SDK_INSTALL_DIR* environment variable is set, it will override the default lookup location for Zephyr SDK. If you install Zephyr SDK to one of the *recommended locations*, you can unset this variable. Otherwise, set it to your chosen install location.

For more information about these environment variables in Zephyr, see *Important Environment Variables*.

2.2.5 Cloning the Zephyr Repositories

The Zephyr project source is maintained in the GitHub zephyr repo. External modules used by Zephyr are found in the parent GitHub Zephyr project. Because of these dependencies, it's convenient to use the Zephyr-created *west* tool to fetch and manage the Zephyr and external module source code. See *Basics* for more details.

Once your development tools are installed, use *West (Zephyr's meta-tool)* to create, initialize, and download sources from the zephyr and external module repos. We'll use the name `zephyrproject`, but you can choose any name that does not contain a space anywhere in the path.

```
west init zephyrproject
cd zephyrproject
west update
```

The `west update` command fetches and keeps *Modules (External projects)* in the `zephyrproject` folder in sync with the code in the local zephyr repo.

Warning

You must run `west update` any time the `zephyr/west.yml` changes, caused, for example, when you pull the `zephyr` repository, switch branches in it, or perform a `git bisect` inside of it.

Keeping Zephyr updated

To update the Zephyr project source code, you need to get the latest changes via `git`. Afterwards, run `west update` as mentioned in the previous paragraph.

```
# replace zephyrproject with the path you gave west init
cd zephyrproject/zephyr
git pull
west update
```

2.2.6 Export Zephyr CMake package

The *Zephyr CMake Package* can be exported to CMake's user package registry if it has not already been done as part of *Getting Started Guide*.

2.2.7 Board Aliases

Developers who work with multiple boards may find explicit board names cumbersome and want to use aliases for common targets. This is supported by a CMake file with content like this:

```
# Variable foo_BOARD_ALIAS=bar replaces BOARD=foo with BOARD=bar and
# sets BOARD_ALIAS=foo in the CMake cache.
set(pca10028_BOARD_ALIAS nrf51dk/nrf51822)
set(pca10056_BOARD_ALIAS nrf52840dk/nrf52840)
set(k64f_BOARD_ALIAS frdm_k64f)
set(sltb004a_BOARD_ALIAS efr32mg_sltb004a)
```

and specifying its location in *ZEPHYR_BOARD_ALIASES*. This enables use of aliases `pca10028` in contexts like `cmake -DBOARD=pca10028` and `west -b pca10028`.

2.2.8 Build and Run an Application

You can build, flash, and run Zephyr applications on real hardware using a supported host system. Depending on your operating system, you can also run it in emulation with QEMU, or as a native application with native_sim. Additional information about building applications can be found in the *Building an Application* section.

Build Blinky

Let's build the blinky sample application.

Zephyr applications are built to run on specific hardware, called a "board"[2]. We'll use the Phytec reel_board here, but you can change the `reel_board` build target to another value if you have a different board. See boards or run `west boards` from anywhere inside the `zephyrproject` directory for a list of supported boards.

1. Go to the zephyr repository:

   ```
   cd zephyrproject/zephyr
   ```

2. Build the blinky sample for the `reel_board`:

   ```
   west build -b reel_board samples/basic/blinky
   ```

[2] This has become something of a misnomer over time. While the target can be, and often is, a microprocessor running on its own dedicated hardware board, Zephyr also supports using QEMU to run targets built for other architectures in emulation, targets which produce native host system binaries that implement Zephyr's driver interfaces with POSIX APIs, and even running different Zephyr-based binaries on CPU cores of differing architectures on the same physical chip. Each of these hardware configurations is called a "board," even though that doesn't always make perfect sense in context.

The main build products will be in `build/zephyr`; `build/zephyr/zephyr.elf` is the blinky application binary in ELF format. Other binary formats, disassembly, and map files may be present depending on your board.

The other sample applications in the samples folder are documented in samples-and-demos.

> **Note**
>
> If you want to reuse an existing build directory for another board or application, you need to add the parameter `-p=auto` to `west build` to clean out settings and artifacts from the previous build.

Run the Application by Flashing to a Board

Most hardware boards supported by Zephyr can be flashed by running `west flash`. This may require board-specific tool installation and configuration to work properly.

See *Run an Application* and your specific board's documentation in boards for additional details.

Setting udev rules

Flashing a board requires permission to directly access the board hardware, usually managed by installation of the flashing tools. On Linux systems, if the `west flash` command fails, you likely need to define udev rules to grant the needed access permission.

Udev is a device manager for the Linux kernel and the udev daemon handles all user space events raised when a hardware device is added (or removed) from the system. We can add a rules file to grant access permission by non-root users to certain USB-connected devices.

The OpenOCD (On-Chip Debugger) project conveniently provides a rules file that defined board-specific rules for most Zephyr-supported arm-based boards, so we recommend installing this rules file by downloading it from their sourceforge repo, or if you've installed the Zephyr SDK there is a copy of this rules file in the SDK folder:

- Either download the OpenOCD rules file and copy it to the right location:

```
wget -O 60-openocd.rules https://sf.net/p/openocd/code/ci/master/tree/contrib/60-
↪openocd.rules?format=raw
sudo cp 60-openocd.rules /etc/udev/rules.d
```

- or copy the rules file from the Zephyr SDK folder:

```
sudo cp ${ZEPHYR_SDK_INSTALL_DIR}/sysroots/x86_64-pokysdk-linux/usr/share/openocd/
↪contrib/60-openocd.rules /etc/udev/rules.d
```

Then, in either case, ask the udev daemon to reload these rules:

```
sudo udevadm control --reload
```

Unplug and plug in the USB connection to your board, and you should have permission to access the board hardware for flashing. Check your board-specific documentation (boards) for further information if needed.

Run the Application in QEMU

On Linux and macOS, you can run Zephyr applications via emulation on your host system using QEMU when targeting either the x86 or ARM Cortex-M3 architectures. (QEMU is included with the Zephyr SDK installation.)

On Windows, you need to install QEMU manually from Download QEMU. After installation, add path to QEMU installation folder to PATH environment variable. To enable QEMU in Test Runner (Twister) on Windows, *set the environment variable* `QEMU_BIN_PATH` to the path of QEMU installation folder.

For example, you can build and run the hello_world sample using the x86 emulation board configuration (`qemu_x86`), with:

```
# From the root of the zephyr repository
west build -b qemu_x86 samples/hello_world
west build -t run
```

To exit QEMU, type `Ctrl-a`, then `x`.

Use `qemu_cortex_m3` to target an emulated Arm Cortex-M3 sample.

Run a Sample Application natively (Linux)

You can compile some samples to run as host programs on Linux. See native_sim for more information. On 64-bit host operating systems, you need to install a 32-bit C library, or build targeting native_sim/native/64.

First, build Hello World for `native_sim`.

```
# From the root of the zephyr repository
west build -b native_sim samples/hello_world
```

Next, run the application.

```
west build -t run
# or just run zephyr.exe directly:
./build/zephyr/zephyr.exe
```

Press `Ctrl-C` to exit.

You can run `./build/zephyr/zephyr.exe --help` to get a list of available options.

This executable can be instrumented using standard tools, such as gdb or valgrind.

2.3 Environment Variables

Various pages in this documentation refer to setting Zephyr-specific environment variables. This page describes how.

2.3.1 Setting Variables

Option 1: Just Once

To set the environment variable `MY_VARIABLE` to `foo` for the lifetime of your current terminal window:

Linux/macOS

```
export MY_VARIABLE=foo
```

Windows

```
set MY_VARIABLE=foo
```

Warning

This is best for experimentation. If you close your terminal window, use another terminal window or tab, restart your computer, etc., this setting will be lost forever.

Using options 2 or 3 is recommended if you want to keep using the setting.

Option 2: In all Terminals

Linux/macOS

Add the `export MY_VARIABLE=foo` line to your shell's startup script in your home directory. For Bash, this is usually `~/.bashrc` on Linux or `~/.bash_profile` on macOS. Changes in these startup scripts don't affect shell instances already started; try opening a new terminal window to get the new settings.

Windows

You can use the `setx` program in `cmd.exe` or the third-party RapidEE program.

To use `setx`, type this command, then close the terminal window. Any new `cmd.exe` windows will have `MY_VARIABLE` set to `foo`.

```
setx MY_VARIABLE foo
```

To install RapidEE, a freeware graphical environment variable editor, using Chocolatey in an Administrator command prompt:

```
choco install rapidee
```

You can then run `rapidee` from your terminal to launch the program and set environment variables. Make sure to use the "User" environment variables area – otherwise, you have to run RapidEE as administrator. Also make sure to save your changes by clicking the Save button at top left before exiting. Settings you make in RapidEE will be available whenever you open a new terminal window.

Option 3: Using `zephyrrc` files

Choose this option if you don't want to make the variable's setting available to all of your terminals, but still want to save the value for loading into your environment when you are using Zephyr.

Linux/macOS

Create a file named `~/.zephyrrc` if it doesn't exist, then add this line to it:

```
export MY_VARIABLE=foo
```

To get this value back into your current terminal environment, **you must run** `source zephyr-env.sh` from the main `zephyr` repository. Among other things, this script sources `~/.zephyrrc`.

The value will be lost if you close the window, etc.; run `source zephyr-env.sh` again to get it back.

Windows

Add the line `set MY_VARIABLE=foo` to the file `%userprofile%\zephyrrc.cmd` using a text editor such as Notepad to save the value.

To get this value back into your current terminal environment, **you must run** `zephyr-env.cmd` in a `cmd.exe` window after changing directory to the main `zephyr` repository. Among other things, this script runs `%userprofile%\zephyrrc.cmd`.

The value will be lost if you close the window, etc.; run `zephyr-env.cmd` again to get it back.

These scripts:

- set *ZEPHYR_BASE* to the location of the zephyr repository
- adds some Zephyr-specific locations (such as zephyr's `scripts` directory) to your *PATH* environment variable
- loads any settings from the `zephyrrc` files described above in *Option 3: Using zephyrrc files*.

You can thus use them any time you need any of these settings.

2.3.2 Zephyr Environment Scripts

You can use the zephyr repository scripts `zephyr-env.sh` (for macOS and Linux) and `zephyr-env.cmd` (for Windows) to load Zephyr-specific settings into your current terminal's environment. To do so, run this command from the zephyr repository:

Linux/macOS

```
source zephyr-env.sh
```

Windows

```
zephyr-env.cmd
```

These scripts:

- set *ZEPHYR_BASE* to the location of the zephyr repository
- adds some Zephyr-specific locations (such as zephyr's `scripts` directory) to your `PATH` environment variable
- loads any settings from the `zephyrrc` files described above in *Option 3: Using zephyrrc files*.

You can thus use them any time you need any of these settings.

2.3.3 Important Environment Variables

Some *Important Build System Variables* can also be set in the environment. Here is a description of some of these important environment variables. This is not a comprehensive list.

`BOARD`
: See *Important Build System Variables*.

`CONF_FILE`
: See *Important Build System Variables*.

`SHIELD`
: See *Shields*.

`ZEPHYR_BASE`
: See *Important Build System Variables*.

`EXTRA_ZEPHYR_MODULES`
: See *Important Build System Variables*.

`ZEPHYR_MODULES`
: See *Important Build System Variables*.

`ZEPHYR_BOARD_ALIASES`
: See *Board Aliases*

The following additional environment variables are significant when configuring the *toolchain* used to build Zephyr applications.

`ZEPHYR_SDK_INSTALL_DIR`
: Path where Zephyr SDK is installed.

`ZEPHYR_TOOLCHAIN_VARIANT`
: The name of the toolchain to use.

`{TOOLCHAIN}_TOOLCHAIN_PATH`
: Path to the toolchain specified by *ZEPHYR_TOOLCHAIN_VARIANT*. For example, if `ZEPHYR_TOOLCHAIN_VARIANT=llvm`, use `LLVM_TOOLCHAIN_PATH`. (Note the capitalization when forming the environment variable name.)

You might need to update some of these variables when you *update the Zephyr SDK toolchain*.

Emulators and boards may also depend on additional programs. The build system will try to locate those programs automatically, but may rely on additional CMake or environment variables to do so. Please consult your emulator's or board's documentation for more information. The following environment variables may be useful in such situations:

`PATH`
: `PATH` is an environment variable used on Unix-like or Microsoft Windows operating systems to specify a set of directories where executable programs are located.

2.4 Application Development

Note

In this document, we'll assume:

- your **application directory**, <app>, is something like `<home>/zephyrproject/app`
- its **build directory** is `<app>/build`

These terms are defined below. On Linux/macOS, <home> is equivalent to ~. On Windows, it's `%userprofile%`.

Keeping your application inside the workspace (`<home>/zephyrproject`) makes it easier to use `west build` and other commands with it. (You can put your application anywhere as long as *ZEPHYR_BASE* is set appropriately, though.)

2.4.1 Overview

Zephyr's build system is based on CMake.

The build system is application-centric, and requires Zephyr-based applications to initiate building the Zephyr source code. The application build controls the configuration and build process of both the application and Zephyr itself, compiling them into a single binary.

The main zephyr repository contains Zephyr's source code, configuration files, and build system. You also likely have installed various *Modules (External projects)* alongside the zephyr repository, which provide third party source code integration.

The files in the **application directory** link Zephyr and any modules with the application. This directory contains all application-specific files, such as application-specific configuration files and source code.

Here are the files in a simple Zephyr application:

```
<app>
├── CMakeLists.txt
├── app.overlay
├── prj.conf
├── VERSION
└── src
    └── main.c
```

These contents are:

- **CMakeLists.txt**: This file tells the build system where to find the other application files, and links the application directory with Zephyr's CMake build system. This link provides features supported by Zephyr's build system, such as board-specific configuration files, the ability to run and debug compiled binaries on real or emulated hardware, and more.
- **app.overlay**: This is a devicetree overlay file that specifies application-specific changes which should be applied to the base devicetree for any board you build for. The purpose of devicetree overlays is usually to configure something about the hardware used by the application.

  The build system looks for `app.overlay` by default, but you can add more devicetree overlays, and other default files are also searched for.

  See *Devicetree* for more information about devicetree.
- **prj.conf**: This is a Kconfig fragment that specifies application-specific values for one or more Kconfig options. These application settings are merged with other settings to produce the final configuration. The purpose of Kconfig fragments is usually to configure the software features used by the application.

  The build system looks for `prj.conf` by default, but you can add more Kconfig fragments, and other default files are also searched for.

  See *Kconfig Configuration* below for more information.
- **VERSION**: A text file that contains several version information fields. These fields let you manage the lifecycle of the application and automate providing the application version when signing application images.

  See *Application version management* for more information about this file and how to use it.
- **main.c**: A source code file. Applications typically contain source files written in C, C++, or assembly language. The Zephyr convention is to place them in a subdirectory of `<app>` named `src`.

Once an application has been defined, you will use CMake to generate a **build directory**, which contains the files you need to build the application and Zephyr, then link them together into a final binary you can run on your board. The easiest way to do this is with *west build*, but you can use CMake directly also. Application build artifacts are always generated in a separate build directory: Zephyr does not support "in-tree" builds.

The following sections describe how to create, build, and run Zephyr applications, followed by more detailed reference material.

2.4.2 Application types

We distinguish three basic types of Zephyr application based on where <app> is located:

Application type	<app> location
repository	zephyr repository
workspace	west workspace where Zephyr is installed
freestanding	other locations

We'll discuss these more below. To learn how the build system supports each type, see *Zephyr CMake Package*.

Zephyr repository application

An application located within the `zephyr` source code repository in a Zephyr *west workspace* is referred to as a Zephyr repository application. In the following example, the hello_world sample is a Zephyr repository application:

```
zephyrproject/
├── .west/
│   └── config
└── zephyr/
    ├── arch/
    ├── boards/
    ├── cmake/
    ├── samples/
    │   ├── hello_world/
    │   └── ...
    ├── tests/
    └── ...
```

Zephyr workspace application

An application located within a *workspace*, but outside the zephyr repository itself, is referred to as a Zephyr workspace application. In the following example, `app` is a Zephyr workspace application:

```
zephyrproject/
├── .west/
│   └── config
├── zephyr/
├── bootloader/
├── modules/
├── tools/
├── <vendor/private-repositories>/
└── applications/
    └── app/
```

Zephyr freestanding application

A Zephyr application located outside of a Zephyr *workspace* is referred to as a Zephyr freestanding application. In the following example, `app` is a Zephyr freestanding application:

```
<home>/
├── zephyrproject/
│     ├── .west/
│     │    └── config
│     ├── zephyr/
│     ├── bootloader/
│     ├── modules/
│     └── ...
│
└── app/
     ├── CMakeLists.txt
     ├── prj.conf
     └── src/
         └── main.c
```

2.4.3 Creating an Application

In Zephyr, you can either use a reference workspace application or create your application by hand.

Using a Reference Workspace Application

The example-application Git repository contains a reference *workspace application*. It is recommended to use it as a reference when creating your own application as described in the following sections.

The example-application repository demonstrates how to use several commonly-used features, such as:

- Custom *board ports*
- Custom *devicetree bindings*
- Custom *device drivers*
- Continuous Integration (CI) setup, including using *twister*
- A custom west *extension command*

Basic example-application Usage The easiest way to get started with the example-application repository within an existing Zephyr workspace is to follow these steps:

```
cd <home>/zephyrproject
git clone https://github.com/zephyrproject-rtos/example-application my-app
```

The directory name `my-app` above is arbitrary: change it as needed. You can now go into this directory and adapt its contents to suit your needs. Since you are using an existing Zephyr workspace, you can use `west build` or any other west commands to build, flash, and debug.

Advanced example-application Usage You can also use the example-application repository as a starting point for building your own customized Zephyr-based software distribution. This lets you do things like:

- remove Zephyr modules you don't need
- add additional custom repositories of your own
- override repositories provided by Zephyr with your own versions

- share the results with others and collaborate further

The example-application repository contains a `west.yml` file and is therefore also a west *manifest repository*. Use this to create a new, customized workspace by following these steps:

```
cd <home>
mkdir my-workspace
cd my-workspace
git clone https://github.com/zephyrproject-rtos/example-application my-manifest-repo
west init -l my-manifest-repo
```

This will create a new workspace with the *T2 topology*, with `my-manifest-repo` as the manifest repository. The `my-workspace` and `my-manifest-repo` names are arbitrary: change them as needed.

Next, customize the manifest repository. The initial contents of this repository will match the example-application's contents when you clone it. You can then edit `my-manifest-repo/west.yml` to your liking, changing the set of repositories in it as you wish. See *Manifest Imports* for many examples of how to add or remove different repositories from your workspace as needed. Make any other changes you need to other files.

When you are satisfied, you can run:

```
west update
```

and your workspace will be ready for use.

If you push the resulting `my-manifest-repo` repository somewhere else, you can share your work with others. For example, let's say you push the repository to `https://git.example.com/my-manifest-repo`. Other people can then set up a matching workspace by running:

```
west init -m https://git.example.com/my-manifest-repo my-workspace
cd my-workspace
west update
```

From now on, you can collaborate on the shared software by pushing changes to the repositories you are using and updating `my-manifest-repo/west.yml` as needed to add and remove repositories, or change their contents.

Creating an Application by Hand

You can follow these steps to create a basic application directory from scratch. However, using the example-application repository or one of Zephyr's samples-and-demos as a starting point is likely to be easier.

1. Create an application directory.

   For example, in a Unix shell or Windows `cmd.exe` prompt:

   ```
   mkdir app
   ```

   Warning

   Building Zephyr or creating an application in a directory with spaces anywhere on the path is not supported. So the Windows path `C:\Users\YourName\app` will work, but `C:\Users\Your Name\app` will not.

2. Create your source code files.

   It's recommended to place all application source code in a subdirectory named `src`. This makes it easier to distinguish between project files and sources.

Continuing the previous example, enter:

```
cd app
mkdir src
```

3. Place your application source code in the `src` sub-directory. For this example, we'll assume you created a file named `src/main.c`.
4. Create a file named `CMakeLists.txt` in the app directory with the following contents:

```
cmake_minimum_required(VERSION 3.20.0)

find_package(Zephyr)
project(my_zephyr_app)

target_sources(app PRIVATE src/main.c)
```

Notes:

- The `cmake_minimum_required()` call is required by CMake. It is also invoked by the Zephyr package on the next line. CMake will error out if its version is older than either the version in your `CMakeLists.txt` or the version number in the Zephyr package.
- `find_package(Zephyr)` pulls in the Zephyr build system, which creates a CMake target named `app` (see *Zephyr CMake Package*). Adding sources to this target is how you include them in the build. The Zephyr package will define `Zephyr-Kernel` as a CMake project and enable support for the `C`, `CXX`, `ASM` languages.
- `project(my_zephyr_app)` defines your application's CMake project. This must be called after `find_package(Zephyr)` to avoid interference with Zephyr's `project(Zephyr-Kernel)`.
- `target_sources(app PRIVATE src/main.c)` is to add your source file to the `app` target. This must come after `find_package(Zephyr)` which defines the target. You can add as many files as you want with `target_sources()`.

5. Create at least one Kconfig fragment for your application (usually named `prj.conf`) and set Kconfig option values needed by your application there. See *Kconfig Configuration*. If no Kconfig options need to be set, create an empty file.
6. Configure any devicetree overlays needed by your application, usually in a file named `app.overlay`. See *Set devicetree overlays*.
7. Set up any other files you may need, such as *twister* configuration files, continuous integration files, documentation, etc.

2.4.4 Important Build System Variables

You can control the Zephyr build system using many variables. This section describes the most important ones that every Zephyr developer should know about.

Note

The variables **BOARD**, **CONF_FILE**, and **DTC_OVERLAY_FILE** can be supplied to the build system in 3 ways (in order of precedence):

- As a parameter to the `west build` or `cmake` invocation via the `-D` command-line switch. If you have multiple overlay files, you should use quotations, `"file1.overlay;file2.overlay"`
- As *Environment Variables*.
- As a `set(<VARIABLE> <VALUE>)` statement in your `CMakeLists.txt`

- `ZEPHYR_BASE`: Zephyr base variable used by the build system. `find_package(Zephyr)` will automatically set this as a cached CMake variable. But `ZEPHYR_BASE` can also be set as an environment variable in order to force CMake to use a specific Zephyr installation.
- `BOARD`: Selects the board that the application's build will use for the default configuration. See boards for built-in boards, and *Board Porting Guide* for information on adding board support.
- `CONF_FILE`: Indicates the name of one or more Kconfig configuration fragment files. Multiple filenames can be separated with either spaces or semicolons. Each file includes Kconfig configuration values that override the default configuration values.

  See *The Initial Configuration* for more information.
- `EXTRA_CONF_FILE`: Additional Kconfig configuration fragment files. Multiple filenames can be separated with either spaces or semicolons. This can be useful in order to leave `CONF_FILE` at its default value, but "mix in" some additional configuration options.
- `DTC_OVERLAY_FILE`: One or more devicetree overlay files to use. Multiple files can be separated with semicolons. See *Set devicetree overlays* for examples and *Introduction to devicetree* for information about devicetree and Zephyr.
- `EXTRA_DTC_OVERLAY_FILE`: Additional devicetree overlay files to use. Multiple files can be separated with semicolons. This can be useful to leave `DTC_OVERLAY_FILE` at its default value, but "mix in" some additional overlay files.
- `SHIELD`: see *Shields*
- `ZEPHYR_MODULES`: A CMake list containing absolute paths of additional directories with source code, Kconfig, etc. that should be used in the application build. See *Modules (External projects)* for details. If you set this variable, it must be a complete list of all modules to use, as the build system will not automatically pick up any modules from west.
- `EXTRA_ZEPHYR_MODULES`: Like `ZEPHYR_MODULES`, except these will be added to the list of modules found via west, instead of replacing it.
- `FILE_SUFFIX`: Optional suffix for filenames that will be added to Kconfig fragments and devicetree overlays (if these files exists, otherwise will fallback to the name without the prefix). See *File Suffixes* for details.

Note

You can use a *Zephyr Build Configuration CMake packages* to share common settings for these variables.

2.4.5 Application CMakeLists.txt

Every application must have a `CMakeLists.txt` file. This file is the entry point, or top level, of the build system. The final `zephyr.elf` image contains both the application and the kernel libraries.

This section describes some of what you can do in your `CMakeLists.txt`. Make sure to follow these steps in order.

1. If you only want to build for one board, add the name of the board configuration for your application on a new line. For example:

   ```
   set(BOARD qemu_x86)
   ```

   Refer to boards for more information on available boards.

The Zephyr build system determines a value for **BOARD** by checking the following, in order (when a BOARD value is found, CMake stops looking further down the list):

- Any previously used value as determined by the CMake cache takes highest precedence. This ensures you don't try to run a build with a different **BOARD** value than you set during the build configuration step.
- Any value given on the CMake command line (directly or indirectly via `west build`) using `-DBOARD=YOUR_BOARD` will be checked for and used next.
- If an *environment variable* `BOARD` is set, its value will then be used.
- Finally, if you set `BOARD` in your application `CMakeLists.txt` as described in this step, this value will be used.

2. If your application uses a configuration file or files other than the usual `prj.conf`, add lines setting the **CONF_FILE** variable to these files appropriately. If multiple filenames are given, separate them by a single space or semicolon. CMake lists can be used to build up configuration fragment files in a modular way when you want to avoid setting **CONF_FILE** in a single place. For example:

```
set(CONF_FILE "fragment_file1.conf")
list(APPEND CONF_FILE "fragment_file2.conf")
```

   See *The Initial Configuration* for more information.

3. If your application uses devicetree overlays, you may need to set *DTC_OVERLAY_FILE*. See *Set devicetree overlays*.

4. If your application has its own kernel configuration options, create a `Kconfig` file in the same directory as your application's `CMakeLists.txt`.

   See *the Kconfig section of the manual* for detailed Kconfig documentation.

   An (unlikely) advanced use case would be if your application has its own unique configuration **options** that are set differently depending on the build configuration.

   If you just want to set application specific **values** for existing Zephyr configuration options, refer to the **CONF_FILE** description above.

   Structure your `Kconfig` file like this:

```
# SPDX-License-Identifier: Apache-2.0

mainmenu "Your Application Name"

# Your application configuration options go here

# Sources Kconfig.zephyr in the Zephyr root directory.
#
# Note: All 'source' statements work relative to the Zephyr root directory (due
# to the $srctree environment variable being set to $ZEPHYR_BASE). If you want
# to 'source' relative to the current Kconfig file instead, use 'rsource' (or a
# path relative to the Zephyr root).
source "Kconfig.zephyr"
```

   Note

   Environment variables in `source` statements are expanded directly, so you do not need to define an `option env="ZEPHYR_BASE"` Kconfig "bounce" symbol. If you use such a symbol, it must have the same name as the environment variable.

   See *Kconfig extensions* for more information.

The `Kconfig` file is automatically detected when placed in the application directory, but it is also possible for it to be found elsewhere if the CMake variable `KCONFIG_ROOT` is set with an absolute path.

5. Specify that the application requires Zephyr on a new line, **after any lines added from the steps above**:

```
find_package(Zephyr)
project(my_zephyr_app)
```

> **Note**
>
> `find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})` can be used if enforcing a specific Zephyr installation by explicitly setting the `ZEPHYR_BASE` environment variable should be supported. All samples in Zephyr supports the `ZEPHYR_BASE` environment variable.

6. Now add any application source files to the 'app' target library, each on their own line, like so:

```
target_sources(app PRIVATE src/main.c)
```

Below is a simple example `CMakeList.txt`:

```
set(BOARD qemu_x86)

find_package(Zephyr)
project(my_zephyr_app)

target_sources(app PRIVATE src/main.c)
```

The Cmake property `HEX_FILES_TO_MERGE` leverages the application configuration provided by Kconfig and CMake to let you merge externally built hex files with the hex file generated when building the Zephyr application. For example:

```
set_property(GLOBAL APPEND PROPERTY HEX_FILES_TO_MERGE
    ${app_bootloader_hex}
    ${PROJECT_BINARY_DIR}/${KERNEL_HEX_NAME}
    ${app_provision_hex})
```

2.4.6 CMakeCache.txt

CMake uses a CMakeCache.txt file as persistent key/value string storage used to cache values between runs, including compile and build options and paths to library dependencies. This cache file is created when CMake is run in an empty build folder.

For more details about the CMakeCache.txt file see the official CMake documentation runningcmake .

2.4.7 Application Configuration

Application Configuration Directory

Zephyr will use configuration files from the application's configuration directory except for files with an absolute path provided by the arguments described earlier, for example `CONF_FILE`, `EXTRA_CONF_FILE`, `DTC_OVERLAY_FILE`, and `EXTRA_DTC_OVERLAY_FILE`.

The application configuration directory is defined by the APPLICATION_CONFIG_DIR variable.

APPLICATION_CONFIG_DIR will be set by one of the sources below with the highest priority listed first.

1. If APPLICATION_CONFIG_DIR is specified by the user with -DAPPLICATION_CONFIG_DIR=<path> or in a CMake file before find_package(Zephyr) then this folder is used a the application's configuration directory.
2. The application's source directory.

Kconfig Configuration

Application configuration options are usually set in prj.conf in the application directory. For example, C++ support could be enabled with this assignment:

```
CONFIG_CPP=y
```

Looking at existing samples is a good way to get started.

See *Setting Kconfig configuration values* for detailed documentation on setting Kconfig configuration values. The *The Initial Configuration* section on the same page explains how the initial configuration is derived. See kconfig-search for a complete list of configuration options. See *Hardening Tool* for security information related with Kconfig options.

The other pages in the *Kconfig section of the manual* are also worth going through, especially if you planning to add new configuration options.

Experimental features Zephyr is a project under constant development and thus there are features that are still in early stages of their development cycle. Such features will be marked [EXPERIMENTAL] in their Kconfig title.

The CONFIG_WARN_EXPERIMENTAL setting can be used to enable warnings at CMake configure time if any experimental feature is enabled.

```
CONFIG_WARN_EXPERIMENTAL=y
```

For example, if option CONFIG_FOO is experimental, then enabling it and CONFIG_WARN_EXPERIMENTAL will print the following warning at CMake configure time when you build an application:

```
warning: Experimental symbol FOO is enabled.
```

Devicetree Overlays

See *Set devicetree overlays*.

File Suffixes

Zephyr applications might want to have a single code base with multiple configurations for different build/product variants which would necessitate different Kconfig options and devicetree configuration. In order to better configure this, Zephyr provides a FILE_SUFFIX option when configuring applications that can be automatically appended to filenames. This is applied to Kconfig fragments and board overlays but with a fallback so that if such files do not exist, the files without these suffixes will be used instead.

Given the following example project layout:

```
<app>
├── CMakeLists.txt
├── prj.conf
├── prj_mouse.conf
├── boards
│   ├── native_sim.overlay
│   └── qemu_cortex_m3_mouse.overlay
└── src
    └── main.c
```

- If this is built normally without `FILE_SUFFIX` being defined for `native_sim` then `prj.conf` and `boards/native_sim.overlay` will be used.
- If this is build normally without `FILE_SUFFIX` being defined for `qemu_cortex_m3` then `prj.conf` will be used, no application devicetree overlay will be used.
- If this is built with `FILE_SUFFIX` set to `mouse` for `native_sim` then `prj_mouse.conf` and `boards/native_sim.overlay` will be used (there is no `native_sim_mouse.overlay` file so it falls back to `native_sim.overlay`).
- If this is build with `FILE_SUFFIX` set to `mouse` for `qemu_cortex_m3` then `prj_mouse.conf` will be used and `boards/qemu_cortex_m3_mouse.overlay` will be used.

Note

When `CONF_FILE` is set in the form of `prj_X.conf` then the `X` will be used as the build type. If this is combined with `FILE_SUFFIX` then the file suffix option will take priority over the build type.

2.4.8 Application-Specific Code

Application-specific source code files are normally added to the application's `src` directory. If the application adds a large number of files the developer can group them into sub-directories under `src`, to whatever depth is needed.

Application-specific source code should not use symbol name prefixes that have been reserved by the kernel for its own use. For more information, see Naming Conventions.

Third-party Library Code

It is possible to build library code outside the application's `src` directory but it is important that both application and library code targets the same Application Binary Interface (ABI). On most architectures there are compiler flags that control the ABI targeted, making it important that both libraries and applications have certain compiler flags in common. It may also be useful for glue code to have access to Zephyr kernel header files.

To make it easier to integrate third-party components, the Zephyr build system has defined CMake functions that give application build scripts access to the zephyr compiler options. The functions are documented and defined in cmake/modules/extensions.cmake and follow the naming convention `zephyr_get_<type>_<format>`.

The following variables will often need to be exported to the third-party build system.

- `CMAKE_C_COMPILER`, `CMAKE_AR`.
- `ARCH` and `BOARD`, together with several variables that identify the Zephyr kernel version.

samples/application_development/external_lib is a sample project that demonstrates some of these features.

2.4.9 Building an Application

The Zephyr build system compiles and links all components of an application into a single application image that can be run on simulated hardware or real hardware.

Like any other CMake-based system, the build process takes place *in two stages*. First, build files (also known as a buildsystem) are generated using the `cmake` command-line tool while specifying a generator. This generator determines the native build tool the buildsystem will use in the second stage. The second stage runs the native build tool to actually build the source files and generate an image. To learn more about these concepts refer to the CMake introduction in the official CMake documentation.

Although the default build tool in Zephyr is *west*, Zephyr's meta-tool, which invokes `cmake` and the underlying build tool (`ninja` or `make`) behind the scenes, you can also choose to invoke `cmake` directly if you prefer. On Linux and macOS you can choose between the `make` and `ninja` generators (i.e. build tools), whereas on Windows you need to use `ninja`, since `make` is not supported on this platform. For simplicity we will use `ninja` throughout this guide, and if you choose to use `west build` to build your application know that it will default to `ninja` under the hood.

As an example, let's build the Hello World sample for the `reel_board`:

Using west:

```
west build -b reel_board samples/hello_world
```

Using CMake and ninja:

```
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=reel_board samples/hello_world

# Now run the build tool on the generated build system:
ninja -Cbuild
```

On Linux and macOS, you can also build with `make` instead of `ninja`:

Using west:

- to use `make` just once, add `-- -G"Unix Makefiles"` to the west build command line; see the *west build* documentation for an example.
- to use `make` by default from now on, run `west config build.generator "Unix Makefiles"`.

Using CMake directly:

```
# Use cmake to configure a Make-based buildsystem:
cmake -Bbuild -DBOARD=reel_board samples/hello_world

# Now run the build tool on the generated build system:
make -Cbuild
```

Basics

1. Navigate to the application directory <app>.
2. Enter the following commands to build the application's `zephyr.elf` image for the board specified in the command-line parameters:

   Using west:

   ```
   west build -b <board>
   ```

   Using CMake and ninja:

```
mkdir build && cd build

# Use cmake to configure a Ninja-based buildsystem:
cmake -GNinja -DBOARD=<board> ..

# Now run the build tool on the generated build system:
ninja
```

If desired, you can build the application using the configuration settings specified in an alternate `.conf` file using the `CONF_FILE` parameter. These settings will override the settings in the application's `.config` file or its default `.conf` file. For example:

Using west:

```
west build -b <board> -- -DCONF_FILE=prj.alternate.conf
```

Using CMake and ninja:

```
mkdir build && cd build
cmake -GNinja -DBOARD=<board> -DCONF_FILE=prj.alternate.conf ..
ninja
```

As described in the previous section, you can instead choose to permanently set the board and configuration settings by either exporting **`BOARD`** and **`CONF_FILE`** environment variables or by setting their values in your `CMakeLists.txt` using `set()` statements. Additionally, `west` allows you to *set a default board*.

Build Directory Contents

When using the Ninja generator a build directory looks like this:

```
<app>/build
├── build.ninja
├── CMakeCache.txt
├── CMakeFiles
├── cmake_install.cmake
├── rules.ninja
└── zephyr
```

The most notable files in the build directory are:

- `build.ninja`, which can be invoked to build the application.
- A `zephyr` directory, which is the working directory of the generated build system, and where most generated files are created and stored.

After running `ninja`, the following build output files will be written to the `zephyr` sub-directory of the build directory. (This is **not the Zephyr base directory**, which contains the Zephyr source code etc. and is described above.)

- `.config`, which contains the configuration settings used to build the application.

> **Note**
>
> The previous version of `.config` is saved to `.config.old` whenever the configuration is updated. This is for convenience, as comparing the old and new versions can be handy.

- Various object files (`.o` files and `.a` files) containing compiled kernel and application code.
- `zephyr.elf`, which contains the final combined application and kernel binary. Other binary output formats, such as `.hex` and `.bin`, are also supported.

Rebuilding an Application

Application development is usually fastest when changes are continually tested. Frequently rebuilding your application makes debugging less painful as the application becomes more complex. It's usually a good idea to rebuild and test after any major changes to the application's source files, CMakeLists.txt files, or configuration settings.

Important

The Zephyr build system rebuilds only the parts of the application image potentially affected by the changes. Consequently, rebuilding an application is often significantly faster than building it the first time.

Sometimes the build system doesn't rebuild the application correctly because it fails to recompile one or more necessary files. You can force the build system to rebuild the entire application from scratch with the following procedure:

1. Open a terminal console on your host computer, and navigate to the build directory `<app>/build`.
2. Enter one of the following commands, depending on whether you want to use `west` or `cmake` directly to delete the application's generated files, except for the `.config` file that contains the application's current configuration information.

   ```
   west build -t clean
   ```

   or

   ```
   ninja clean
   ```

   Alternatively, enter one of the following commands to delete *all* generated files, including the `.config` files that contain the application's current configuration information for those board types.

   ```
   west build -t pristine
   ```

   or

   ```
   ninja pristine
   ```

   If you use west, you can take advantage of its capability to automatically *make the build folder pristine* whenever it is required.
3. Rebuild the application normally following the steps specified in *Building an Application* above.

Building for a board revision

The Zephyr build system has support for specifying multiple hardware revisions of a single board with small variations. Using revisions allows the board support files to make minor adjustments to a board configuration without duplicating all the files described in *Create your board directory* for each revision.

To build for a particular revision, use `<board>@<revision>` instead of plain `<board>`. For example:

Using west:

```
west build -b <board>@<revision>
```

Using CMake and ninja:

```
mkdir build && cd build
cmake -GNinja -DBOARD=<board>@<revision> ..
ninja
```

Check your board's documentation for details on whether it has multiple revisions, and what revisions are supported.

When targeting a board revision, the active revision will be printed at CMake configure time, like this:

```
-- Board: plank, Revision: 1.5.0
```

2.4.10 Run an Application

An application image can be run on a real board or emulated hardware.

Running on a Board

Most boards supported by Zephyr let you flash a compiled binary using the `flash` target to copy the binary to the board and run it. Follow these instructions to flash and run an application on real hardware:

1. Build your application, as described in *Building an Application*.
2. Make sure your board is attached to your host computer. Usually, you'll do this via USB.
3. Run one of these console commands from the build directory, `<app>/build`, to flash the compiled Zephyr image and run it on your board:

   ```
   west flash
   ```

   or

   ```
   ninja flash
   ```

The Zephyr build system integrates with the board support files to use hardware-specific tools to flash the Zephyr binary to your hardware, then run it.

Each time you run the flash command, your application is rebuilt and flashed again.

In cases where board support is incomplete, flashing via the Zephyr build system may not be supported. If you receive an error message about flash support being unavailable, consult your board's documentation for additional information on how to flash your board.

> **Note**
>
> When developing on Linux, it's common to need to install board-specific udev rules to enable USB device access to your board as a non-root user. If flashing fails, consult your board's documentation to see if this is necessary.

Running in an Emulator

Zephyr has built-in emulator support for QEMU. It allows you to run and test an application virtually, before (or in lieu of) loading and running it on actual target hardware.

Check out *Beyond the Getting Started Guide* for additional steps needed on Windows.

Follow these instructions to run an application via QEMU:

1. Build your application for one of the QEMU boards, as described in *Building an Application*.

   For example, you could set `BOARD` to:

   - `qemu_x86` to emulate running on an x86-based board
   - `qemu_cortex_m3` to emulate running on an ARM Cortex M3-based board

2. Run one of these console commands from the build directory, `<app>/build`, to run the Zephyr binary in QEMU:

   ```
   west build -t run
   ```

   or

   ```
   ninja run
   ```

3. Press `Ctrl A, X` to stop the application from running in QEMU.

   The application stops running and the terminal console prompt redisplays.

Each time you execute the run command, your application is rebuilt and run again.

Note

If the (Linux only) *Zephyr SDK* is installed, the `run` target will use the SDK's QEMU binary by default. To use another version of QEMU, *set the environment variable* `QEMU_BIN_PATH` to the path of the QEMU binary you want to use instead.

Note

You can choose a specific emulator by appending `_<emulator>` to your target name, for example `west build -t run_qemu` or `ninja run_qemu` for QEMU.

2.4.11 Custom Board, Devicetree and SOC Definitions

In cases where the board or platform you are developing for is not yet supported by Zephyr, you can add board, Devicetree and SOC definitions to your application without having to add them to the Zephyr tree.

The structure needed to support out-of-tree board and SOC development is similar to how boards and SOCs are maintained in the Zephyr tree. By using this structure, it will be much easier to upstream your platform related work into the Zephyr tree after your initial development is done.

Add the custom board to your application or a dedicated repository using the following structure:

```
boards/
soc/
CMakeLists.txt
prj.conf
README.rst
src/
```

where the `boards` directory hosts the board you are building for:

```
.
├── boards
│   └── vendor
│       └── my_custom_board
```

(continues on next page)

(continued from previous page)

```
|           ├── doc
|           |   └── img
|           └── support
└── src
```

and the soc directory hosts any SOC code. You can also have boards that are supported by a SOC that is available in the Zephyr tree.

Boards

Use the vendor name as the folder name (which must match the vendor prefix in dts/bindings/vendor-prefixes.txt if submitting upstream to Zephyr, or be `others` if it is not a vendor board) under `boards` for `my_custom_board`.

Documentation (under `doc/`) and support files (under `support/`) are optional, but will be needed when submitting to Zephyr.

The contents of `my_custom_board` should follow the same guidelines for any Zephyr board, and provide the following files:

```
my_custom_board_defconfig
my_custom_board.dts
my_custom_board.yaml
board.cmake
board.h
CMakeLists.txt
doc/
Kconfig.my_custom_board
Kconfig.defconfig
support/
```

Once the board structure is in place, you can build your application targeting this board by specifying the location of your custom board information with the `-DBOARD_ROOT` parameter to the CMake build system:

Using west:

```
west build -b <board name> -- -DBOARD_ROOT=<path to boards>
```

Using CMake and ninja:

```
cmake -Bbuild -GNinja -DBOARD=<board name> -DBOARD_ROOT=<path to boards> .
ninja -Cbuild
```

This will use your custom board configuration and will generate the Zephyr binary into your application directory.

You can also define the `BOARD_ROOT` variable in the application `CMakeLists.txt` file. Make sure to do so **before** pulling in the Zephyr boilerplate with `find_package(Zephyr ...)`.

Note

When specifying `BOARD_ROOT` in a CMakeLists.txt, then an absolute path must be provided, for example `list(APPEND BOARD_ROOT ${CMAKE_CURRENT_SOURCE_DIR}/<extra-board-root>)`. When using `-DBOARD_ROOT=<board-root>` both absolute and relative paths can be used. Relative paths are treated relatively to the application directory.

SOC Definitions

Similar to board support, the structure is similar to how SOCs are maintained in the Zephyr tree, for example:

```
soc
└── st
    └── stm32
        ├── common
        └── stm32l0x
```

The file soc/Kconfig will create the top-level `SoC/CPU/Configuration Selection` menu in Kconfig.

Out of tree SoC definitions can be added to this menu using the `SOC_ROOT` CMake variable. This variable contains a semicolon-separated list of directories which contain SoC support files.

Following the structure above, the following files can be added to load more SoCs into the menu.

```
soc
└── st
    └── stm32
        └── stm32l0x
            ├── Kconfig
            ├── Kconfig.soc
            └── Kconfig.defconfig
```

The Kconfig files above may describe the SoC or load additional SoC Kconfig files.

An example of loading `stm31l0` specific Kconfig files in this structure:

```
soc
└── st
    └── stm32
        ├── Kconfig.soc
        └── stm32l0x
            └── Kconfig.soc
```

can be done with the following content in `st/stm32/Kconfig.soc`:

```
rsource "*/Kconfig.soc"
```

Once the SOC structure is in place, you can build your application targeting this platform by specifying the location of your custom platform information with the `-DSOC_ROOT` parameter to the CMake build system:

Using west:

```
west build -b <board name> -- -DSOC_ROOT=<path to soc> -DBOARD_ROOT=<path to boards>
```

Using CMake and ninja:

```
cmake -Bbuild -GNinja -DBOARD=<board name> -DSOC_ROOT=<path to soc> -DBOARD_ROOT=<path to␣
↪boards> .
ninja -Cbuild
```

This will use your custom platform configurations and will generate the Zephyr binary into your application directory.

See *Build settings* for information on setting SOC_ROOT in a module's `zephyr/module.yml` file.

Or you can define the `SOC_ROOT` variable in the application `CMakeLists.txt` file. Make sure to do so **before** pulling in the Zephyr boilerplate with `find_package(Zephyr ...)`.

Note

When specifying SOC_ROOT in a CMakeLists.txt, then an absolute path must be provided, for example `list(APPEND SOC_ROOT ${CMAKE_CURRENT_SOURCE_DIR}/<extra-soc-root>`. When using `-DSOC_ROOT=<soc-root>` both absolute and relative paths can be used. Relative paths are treated relatively to the application directory.

Devicetree Definitions

Devicetree directory trees are found in APPLICATION_SOURCE_DIR, BOARD_DIR, and ZEPHYR_BASE, but additional trees, or DTS_ROOTs, can be added by creating this directory tree:

```
include/
dts/common/
dts/arm/
dts/
dts/bindings/
```

Where 'arm' is changed to the appropriate architecture. Each directory is optional. The binding directory contains bindings and the other directories contain files that can be included from DT sources.

Once the directory structure is in place, you can use it by specifying its location through the DTS_ROOT CMake Cache variable:

Using west:

```
west build -b <board name> -- -DDTS_ROOT=<path to dts root>
```

Using CMake and ninja:

```
cmake -Bbuild -GNinja -DBOARD=<board name> -DDTS_ROOT=<path to dts root> .
ninja -Cbuild
```

You can also define the variable in the application `CMakeLists.txt` file. Make sure to do so **before** pulling in the Zephyr boilerplate with `find_package(Zephyr ...)`.

Note

When specifying DTS_ROOT in a CMakeLists.txt, then an absolute path must be provided, for example `list(APPEND DTS_ROOT ${CMAKE_CURRENT_SOURCE_DIR}/<extra-dts-root>`. When using `-DDTS_ROOT=<dts-root>` both absolute and relative paths can be used. Relative paths are treated relatively to the application directory.

Devicetree source are passed through the C preprocessor, so you can include files that can be located in a DTS_ROOT directory. By convention devicetree include files have a `.dtsi` extension.

You can also use the preprocessor to control the content of a devicetree file, by specifying directives through the DTS_EXTRA_CPPFLAGS CMake Cache variable:

Using west:

```
west build -b <board name> -- -DDTS_EXTRA_CPPFLAGS=-DTEST_ENABLE_FEATURE
```

Using CMake and ninja:

```
cmake -Bbuild -GNinja -DBOARD=<board name> -DDTS_EXTRA_CPPFLAGS=-DTEST_ENABLE_FEATURE .
ninja -Cbuild
```

2.5 Debugging

2.5.1 Application Debugging

This section is a quick hands-on reference to start debugging your application with QEMU. Most content in this section is already covered in QEMU and GNU_Debugger reference manuals.

In this quick reference, you'll find shortcuts, specific environmental variables, and parameters that can help you to quickly set up your debugging environment.

The simplest way to debug an application running in QEMU is using the GNU Debugger and setting a local GDB server in your development system through QEMU.

You will need an ELF (Executable and Linkable Format) binary image for debugging purposes. The build system generates the image in the build directory. By default, the kernel binary name is `zephyr.elf`. The name can be changed using `CONFIG_KERNEL_BIN_NAME`.

GDB server

We will use the standard 1234 TCP port to open a GDB (GNU Debugger) server instance. This port number can be changed for a port that best suits the development environment. There are multiple ways to do this. Each way starts a QEMU instance with the processor halted at startup and with a GDB server instance listening for a connection.

Running QEMU directly You can run QEMU to listen for a "gdb connection" before it starts executing any code to debug it.

```
qemu -s -S <image>
```

will setup Qemu to listen on port 1234 and wait for a GDB connection to it.

The options used above have the following meaning:

- `-S` Do not start CPU at startup; rather, you must type 'c' in the monitor.
- `-s` Shorthand for `-gdb tcp::1234`: open a GDB server on TCP port 1234.

Running QEMU via `ninja` Run the following inside the build directory of an application:

```
ninja debugserver
```

QEMU will write the console output to the path specified in **`${QEMU_PIPE}`** via CMake, typically `qemu-fifo` within the build directory. You may monitor this file during the run with **`tail -f qemu-fifo`**.

Running QEMU via `west` Run the following from your project root:

```
west build -t debugserver_qemu
```

QEMU will write the console output to the terminal from which you invoked **`west`**.

Configuring the `gdbserver` listening device The Kconfig option `CONFIG_QEMU_GDBSERVER_LISTEN_DEV` controls the listening device, which can be a TCP port number or a path to a character device. GDB releases 9.0 and newer also support Unix domain sockets.

If the option is unset, then the QEMU invocation will lack a `-s` or a `-gdb` parameter. You can then use the `QEMU_EXTRA_FLAGS` shell environment variable to pass in your own listen device configuration.

GDB client

Connect to the server by running **gdb** and giving these commands:

```
$ path/to/gdb path/to/zephyr.elf
(gdb) target remote localhost:1234
(gdb) dir ZEPHYR_BASE
```

> **Note**
>
> Substitute the correct *ZEPHYR_BASE* for your system.

You can use a local GDB configuration `.gdbinit` to initialize your GDB instance on every run. Your home directory is a typical location for `.gdbinit`, but you can configure GDB to load from other locations, including the directory from which you invoked **gdb**. This example file performs the same configuration as above:

```
target remote localhost:1234
dir ZEPHYR_BASE
```

Alternate interfaces GDB provides a curses-based interface that runs in the terminal. Pass the `--tui` option when invoking **gdb** or give the `tui enable` command within **gdb**.

> **Note**
>
> The GDB version on your development system might not support the `--tui` option. Please make sure you use the GDB binary from the SDK which corresponds to the toolchain that has been used to build the binary.

Finally, the command below connects to the GDB server using the DDD (Data Display Debugger), a graphical frontend for GDB. The following command loads the symbol table from the ELF binary file, in this instance, `zephyr.elf`.

```
ddd --gdb --debugger "gdb zephyr.elf"
```

Both commands execute **gdb**. The command name might change depending on the toolchain you are using and your cross-development tools.

ddd may not be installed in your development system by default. Follow your system instructions to install it. For example, use **sudo apt-get install ddd** on an Ubuntu system.

Debugging

As configured above, when you connect the GDB client, the application will be stopped at system startup. You may set breakpoints, step through code, etc. as when running the application directly within **gdb**.

> **Note**
>
> **gdb** will not print the system console output as the application runs, unlike when you run a native application in GDB directly. If you just **continue** after connecting the client, the application will run, but nothing will appear to happen. Check the console output as described above.

2.5.2 Debug with Eclipse

Overview

CMake supports generating a project description file that can be imported into the Eclipse Integrated Development Environment (IDE) and used for graphical debugging.

The GNU MCU Eclipse plug-ins provide a mechanism to debug ARM projects in Eclipse with pyOCD, Segger J-Link, and OpenOCD debugging tools.

The following tutorial demonstrates how to debug a Zephyr application in Eclipse with pyOCD in Windows. It assumes you have already installed the GCC ARM Embedded toolchain and pyOCD.

Set Up the Eclipse Development Environment

1. Download and install Eclipse IDE for C/C++ Developers.
2. In Eclipse, install the GNU MCU Eclipse plug-ins by opening the menu `Window->Eclipse Marketplace...`, searching for `GNU MCU Eclipse`, and clicking `Install` on the matching result.
3. Configure the path to the pyOCD GDB server by opening the menu `Window->Preferences`, navigating to `MCU`, and setting the `Global pyOCD Path`.

Generate and Import an Eclipse Project

1. Set up a GNU Arm Embedded toolchain as described in *GNU Arm Embedded*.
2. Navigate to a folder outside of the Zephyr tree to build your application.

```
# On Windows
cd %userprofile%
```

> **Note**
>
> If the build directory is a subdirectory of the source directory, as is usually done in Zephyr, CMake will warn:
>
> "The build directory is a subdirectory of the source directory.
>
> This is not supported well by Eclipse. It is strongly recommended to use a build directory which is a sibling of the source directory."

3. Configure your application with CMake and build it with ninja. Note the different CMake generator specified by the `-G"Eclipse CDT4 - Ninja"` argument. This will generate an Eclipse project description file, `.project`, in addition to the usual ninja build files.

   Using west:

```
west build -b frdm_k64f %ZEPHYR_BASE%\samples\synchronization -- -G"Eclipse CDT4 -␣
↪Ninja"
```

Using CMake and ninja:

```
cmake -Bbuild -GNinja -DBOARD=frdm_k64f -G"Eclipse CDT4 - Ninja" %ZEPHYR_BASE%\samples\
↪synchronization
ninja -Cbuild
```

4. In Eclipse, import your generated project by opening the menu `File->Import...` and selecting the option `Existing Projects into Workspace`. Browse to your application build directory in the choice, `Select root directory:`. Check the box for your project in the list of projects found and click the `Finish` button.

Create a Debugger Configuration

1. Open the menu `Run->Debug Configurations...`.
2. Select `GDB PyOCD Debugging`, click the `New` button, and configure the following options:
   - In the Main tab:
     - Project: `my_zephyr_app@build`
     - C/C++ Application: `zephyr/zephyr.elf`
   - In the Debugger tab:
     - pyOCD Setup
       - Executable path: `$pyocd_path\$pyocd_executable`
       - Uncheck “Allocate console for semihosting”
     - Board Setup
       - Bus speed: 8000000 Hz
       - Uncheck “Enable semihosting”
     - GDB Client Setup
       - Executable path example (use your `GNUARMEMB_TOOLCHAIN_PATH`): `C:\gcc-arm-none-eabi-6_2017-q2-update\bin\arm-none-eabi-gdb.exe`
   - In the SVD Path tab:
     - File path: `<workspace top>\modules\hal\nxp\mcux\devices\MK64F12\MK64F12.xml`

> **Note**
>
> This is optional. It provides the SoC’s memory-mapped register addresses and bitfields to the debugger.

3. Click the `Debug` button to start debugging.

RTOS Awareness

Support for Zephyr RTOS awareness is implemented in pyOCD v0.11.0 and later. It is compatible with GDB PyOCD Debugging in Eclipse, but you must enable CONFIG_DEBUG_THREAD_INFO=y in your application.

2.5.3 Debugging I2C communication

There is a possibility to log all or some of the I2C transactions done by the application. This feature is enabled by the Kconfig option `CONFIG_I2C_DUMP_MESSAGES`, but it uses the `LOG_DBG` function to print the contents so the `CONFIG_I2C_LOG_LEVEL_DBG` option must also be enabled.

The sample output of the dump looks like this:

```
D: I2C msg: io_i2c_ctrl7_port0, addr=50
D:    W      len=01: 00
D:    R Sr P len=08:
D: contents:
D: 43 42 41 00 00 00 00 00 |CBA.....
```

The first line indicates the I2C controller and the target address of the transaction. In above example, the I2C controller is named `io_i2c_ctrl7_port0` and the target device address is `0x50`

Note

the address, length and contents values are in hexadecimal, but lack the `0x` prefix

Next lines contain messages, both sent and received. The contents of write messages is always shown, while the content of read messages is controlled by a parameter to the function `i2c_dump_msgs_rw`. This function is available for use by user, but is also called internally by `i2c_transfer` API function with read content dump enabled. Before the length parameter, the header of the message is printed using abbreviations:

- W - write message
- R - read message
- Sr - restart bit
- P - stop bit

The above example shows one write message with byte `0x00` representing the address of register to read from the I2C target. After that the log shows the length of received message and following that, the bytes read from the target `43 42 41 00 00 00 00 00`. The content dump consist of both the hex and ASCII representation.

Filtering the I2C communication dump

By default, all I2C communication is logged between all I2C controllers and I2C targets. It may litter the log with unrelated devices and make it difficult to effectively debug the communication with a device of interest.

Enable the Kconfig option `CONFIG_I2C_DUMP_MESSAGES_ALLOWLIST` to create an allowlist of I2C targets to log. The allowlist of devices is configured using the devicetree, for example:

```
/ {
    i2c {
        display0: some-display@a {
            ...
        };
        sensor3: some-sensor@b {
            ...
        };
    };

    i2c-dump-allowlist {
```

(continues on next page)

(continued from previous page)

```
        compatible = "zephyr,i2c-dump-allowlist";
        devices = < &display0 >, < &sensor3 >;
    };
};
```

The filters nodes are identified by the compatible string with `zephyr,i2c-dump-allowlist` value. The devices are selected using the `devices` property with phandles to the devices on the I2C bus.

In the above example, the communication with device `display0` and `sensor3` will be displayed in the log.

2.6 API Status and Guidelines

2.6.1 API Overview

The table lists Zephyr's APIs and information about them, including their current *stability level*. More details about API changes between major releases are available in the zephyr_release_notes.

The version column uses semantic version, and has the following expectations:

- Major version zero (0.y.z) is for initial development. Anything MAY change at any time. The public API SHOULD NOT be considered stable.
  - If minor version is up to one (0.1.z), API is considered *experimental*.
  - If minor version is larger than one (0.y.z | y > 1), API is considered *unstable*.
- Version 1.0.0 defines the public API. The way in which the version number is incremented after this release is dependent on this public API and how it changes.
  - APIs with major versions equal or larger than one (x.y.z | x >= 1) are considered *stable*.
  - All existing stable APIs in Zephyr will be start with version 1.0.0.
- Patch version Z (x.y.Z | x > 0) MUST be incremented if only backwards compatible bug fixes are introduced. A bug fix is defined as an internal change that fixes incorrect behavior.
- Minor version Y (x.Y.z | x > 0) MUST be incremented if new, backwards compatible functionality is introduced to the public API. It MUST be incremented if any public API functionality is marked as deprecated. It MAY be incremented if substantial new functionality or improvements are introduced within the private code. It MAY include patch level changes. Patch version MUST be reset to 0 when minor version is incremented.
- Major version X (x.Y.z | x > 0) MUST be incremented if a compatibility breaking change was made to the API.

Note

Version for existing APIs are initially set based on the current state of the APIs:

- 0.1.0 denotes an *experimental* API
- 0.8.0 denote an *unstable* API,
- and finally 1.0.0 indicates a *stable* APIs.

Changes to APIs in the future will require adapting the version following the guidelines above.

API	Version	Available in Zephyr Since
Audio		
Audio Codec Interface	0.1.0	v1.13.0
Digital Microphone Interface	0.1.0	v1.13.0
Connectivity		
Bluetooth APIs		
AUDIO		
Attribute Protocol (ATT)		
Audio Input Control Service (AICS)	0.8.0	v2.6.0
Basic Audio Profile (BAP) LC3 Presets	0.8.0	v3.0.0
Battery Service (BAS)		
BlueNRG HCI driver extended API		
Bluetooth Audio		
Codec capability parsing APIs		
Codec config parsing APIs		
Bluetooth Basic Audio Profile	0.8.0	v3.0.0
BAP Broadcast APIs		
BAP Broadcast Sink APIs		
BAP Broadcast Source APIs		
BAP Broadcast Sink APIs		
BAP Broadcast Source APIs		
BAP Unicast Client APIs		
BAP Unicast Server APIs		
Bluetooth Controller		
Bluetooth Gaming Audio Profile	0.8.0	v3.5.0
Bluetooth HCI APIs	0.2.0	v3.7.0
Bluetooth Mesh		
Access layer		
Bluetooth Mesh BLOB Transfer Client model API		
Bluetooth Mesh BLOB Transfer Server model API		
Bluetooth Mesh BLOB flash stream		
Bluetooth Mesh BLOB model API		
Bluetooth Mesh Device Firmware Update		
Bluetooth Mesh Device Firmware Update (DFU) metadata		
Firmware Update Server model		
Firmware Uppdate Client model		
Bluetooth Mesh On-Demand Private GATT Proxy Client		
Bluetooth Mesh On-Demand Private GATT Proxy Server		
Bluetooth Mesh Private Beacon Client		
Bluetooth Mesh Private Beacon Server		
Bluetooth Mesh SAR Configuration Client Model		
Bluetooth Mesh SAR Configuration Server Model		
Bluetooth Mesh Solicitation PDU RPL Client		
Bluetooth Mesh Solicitation PDU RPL Server		
Configuration Client Model		
Configuration Server Model		
Firmware Distribution models		
Firmware Distribution Server model		
Health Client Model		
Health Server Model		
Health faults		
Heartbeat		
Large Composition Data Client model		
Large Composition Data Server model		
Message		
Opcodes Aggregator Client model		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Opcodes Aggregator Server model		
Provisioning		
Proxy		
Remote Provisioning Client model		
Remote Provisioning models		
Remote provisioning server		
Runtime Configuration		
Application Configuration		
Subnet Configuration		
SAR Configuration common header		
Statistic		
Bluetooth testing callbacks		
Byteorder		
Common Audio Profile (CAP)	0.8.0	v3.2.0
Connection management		
Coordinated Set Identification Profile (CSIP)	0.8.0	v3.0.0
Cryptography		
Data buffers		
Device Address		
Encrypted Advertising Data (EAD)		
Gaming Audio Profile (GMAP) LC3 Presets	0.8.0	v3.5.0
Generic Access Profile (GAP)	1.0.0	v1.0.0
Defines and Assigned Numbers		
Generic Attribute Profile (GATT)		
GATT Client APIs		
GATT Server APIs		
HCI RAW channel		
HCI drivers		
Hands Free Profile (HFP)		
Hands Free Profile - Audio Gateway (HFP-AG)		
Hearing Access Service (HAS)	0.8.0	v3.1.0
Heart Rate Service (HRS)		
Immediate Alert Service (IAS)		
Isochronous channels (ISO)	0.8.0	v2.3.0
L2CAP		
Media Control Client (MCC)	0.8.0	v3.0.0
Media Control Service (MCS)	0.8.0	v3.0.0
Media Proxy	0.8.0	v3.0.0
Microphone Control Profile (MICP)	0.8.0	v2.7.0
Object Transfer Service (OTS)		
Public Broadcast Profile (PBP)	0.8.0	v3.5.0
RFCOMM		
Service Discovery Protocol (SDP)		
Telephone Bearer Service (TBS)	0.8.0	v3.0.0
UUIDs		
Volume Control Profile (VCP)	0.8.0	v2.7.0
Volume Offset Control Service (VOCS)	0.8.0	v2.6.0
CAN ISO-TP Protocol		
IEEE 802.15.4 and Thread APIs	0.8.0	v1.0.0
IEEE 802.15.4 Drivers	0.8.0	v1.0.0
IEEE 802.15.4 L2	0.8.0	v1.0.0
IEEE 802.15.4 Net Management	0.8.0	v1.0.0
OpenThread L2 abstraction layer		
LoRaWAN APIs	0.1.0	v2.5.0
Modem APIs	0.1.0	v3.5.0

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Modem CMUX		
Modem PPP		
Modem Pipe		
Modem Ubx		
Modem pipelink		
Networking	1.0.0	v1.0.0
Application network context		
BSD Sockets compatible API		
Socket options for TLS		
BSD socket service API		
COAP Library	0.8.0	v1.10.0
CoAP Manager Events		
CoAP client API		
CoAP service API		
Connection Manager API		
Connection Manager Connectivity API	0.1.0	v3.4.0
Connection Manager Connectivity Bulk API		
Connection Manager Connectivity Implementation API		
DHCPv4		
DHCPv4 server		
DHCPv6		
DNS Resolve Library		
DNS Service Discovery		
Distributed Switch Architecture definitions and helpers		
Dummy L2/driver Support Functions		
Ethernet Bridging API		
Ethernet Library		
Ethernet PHY Interface		
Ethernet Support Functions		
Ethernet MII Support Functions		
IEEE 802.3 management interface		
HTTP HPACK		
HTTP client API	0.2.0	v2.1.0
HTTP request methods		
HTTP response status codes		
HTTP server API		
HTTP service API		
IGMP API		
IPv4/IPv6 primitives and helpers		
Link Layer Discovery Protocol definitions and helpers		
LwM2M high-level API	0.8.0	v1.9.0
LwM2M path helper macros		
MQTT Client library	0.8.0	v1.14.0
MQTT-SN Client library		
Network Buffer Library		
Network Configuration Library		
Network Core Library		
Network Core Library		
Network Core Library		
Network Hostname Library		
Network Interface abstraction layer		
Network L2 Abstraction Layer		
Network Link Address Library		
Network Management		
Network Offloading Interface		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Network Packet Filter API		
Basic Filter Conditions		
Ethernet Filter Conditions		
Network Packet Library		
Network Statistics Library		
Network long timeout primitives and helpers		
Network packet capture		
Network time representation.		
Offloaded Net Devices		
PPP L2/driver Support Functions		
PTP support		
PTP time		
Promiscuous mode		
SNTP		
Send and receive IPv4 or IPv6 ICMP Echo Request messages.		
TFTP Client library		
TLS credentials management		
Trickle Algorithm Library		
Virtual Interface Library		
Virtual LAN definitions and helpers		
Virtual Network Interface Support Functions		
Websocket API		
Wi-Fi Management		
Wi-Fi Network Manager API		
Zperf API		
gPTP support		
USB		
USB Audio Class 2 device API		
USB BOS support		
USB HID class API		
HID class USB specific definitions		
USB HID common definitions		
Mouse and keyboard report descriptors		
USB HID Item helpers		
USB Host Core API		
USB Mass Storage Class device API		
USB device core API		
USB device core API		
USBD HID device API		
DSP Interface	0.1.0	v3.3.0
Basic Math Functions		
Vector Absolute Value		
Vector Addition		
Vector Clipping		
Vector Dot Product		
Vector Multiplication		
Vector Negate		
Vector Offset		
Vector Scale		
Vector Shift		
Vector Subtraction		
Vector bitwise AND		
Vector bitwise NOT		
Vector bitwise OR		
Vector bitwise XOR		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Helper macros for printing Q values.		
Device Driver APIs		
1-Wire Interface	0.1.0	v3.2.0
1-Wire Sensor API		
1-Wire data link layer		
1-Wire network layer		
ADC driver APIs	1.0.0	v1.0.0
Emulated ADC		
Analog axis API		
BBRAM Interface		
BBRAM emulator backend API		
BC1.2 backed emulator APIs		
BC1.2 driver APIs		
CAN Interface	1.1.0	v1.12.0
CAN Transceiver	0.1.0	v3.1.0
Cache Controller Interface		
Cellular Interface		
Charger Interface		
Clock Control Interface	1.0.0	v1.0.0
LiteX Clock Control driver interface		
Coredump pseudo-device driver APIs		
Counter Interface	0.8.0	v1.14.0
DAC driver APIs	0.8.0	v2.3.0
DAI Interface	0.1.0	v3.1.0
DMA Interface	1.0.0	v1.5.0
Disk Driver Interface	1.0.0	v1.6.0
Display Interface	0.8.0	v1.14.0
LCD Interface		
EC Host Command Interface	0.1.0	v2.4.0
EDAC API	0.8.0	v2.5.0
EEPROM Interface	1.0.0	v2.1.0
ESPI Driver APIs		
Entropy Interface	1.0.0	v1.10.0
External Cache Controller Interface		
FLASH Interface	1.0.0	v1.2.0
FLASH internal Interface		
Fuel Gauge Interface	0.1.0	v3.3.0
Fuel gauge backend emulator APIs		
GNSS Interface	0.1.0	v3.6.0
GPIO Driver APIs	1.0.0	v1.0.0
Emulated GPIO		
MAX32-specific GPIO Flags		
nPM1300-specific GPIO Flags		
nPM6001-specific GPIO Flags		
nRF-specific GPIO Flags		
HW spinlock Interface		
Hardware Info Interface	1.0.0	v1.14.0
I2C EEPROM Target Driver API	1.0.0	v1.13.0
I2C Interface	1.0.0	v1.0.0
I2S Interface	1.0.0	v1.9.0
I3C Interface	0.1.0	v3.2.0
I3C Address-related Helper Code		
I3C Common Command Codes		
I3C Devicetree related bits		
I3C In-Band Interrupts		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
I3C Target Device API		
I3C Transfer API		
IPM Interface	1.0.0	v1.0.0
Input Interface	0.1.0	v3.4.0
Input Event Definitions		
Inter-VM Shared Memory (ivshmem) reference API		
Keyboard Matrix API		
Keyboard Scan Driver APIs	1.0.0	v2.1.0
LED Interface	1.0.0	v1.12.0
LED Strip Interface		
LoRa APIs	0.1.0	v2.2.0
MBOX Interface	0.1.0	v1.0.0
MDIO Interface		
MIPI Display interface		
MIPI-DBI driver APIs	0.1.0	v3.6.0
MIPI-DSI driver APIs	0.1.0	v3.1.0
MODBUS		
MSPI Devicetree related macros		
MSPI Driver APIs		
MSPI Configure API		
MSPI Transfer API		
MSPI callback API		
Miscellaneous Drivers APIs		
Devmux Driver APIs		
FT8xx driver APIs		
FT8xx co-processor		
FT8xx common functions		
FT8xx display list		
FT8xx memory map		
FT8xx reference API		
Multi Function Device Drivers APIs		
MFD AD559X interface		
MFD AXP192 interface		
MFD BD8LB600FS interface		
MFD NPM1300 Interface		
PCI Express Controller Interface		
PCIe Host Interface		
PCIe Capabilities		
PCIe Host MSI Interface		
PCIe Host PTM Interface		
PCIe Virtual Channel Host Interface		
PECI Interface	1.0.0	v2.1.0
PS/2 Driver APIs		
PWM Interface	1.0.0	v1.0.0
Pin Controller Interface	0.1.0	v3.0.0
Dynamic Pin Control		
RTC DS3231 Interface		
RTC Interface	0.1.0	v3.4.0
Regulator Interface	0.1.0	v2.4.0
ADP5360 Devicetree helpers.		
AXP192 Devicetree helpers.		
Devicetree helpers		
MAX20335 Devicetree helpers.		
NPM1100 Devicetree helpers.		
NPM1300 Devicetree helpers.		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
NPM6001 Devicetree helpers.		
Regulator Parent Interface		
PCA9420 Utilities.		
Reset Controller Interface	0.2.0	v3.1.0
Retained memory driver interface	0.8.0	v3.4.0
SDHC interface	0.1.0	v3.1.0
SMBus Interface	0.1.0	v3.4.0
SPI Interface	1.0.0	v1.0.0
SYSCON Interface		
Sensor Interface	1.0.0	v1.2.0
Invensense (TDK) ICM42688 DT Options		
Accelerometer data rate options		
Accelerometer power modes		
Accelerometer scale options		
Gyroscope data rate options		
Gyroscope power modes		
Gyroscope scale options		
Sensor emulator backend API		
TEE Interface		
Text Display Interface	0.1.0	v3.4.0
Time-aware GPIO Interface	0.1.0	v3.5.0
UART Interface	1.0.0	v1.0.0
Async UART API	0.8.0	v1.14.0
Interrupt-driven UART API		
Polling UART API		
USB Power Delivery		
USB Type-C		
USB Type-C Port Controller API	0.1.0	v3.1.0
USB device controller driver API		
USB host controller driver API		
USB-C VBUS API	0.1.0	v3.3.0
Video Controls		
Video Interface	1.0.0	v2.1.0
Video pixel formats		
Watchdog Interface	1.0.0	v1.0.0
Device Model	1.1.0	v1.0.0
Device memory-mapped IO management		
Named MMIO region macros		
Single MMIO region macros		
Top-level MMIO region macros		
Devicetree	1.1.0	v2.2.0
“For-each” macros		
Bus helpers		
Chosen nodes		
Dependency tracking		
Devicetree CAN API		
Devicetree Clocks API		
Devicetree DMA API		
Devicetree Fixed Partition API		
Devicetree GPIO API		
Devicetree IO Channels API		
Devicetree Interrupt Controller API		
Devicetree MBOX API		
Devicetree PWMs API		
Devicetree Reset Controller API		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Devicetree SPI API		
Existence checks		
Instance-based devicetree APIs		
Node identifiers and helpers		
Pin control		
Property accessors		
Vendor and model name helpers		
interrupts property		
ranges property		
reg property		
Error numbers		
Internal and System API		
Architecture Interface		
Architecture thread APIs		
Architecture timing APIs		
Architecture-specific IRQ APIs		
Architecture-specific SMP APIs		
Architecture-specific Thread Local Storage APIs		
Architecture-specific core dump APIs		
Architecture-specific gdbstub APIs		
Architecture-specific memory-mapping APIs		
Architecture-specific power management APIs		
Architecture-specific userspace APIs		
Miscellaneous architecture APIs		
Kernel Memory Management Internal APIs		
User Mode Internal APIs		
User mode and Syscall APIs		
Kernel APIs	1.0.0	v1.0.0
Async polling APIs		
Asynchronous Notification APIs		
Atomic Services APIs		
Barrier Services APIs	0.1.0	v3.4.0
CPU Idling APIs		
Condition Variables APIs		
Event APIs		
FIFO APIs		
FUTEX APIs		
Fatal error APIs		
Fatal error base types		
Floating Point APIs		
Heap APIs		
Interrupt Service Routine APIs		
Kernel Memory Management		
Demand Paging		
Backing Store APIs		
Demand Paging APIs		
Eviction Algorithm APIs		
LIFO APIs		
Mailbox APIs		
Memory Slab APIs		
Memory domain APIs		
Application memory domain APIs		
Message Queue APIs		
Mutex APIs		
Object Core APIs		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Object Core Statistics APIs		
On-Off Service APIs		
Pipe APIs		
Queue APIs		
Semaphore APIs		
Spinlock APIs		
Stack APIs		
System Clock APIs		
Thread APIs		
Thread Stack APIs		
Timer APIs		
User Mode APIs		
User mode mutex APIs		
User mode semaphore APIs		
Version APIs		
Work Queue APIs		
Memory Management APIs		
Memory heaps based on memory attributes		
Memory-Attr Interface		
Operating System Services		
Bindesc Define		
Cache Interface		
Checksum		
CRC		
Console API		
Coredump APIs		
Crypto	1.0.0	v1.7.0
Cipher		
Hash		
Random Function APIs	1.0.0	v1.0.0
File System APIs	1.0.0	v1.5.0
File System Storage		
Flash Circular Buffer (FCB)	1.0.0	v1.11.0
Flash Circular Buffer Data Structures		
fcb API		
fcb non-API prototypes		
Non-volatile Storage (NVS)	1.0.0	v1.12.0
Non-volatile Storage APIs		
Non-volatile Storage Data Structures		
Settings	1.0.0	v1.12.0
Settings backend interface		
Settings name processing		
Settings subsystem runtime		
Flash image API		
Heap Management		
Heap Listener APIs		
Low Level Heap Allocator		
Multi-Heap Wrapper		
Shared multi-heap interface		
IPC		
IPC service APIs		
IPC service RPMsg API		
IPC service backend		
IPC service static VRINGs API		
Icmsg IPC library API		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Icmsg multi-endpoint IPC library API		
Packed Buffer API		
RPMsg service APIs		
Iterable Sections APIs		
Linkable loadable extensions	0.1.0	v3.5.0
ELF constants and data types		
ELF loader context		
Exported symbol definitions		
Logging	1.0.0	v1.13.0
Logger system		v1.13.0
Log link API		
Log message API		
Log output API		
Log output formatting flags.		
Logger backend interface		
Logger multidomain backend helpers		
Logger backend standard interface		
Logger control API		v1.13.0
Logging API		
MCUmgr		
MCUmgr callback API		
MCUmgr fs_mgmt callback API		
MCUmgr img_mgmt callback API		
MCUmgr os_mgmt callback API		
MCUmgr settings_mgmt callback API		
MCUmgr handler API		
MCUmgr img_mgmt API		
MCUmgr img_mgmt_client API		
MCUmgr mgmt API	1.0.0	v1.11.0
MCUmgr os_mgmt_client API		
MCUmgr transport SMP API		
SMP client API		
Memory Management		
Memory Banks Driver APIs		
Memory Blocks APIs		
Memory Management Driver APIs		
Power Management (PM)		v1.2.0
CPU Power Management		
Device		
Device Runtime		
States		
System		v1.2.0
Hooks		
Policy		
RTIO	0.1.0	v3.2.0
RTIO CQE Flags		
RTIO Priorities		
RTIO SQE Flags		
Retention API	0.1.0	v3.4.0
Boot mode interface		
Bootloader info interface	0.1.0	v3.5.0
STP Decoder API		
Semihosting APIs		
Shell API	1.0.0	v1.14.0
State Machine Framework API	0.1.0	

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Storage APIs		
Disk Access Interface		
Stream to flash interface	0.1.0	v2.3.0
flash area Interface	1.0.0	v1.11.0
Symbol Table API		
System Initialization		
System power off		
Task Watchdog APIs	0.8.0	v2.5.0
Thread analyzer		
Timing Measurement APIs		
Arch specific Timing Measurement APIs		
Board specific Timing Measurement APIs		
SoC specific Timing Measurement APIs		
Tracing		
Object tracking		
Tracing APIs		
Conditional Variable Tracing APIs		
Event Tracing APIs		
FIFO Tracing APIs		
Heap Tracing APIs		
LIFO Tracing APIs		
Mailbox Tracing APIs		
Memory Slab Tracing APIs		
Message Queue Tracing APIs		
Mutex Tracing APIs		
Network Socket Tracing APIs		
PM Device Runtime Tracing APIs		
Pipe Tracing APIs		
Poll Tracing APIs		
Queue Tracing APIs		
Semaphore Tracing APIs		
Stack Tracing APIs		
Syscall Tracing APIs		
System PM Tracing APIs		
Thread Tracing APIs		
Timer Tracing APIs		
Work Delayable Tracing APIs		
Work Poll Tracing APIs		
Work Queue Tracing APIs		
Work Tracing APIs		
Tracing format APIs		
Tracing utility macros		
Zbus APIs		
S2RAM APIs		
Sensing		
Data Types		
Sensing Sensor API		
Sensor Callbacks		
Sensing Subsystem API		
Sensor Types		
Testing		
Emulator interface		
I2C Emulation Interface		
MSPI Emulation Interface		
SPI Emulation Interface		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
eSPI Emulation Interface		
FFF extensions		
Zephyr Testing Framework (ZTest)		
Ztest assertion macros		
Ztest assumption macros		
Ztest expectation macros		
Ztest mocking support		
Ztest testing macros		
Ztest ztress macros		
Third-party		
BBC micro:bit display APIs		
Grove display APIs		
MCUboot image control API		
UpdateHub Firmware Over-the-Air		
hawkBit Firmware Over-the-Air		
USB Device Controller API		
USB Device Core API		
USB-C Device API	0.1.0	v3.3.0
Sink_callbacks		
Source_callbacks		
Utilities		
Base64		
Data Structure APIs		
Balanced Red/Black Tree		
Bit array		
Doubly-linked list		
Flagged Single-linked list		
Hashmap		
Hash Functions		
Hashmap Implementations		
MPSC (Multi producer, single consumer) packet buffer API		
MPSC (Multi producer, single consumer) packet header		
MPSC packet buffer flags		
MPSC Lockfree Queue API		
Ring Buffer APIs		
SPSC (Single producer, single consumer) packet buffer API		
SPSC packet buffer flags		
SPSC API		
Single-linked list		
Formatted Output APIs		
Package convert flags		
Package flags		
cbvprintf processing flags.		
JSON		
JSON Web Token (JWT)		
Linear Range		
Math extras		
Monochrome Character Framebuffer		
Navigation		
Time Utility APIs		
Time Representation APIs		
Time Synchronization APIs		
Time Units Helpers		
Utility Functions	0.1.0	v2.4.0
Xtensa APIs		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Xtensa Internal APIs		
Xtensa Memory Management Unit (MMU) APIs		
Xtensa Memory Protection Unit (MPU) APIs		

2.6.2 API Lifecycle

Developers using Zephyr's APIs need to know how long they can trust that a given API will not change in future releases. At the same time, developers maintaining and extending Zephyr's APIs need to be able to introduce new APIs that aren't yet fully proven, and to potentially retire old APIs when they're no longer optimal or supported by the underlying platforms.

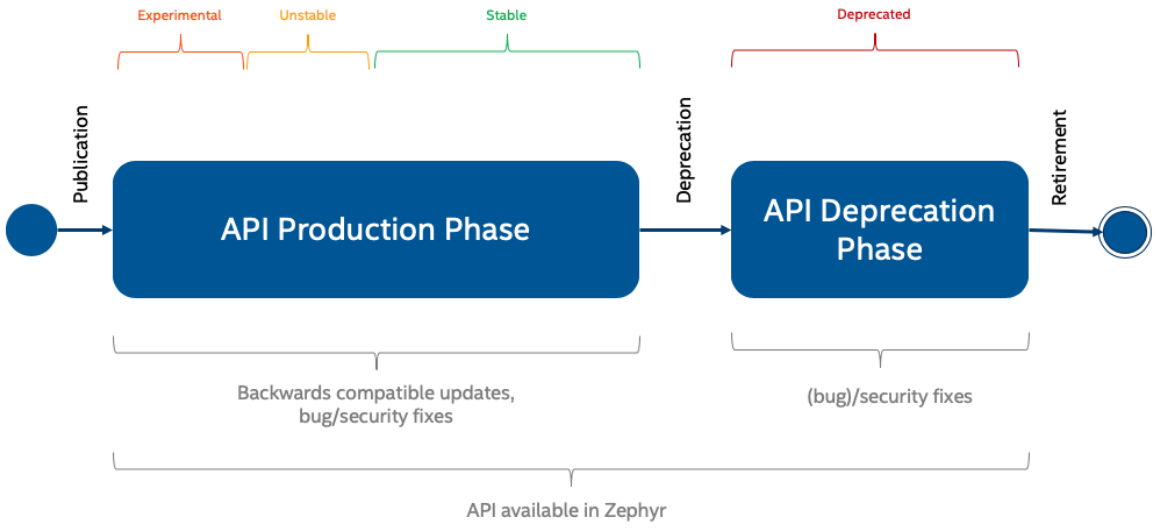


Fig. 2: API Life Cycle

An up-to-date table of all APIs and their maturity level can be found in the *API Overview* page.

Experimental

Experimental APIs denote that a feature was introduced recently, and may change or be removed in future versions. Try it out and provide feedback to the community via the Developer mailing list.

The following requirements apply to all new APIs:

- Documentation of the API (usage) explaining its design and assumptions, how it is to be used, current implementation limitations, and future potential, if appropriate.
- The API introduction should be accompanied by at least one implementation of said API (in the case of peripheral APIs, this corresponds to one driver)
- At least one sample using the new API (may only build on one single board)

When introducing a new and experimental API, mark the API version in the headers where the API is defined. An experimental API shall have a version where the minor version is up to one (0.1.z). (see *API Overview*)

Peripheral APIs (Hardware Related) When introducing an API (public header file with documentation) for a new peripheral or driver subsystem, review of the API is enforced and is driven by the Architecture working group consisting of representatives from different vendors.

The API shall be promoted to `unstable` when it has at least two implementations on different hardware platforms.

Unstable

The API is in the process of settling, but has not yet had sufficient real-world testing to be considered stable. The API is considered generic in nature and can be used on different hardware platforms.

When the API changes status to unstable API, mark the API version in the headers where the API is defined. Unstable APIs shall have a version where the minor version is larger than one (0.y.z | y > 1). (see *API Overview*)

> **Note**
>
> Changes will not be announced.

Peripheral APIs (Hardware Related) The API shall be promoted from `experimental` to `unstable` when it has at least two implementations on different hardware platforms.

Hardware Agnostic APIs For hardware agnostic APIs, multiple applications using it are required to promote an API from `experimental` to `unstable`.

Stable

The API has proven satisfactory, but cleanup in the underlying code may cause minor changes. Backwards-compatibility will be maintained if reasonable.

An API can be declared `stable` after fulfilling the following requirements:

- Test cases for the new API with 100% coverage
- Complete documentation in code. All public interfaces shall be documented and available in online documentation.
- The API has been in-use and was available in at least 2 development releases
- Stable APIs can get backward compatible updates, bug fixes and security fixes at any time.

In order to declare an API `stable`, the following steps need to be followed:

1. A Pull Request must be opened that changes the corresponding entry in the *API Overview* table
2. An email must be sent to the `devel` mailing list announcing the API upgrade request
3. The Pull Request must be submitted for discussion in the next Zephyr Architecture meeting where, barring any objections, the Pull Request will be merged

When the API changes status to stable API, mark the API version in the headers where the API is defined. Stable APIs shall have a version where the major version is one or larger (x.y.z | x >= 1). (see *API Overview*)

Introducing breaking API changes A stable API, as described above, strives to remain backwards-compatible through its life-cycle. There are however cases where fulfilling this objective prevents technical progress, or is simply unfeasible without unreasonable burden on the maintenance of the API and its implementation(s).

A breaking API change is defined as one that forces users to modify their existing code in order to maintain the current behavior of their application. The need for recompilation of applications (without changing the application itself) is not considered a breaking API change.

In order to restrict and control the introduction of a change that breaks the promise of backwards compatibility, the following steps must be followed whenever such a change is considered necessary in order to accept it in the project:

1. An *RFC issue* must be opened on GitHub with the following content:

   ```
   Title:     RFC: Breaking API Change: <subsystem>
   Contents:  - Problem Description:
                - Background information on why the change is required
              - Proposed Change (detailed):
                - Brief description of the API change
              - Detailed RFC:
                - Function call changes
                - Device Tree changes (source and bindings)
                - Kconfig option changes
              - Dependencies:
                - Impact to users of the API, including the steps required
                  to adapt out-of-tree users of the API to the change
   ```

   Instead of a written description of the changes, the RFC issue may link to a Pull Request containing those changes in code form.

2. The RFC issue must be labeled with the GitHub `Breaking API Change` label
3. The RFC issue must be submitted for discussion in the next Zephyr Architecture meeting
4. An email must be sent to the `devel` mailing list with a subject identical to the RFC issue title and that links to the RFC issue

The RFC will then receive feedback through issue comments and will also be discussed in the Zephyr Architecture meeting, where the stakeholders and the community at large will have a chance to discuss it in detail.

Finally, and if not done as part of the first step, a Pull Request must be opened on GitHub. It is left to the person proposing the change to decide whether to introduce both the RFC and the Pull Request at the same time or to wait until the RFC has gathered consensus enough so that the implementation can proceed with confidence that it will be accepted. The Pull Request must include the following:

- A title that matches the RFC issue
- A link to the RFC issue
- The actual changes to the API
  - Changes to the API header file
  - Changes to the API implementation(s)
  - Changes to the relevant API documentation
  - Changes to Device Tree source and bindings
- The changes required to adapt in-tree users of the API to the change. Depending on the scope of this task this might require additional help from the corresponding maintainers
- An entry in the “API Changes” section of the release notes for the next upcoming release

- The labels `API`, `Breaking API Change` and `Release Notes`, as well as any others that are applicable
- The label `Architecture Review` if the RFC was not yet discussed and agreed upon in Zephyr Architecture meeting

Once the steps above have been completed, the outcome of the proposal will depend on the approval of the actual Pull Request by the maintainer of the corresponding subsystem. As with any other Pull Request, the author can request for it to be discussed and ultimately even voted on in the Zephyr TSC meeting.

If the Pull Request is merged then an email must be sent to the `devel` and `user` mailing lists informing them of the change.

The API version shall be changed to signal backward incompatible changes. This is achieved by incrementing the major version (X.y.z | X > 1). It MAY also include minor and patch level changes. Patch and minor versions MUST be reset to 0 when major version is incremented. (see *API Overview*)

Note

Breaking API changes will be listed and described in the migration guide.

Deprecated

Note

Unstable APIs can be removed without deprecation at any time. Deprecation and removal of APIs will be announced in the "API Changes" section of the release notes.

The following are the requirements for deprecating an existing API:

- Deprecation Time (stable APIs): 2 Releases The API needs to be marked as deprecated in at least two full releases. For example, if an API was first deprecated in release 1.14, it will be ready to be removed in 1.16 at the earliest. There may be special circumstances, determined by the Architecture working group, where an API is deprecated sooner.
- What is required when deprecating:
  - Mark as deprecated. This can be done by using the compiler itself (`__deprecated` for function declarations and `__DEPRECATED_MACRO` for macro definitions), or by introducing a Kconfig option (typically one that contains the `DEPRECATED` word in it) that, when enabled, reverts the APIs back to their previous form
  - Document the deprecation
  - Include the deprecation in the "API Changes" of the release notes for the next upcoming release
  - Code using the deprecated API needs to be modified to remove usage of said API
  - The change needs to be atomic and bisectable
  - Create a GitHub issue to track the removal of the deprecated API, and add it to the roadmap targeting the appropriate release (in the example above, 1.16).

During the deprecation waiting period, the API will be in the `deprecated` state. The Zephyr maintainers will track usage of deprecated APIs on `docs.zephyrproject.org` and support developers migrating their code. Zephyr will continue to provide warnings:

- API documentation will inform users that the API is deprecated.

- Attempts to use a deprecated API at build time will log a warning to the console.

Retired

In this phase, the API is removed.

The target removal date is 2 releases after deprecation is announced. The Zephyr maintainers will decide when to actually remove the API: this will depend on how many developers have successfully migrated from the deprecated API, and on how urgently the API needs to be removed.

If it's OK to remove the API, it will be removed. The maintainers will remove the corresponding documentation, and communicate the removal in the usual ways: the release notes, mailing lists, Github issues and pull-requests.

If it's not OK to remove the API, the maintainers will continue to support migration and update the roadmap with the aim to remove the API in the next release.

2.6.3 API Design Guidelines

Zephyr development and evolution is a group effort, and to simplify maintenance and enhancements there are some general policies that should be followed when developing a new capability or interface.

Using Callbacks

Many APIs involve passing a callback as a parameter or as a member of a configuration structure. The following policies should be followed when specifying the signature of a callback:

- The first parameter should be a pointer to the object most closely associated with the callback. In the case of device drivers this would be `const struct device *dev`. For library functions it may be a pointer to another object that was referenced when the callback was provided.
- The next parameter(s) should be additional information specific to the callback invocation, such as a channel identifier, new status value, and/or a message pointer followed by the message length.
- The final parameter should be a `void *user_data` pointer carrying context that allows a shared callback function to locate additional material necessary to process the callback.

An exception to providing `user_data` as the last parameter may be allowed when the callback itself was provided through a structure that will be embedded in another structure. An example of such a case is *gpio_callback*, normally defined within a data structure specific to the code that also defines the callback function. In those cases further context can accessed by the callback indirectly by *CONTAINER_OF*.

Examples

- The requirements of *k_timer_expiry_t* invoked when a system timer alarm fires are satisfied by:

```
void handle_timeout(struct k_timer *timer)
{ ... }
```

  The assumption here, as with *gpio_callback*, is that the timer is embedded in a structure reachable from *CONTAINER_OF* that can provide additional context to the callback.

- The requirements of *counter_alarm_callback_t* invoked when a counter device alarm fires are satisfied by:

```
void handle_alarm(const struct device *dev,
                  uint8_t chan_id,
                  uint32_t ticks,
                  void *user_data)
{ ... }
```

This provides more complete useful information, including which counter channel timed-out and the counter value at which the timeout occurred, as well as user context which may or may not be the *counter_alarm_cfg* used to register the callback, depending on user needs.

Conditional Data and APIs

APIs and libraries may provide features that are expensive in RAM or code size but are optional in the sense that some applications can be implemented without them. Examples of such feature include `capturing a timestamp` or `providing an alternative interface`. The developer in coordination with the community must determine whether enabling the features is to be controllable through a Kconfig option.

In the case where a feature is determined to be optional the following practices should be followed.

- Any data that is accessed only when the feature is enabled should be conditionally included via `#ifdef CONFIG_MYFEATURE` in the structure or union declaration. This reduces memory use for applications that don't need the capability.
- Function declarations that are available only when the option is enabled should be provided unconditionally. Add a note in the description that the function is available only when the specified feature is enabled, referencing the required Kconfig symbol by name. In the cases where the function is used but not enabled the definition of the function shall be excluded from compilation, so references to the unsupported API will result in a link-time error.
- Where code specific to the feature is isolated in a source file that has no other content that file should be conditionally included in `CMakeLists.txt`:

```
zephyr_sources_ifdef(CONFIG_MYFEATURE foo_funcs.c)
```

- Where code specific to the feature is part of a source file that has other content the feature-specific code should be conditionally processed using `#ifdef CONFIG_MYFEATURE`.

The Kconfig flag used to enable the feature should be added to the `PREDEFINED` variable in `doc/zephyr.doxyfile.in` to ensure the conditional API and functions appear in generated documentation.

Return Codes

Implementations of an API, for example an API for accessing a peripheral might implement only a subset of the functions that is required for minimal operation. A distinction is needed between APIs that are not supported and those that are not implemented or optional:

- APIs that are supported but not implemented shall return `-ENOSYS`.
- Optional APIs that are not supported by the hardware should be implemented and the return code in this case shall be `-ENOTSUP`.
- When an API is implemented, but the particular combination of options requested in the call cannot be satisfied by the implementation the call shall return `-ENOTSUP`. (For example, a request for a level-triggered GPIO interrupt on hardware that supports only edge-triggered interrupts)

2.6.4 API Terminology

The following terms may be used as shorthand API tags to indicate the allowed calling context (thread, ISR, pre-kernel), the effect of a call on the current thread state, and other behavioral characteristics.

reschedule
: if executing the function reaches a reschedule point

sleep
: if executing the function can cause the invoking thread to sleep

no-wait
: if a parameter to the function can prevent the invoking thread from trying to sleep

isr-ok
: if the function can be safely called and will have its specified effect whether invoked from interrupt or thread context

pre-kernel-ok
: if the function can be safely called before the kernel has been fully initialized and will have its specified effect when invoked from that context.

async
: if the function may return before the operation it initializes is complete (i.e. function return and operation completion are asynchronous)

supervisor
: if the calling thread must have supervisor privileges to execute the function

Details on the behavioral impact of each attribute are in the following sections.

reschedule

The reschedule attribute is used on a function that can reach a *reschedule point* within its execution.

Details The significance of this attribute is that when a rescheduling function is invoked by a thread it is possible for that thread to be suspended as a consequence of a higher-priority thread being made ready. Whether the suspension actually occurs depends on the operation associated with the reschedule point and the relative priorities of the invoking thread and the head of the ready queue.

Note that in the case of timeslicing, or reschedule points executed from interrupts, any thread may be suspended in any function.

Functions that are not **reschedule** may be invoked from either thread or interrupt context.

Functions that are **reschedule** may be invoked from thread context.

Functions that are **reschedule** but not **sleep** may be invoked from interrupt context.

sleep

The sleep attribute is used on a function that can cause the invoking thread to *sleep*.

Explanation This attribute is of relevance specifically when considering applications that use only non-preemptible threads, because the kernel will not replace a running cooperative-only thread at a reschedule point unless that thread has explicitly invoked an operation that caused it to sleep.

This attribute does not imply the function will sleep unconditionally, but that the operation may require an invoking thread that would have to suspend, wait, or invoke *k_yield()* before it can complete its operation. This behavior may be mediated by **no-wait**.

Functions that are **sleep** are implicitly **reschedule**.

Functions that are **sleep** may be invoked from thread context.

Functions that are **sleep** may be invoked from interrupt and pre-kernel contexts if and only if invoked in **no-wait** mode.

no-wait

The no-wait attribute is used on a function that is also **sleep** to indicate that a parameter to the function can force an execution path that will not cause the invoking thread to sleep.

Explanation The paradigmatic case of a no-wait function is a function that takes a timeout, to which *K_NO_WAIT* can be passed. The semantics of this special timeout value are to execute the function's operation as long as it can be completed immediately, and to return an error code rather than sleep if it cannot.

It is use of the no-wait feature that allows functions like *k_sem_take()* to be invoked from ISRs, since it is not permitted to sleep in interrupt context.

A function with a no-wait path does not imply that taking that path guarantees the function is synchronous.

Functions with this attribute may be invoked from interrupt and pre-kernel contexts only when the parameter selects the no-wait path.

isr-ok

The isr-ok attribute is used on a function to indicate that it works whether it is being invoked from interrupt or thread context.

Explanation Any function that is not **sleep** is inherently **isr-ok**. Functions that are **sleep** are **isr-ok** if the implementation ensures that the documented behavior is implemented even if called from an interrupt context. This may be achieved by having the implementation detect the calling context and transfer the operation that would sleep to a thread, or by documenting that when invoked from a non-thread context the function will return a specific error (generally `-EWOULDBLOCK`).

Note that a function that is **no-wait** is safe to call from interrupt context only when the no-wait path is selected. **isr-ok** functions need not provide a no-wait path.

pre-kernel-ok

The pre-kernel-ok attribute is used on a function to indicate that it works as documented even when invoked before the kernel main thread has been started.

Explanation This attribute is similar to **isr-ok** in function, but is intended for use by any API that is expected to be called in *DEVICE_DEFINE()* or SYS_INIT() calls that may be invoked with PRE_KERNEL_1 or PRE_KERNEL_2 initialization levels.

Generally a function that is **pre-kernel-ok** checks *k_is_pre_kernel()* when determining whether it can fulfill its required behavior. In many cases it would also check *k_is_in_isr()* so it can be **isr-ok** as well.

async

A function is **async** (i.e. asynchronous) if it may return before the operation it initiates has completed. An asynchronous function will generally provide a mechanism by which operation completion is reported, e.g. a callback or event.

A function that is not asynchronous is synchronous, i.e. the operation will always be complete when the function returns. As most functions are synchronous this behavior does not have a distinct attribute to identify it.

Explanation Be aware that **async** is orthogonal to context-switching. Some APIs may provide completion information through a callback, but may suspend while waiting for the resource necessary to initiate the operation; an example is *spi_transceive_signal()*.

If a function is both **no-wait** and **async** then selecting the no-wait path only guarantees that the function will not sleep. It does not affect whether the operation will be completed before the function returns.

supervisor

The supervisor attribute is relevant only in user-mode applications, and indicates that the function cannot be invoked from user mode.

2.7 Language Support

2.7.1 C Language Support

C is a general-purpose low-level programming language that is widely used for writing code for embedded systems.

Zephyr is primarily written in C and natively supports applications written in the C language. All Zephyr API functions and macros are implemented in C and available as part of the C header files under the include directory, so writing Zephyr applications in C gives the developers access to the most features.

The main() function must have the return type of int as Zephyr applications run in a "hosted" environment as defined by the C standard. Applications must return zero (0) from main. All non-zero return values are reserved.

Language Standards

Zephyr does not target a specific version of the C standards; however, the Zephyr codebase makes extensive use of the features newly introduced in the 1999 release of the ISO C standard (ISO/IEC 9899:1999, hereinafter referred to as C99) such as those listed below, effectively requiring the use of a compiler toolchain that supports the C99 standard and above:

- inline functions
- standard boolean types (`bool` in `<stdbool.h>`)
- fixed-width integer types (`[u]intN_t` in `<stdint.h>`)
- designated initializers
- variadic macros
- `restrict` qualification

Some Zephyr components make use of the features newly introduced in the 2011 release of the ISO C standard (ISO/IEC 9899:2011, hereinafter referred to as C11) such as the type-generic expressions using the `_Generic` keyword. For example, the *`cbprintf()`* component, used as the default formatted output processor for Zephyr, makes use of the C11 type-generic expressions, and this effectively requires most Zephyr applications to be compiled using a compiler toolchain that supports the C11 standard and above.

In summary, it is recommended to use a compiler toolchain that supports at least the C11 standard for developing with Zephyr. It is, however, important to note that some optional Zephyr components and external modules may make use of the C language features that have been introduced in more recent versions of the standards, in which case it will be necessary to use a more up-to-date compiler toolchain that supports such standards.

Standard Library

The C Standard Library is an integral part of any C program, and Zephyr provides the support for a number of different C libraries for the applications to choose from, depending on the compiler toolchain being used to build the application.

Common C library code Zephyr provides some C library functions that are designed to be used in conjunction with multiple C libraries. These either provide functions not available in multiple C libraries or are designed to replace functionality in the C library with code better suited for use in the Zephyr environment

Time function This provides an implementation of the standard C function, `time()`, relying on the Zephyr function, `clock_gettime()`. This function can be enabled by selecting `COMMON_LIBC_TIME`.

Dynamic Memory Management The common dynamic memory management implementation can be enabled by selecting the `CONFIG_COMMON_LIBC_MALLOC` in the application configuration file.

The common C library internally uses the *kernel memory heap API* to manage the memory heap used by the standard dynamic memory management interface functions such as `malloc()` and `free()`.

The internal memory heap is normally located in the `.bss` section. When userspace is enabled, however, it is placed in a dedicated memory partition called `z_malloc_partition`, which can be accessed from the user mode threads. The size of the internal memory heap is specified by the `CONFIG_COMMON_LIBC_MALLOC_ARENA_SIZE`.

The default heap size for applications using the common C library is zero (no heap). For other C library users, if there is an MMU present, then the default heap is 16kB. Otherwise, the heap uses all available memory.

There are also separate controls to select `calloc()` (`COMMON_LIBC_CALLOC`) and `reallocarray()` (`COMMON_LIBC_REALLOCARRAY`). Both of these are enabled by default as that doesn't impact memory usage in applications not using them.

The standard dynamic memory management interface functions implemented by the common C library are thread safe and may be simultaneously called by multiple threads. These functions are implemented in `lib/libc/common/source/stdlib/malloc.c`.

Minimal libc The most basic C library, named "minimal libc", is part of the Zephyr codebase and provides the minimal subset of the standard C library required to meet the needs of Zephyr and its subsystems, primarily in the areas of string manipulation and display.

It is very low footprint and is suitable for projects that do not rely on less frequently used portions of the ISO C standard library. It can also be used with a number of different toolchains.

The minimal libc implementation can be found in `lib/libc/minimal` in the main Zephyr tree.

Functions The minimal libc implements the minimal subset of the ISO/IEC 9899:2011 standard C library functions required to meet the needs of the Zephyr kernel, as defined by the *Coding Guidelines Rule A.4*.

Formatted Output The minimal libc does not implement its own formatted output processor; instead, it maps the C standard formatted output functions such as `printf` and `sprintf` to the *`cbprintf()`* function, which is Zephyr's own C99-compatible formatted output implementation.

For more details, refer to the *Formatted Output* OS service documentation.

Dynamic Memory Management The minimal libc uses the malloc api family implementation provided by the *common C library*, which itself is built upon the *kernel memory heap API*.

Error numbers Error numbers are used throughout Zephyr APIs to signal error conditions as return values from functions. They are typically returned as the negative value of the integer literals defined in this section, and are defined in the `errno.h` header file.

A subset of the error numbers defined in the POSIX errno.h specification and other de-facto standard sources have been added to the minimal libc.

A conscious effort is made in Zephyr to keep the values of the minimal libc error numbers consistent with the different implementations of the C standard libraries supported by Zephyr. The minimal libc `errno.h` is checked against that of the *Newlib* to ensure that the error numbers are kept aligned.

Below is a list of the error number definitions. For the actual numeric values please refer to errno.h.

group `system_errno`

System error numbers Error codes returned by functions.

Includes a list of those defined by IEEE Std 1003.1-2017.

Defines

`errno`

`EPERM`

Not owner.

`ENOENT`
: No such file or directory.

`ESRCH`
: No such context.

`EINTR`
: Interrupted system call.

`EIO`
: I/O error.

`ENXIO`
: No such device or address.

`E2BIG`
: Arg list too long.

`ENOEXEC`
: Exec format error.

`EBADF`
: Bad file number.

`ECHILD`
: No children.

`EAGAIN`
: No more contexts.

`ENOMEM`
: Not enough core.

`EACCES`
: Permission denied.

`EFAULT`
: Bad address.

`ENOTBLK`
: Block device required.

`EBUSY`
: Mount device busy.

`EEXIST`
: File exists.

`EXDEV`
: Cross-device link.

`ENODEV`
: No such device.

`ENOTDIR`
: Not a directory.

`EISDIR`
: Is a directory.

`EINVAL`
: Invalid argument.

`ENFILE`
: File table overflow.

`EMFILE`
: Too many open files.

`ENOTTY`
: Not a typewriter.

`ETXTBSY`
: Text file busy.

`EFBIG`
: File too large.

`ENOSPC`
: No space left on device.

`ESPIPE`
: Illegal seek.

`EROFS`
: Read-only file system.

`EMLINK`
: Too many links.

`EPIPE`
: Broken pipe.

`EDOM`
: Argument too large.

ERANGE
: Result too large.

ENOMSG
: Unexpected message type.

EDEADLK
: Resource deadlock avoided.

ENOLCK
: No locks available.

ENOSTR
: STREAMS device required.

ENODATA
: Missing expected message data.

ETIME
: STREAMS timeout occurred.

ENOSR
: Insufficient memory.

EPROTO
: Generic STREAMS error.

EBADMSG
: Invalid STREAMS message.

ENOSYS
: Function not implemented.

ENOTEMPTY
: Directory not empty.

ENAMETOOLONG
: File name too long.

ELOOP
: Too many levels of symbolic links.

EOPNOTSUPP
: Operation not supported on socket.

EPFNOSUPPORT
: Protocol family not supported.

ECONNRESET
: Connection reset by peer.

ENOBUFS
: No buffer space available.

EAFNOSUPPORT
: Addr family not supported.

EPROTOTYPE
: Protocol wrong type for socket.

ENOTSOCK
: Socket operation on non-socket.

ENOPROTOOPT
: Protocol not available.

ESHUTDOWN
: Can't send after socket shutdown.

ECONNREFUSED
: Connection refused.

EADDRINUSE
: Address already in use.

ECONNABORTED
: Software caused connection abort.

ENETUNREACH
: Network is unreachable.

ENETDOWN
: Network is down.

ETIMEDOUT
: Connection timed out.

EHOSTDOWN
: Host is down.

EHOSTUNREACH
: No route to host.

EINPROGRESS
: Operation now in progress.

`EALREADY`
: Operation already in progress.

`EDESTADDRREQ`
: Destination address required.

`EMSGSIZE`
: Message size.

`EPROTONOSUPPORT`
: Protocol not supported.

`ESOCKTNOSUPPORT`
: Socket type not supported.

`EADDRNOTAVAIL`
: Can't assign requested address.

`ENETRESET`
: Network dropped connection on reset.

`EISCONN`
: Socket is already connected.

`ENOTCONN`
: Socket is not connected.

`ETOOMANYREFS`
: Too many references: can't splice.

`ENOTSUP`
: Unsupported value.

`EILSEQ`
: Illegal byte sequence.

`EOVERFLOW`
: Value overflow.

`ECANCELED`
: Operation canceled.

`EWOULDBLOCK`
: Operation would block.

Newlib Newlib is a complete C library implementation written for the embedded systems. It is a separate open source project and is not included in source code form with Zephyr. Instead, the *Zephyr SDK* includes a precompiled library for each supported architecture (`libc.a` and `libm.a`).

Note

Other 3rd-party toolchains, such as *GNU Arm Embedded*, also bundle the Newlib as a precompiled library.

Zephyr implements the "API hook" functions that are invoked by the C standard library functions in the Newlib. These hook functions are implemented in `lib/libc/newlib/libc-hooks.c` and translate the library internal system calls to the equivalent Zephyr API calls.

Types of Newlib The Newlib included in the *Zephyr SDK* comes in two versions: 'full' and 'nano' variants.

Full Newlib The Newlib full variant (`libc.a` and `libm.a`) is the most capable variant of the Newlib available in the Zephyr SDK, and supports almost all standard C library features. It is optimized for performance (prefers performance over code size) and its footprint is significantly larger than the nano variant.

This variant can be enabled by selecting the `CONFIG_NEWLIB_LIBC` and de-selecting the `CONFIG_NEWLIB_LIBC_NANO` in the application configuration file.

Nano Newlib The Newlib nano variant (`libc_nano.a` and `libm_nano.a`) is the size-optimized version of the Newlib, and supports all features that the full variant supports except the new format specifiers introduced in C99, such as the `char`, `long long` type format specifiers (i.e. `%hhX` and `%llX`).

This variant can be enabled by selecting the `CONFIG_NEWLIB_LIBC` and `CONFIG_NEWLIB_LIBC_NANO` in the application configuration file.

Note that the Newlib nano variant is not available for all architectures. The availability of the nano variant is specified by the `CONFIG_HAS_NEWLIB_LIBC_NANO`.

Formatted Output Newlib supports all standard C formatted input and output functions, including `printf`, `fprintf`, `sprintf` and `sscanf`.

The Newlib formatted input and output function implementation supports all format specifiers defined by the C standard with the following exceptions:

- Floating point format specifiers (e.g. `%f`) require `CONFIG_NEWLIB_LIBC_FLOAT_PRINTF` and `CONFIG_NEWLIB_LIBC_FLOAT_SCANF` to be enabled.
- C99 format specifiers are not supported in the Newlib nano variant (i.e. `%hhX` for `char`, `%llX` for `long long`, `%jX` for `intmax_t`, `%zX` for `size_t`, `%tX` for `ptrdiff_t`).

Dynamic Memory Management Newlib implements an internal heap allocator to manage the memory blocks used by the standard dynamic memory management interface functions (for example, `malloc()` and `free()`).

The internal heap allocator implemented by the Newlib may vary across the different types of the Newlib used. For example, the heap allocator implemented in the Full Newlib (`libc.a` and `libm.a`) of the Zephyr SDK requests larger memory chunks to the operating system and has a significantly higher minimum memory requirement compared to that of the Nano Newlib (`libc_nano.a` and `libm_nano.a`).

The only interface between the Newlib dynamic memory management functions and the Zephyr-side libc hooks is the `sbrk()` function, which is used by the Newlib to manage the size of the memory pool reserved for its internal heap allocator.

The `_sbrk()` hook function, implemented in `libc-hooks.c`, handles the memory pool size change requests from the Newlib and ensures that the Newlib internal heap allocator memory pool size does not exceed the amount of available memory space by returning an error when the system is out of memory.

When userspace is enabled, the Newlib internal heap allocator memory pool is placed in a dedicated memory partition called `z_malloc_partition`, which can be accessed from the user mode threads.

The amount of memory space available for the Newlib heap depends on the system configurations:

- When MMU is enabled (`CONFIG_MMU` is selected), the amount of memory space reserved for the Newlib heap is set by the size of the free memory space returned by the *`k_mem_free_get()`* function or the `CONFIG_NEWLIB_LIBC_MAX_MAPPED_REGION_SIZE`, whichever is the smallest.
- When MPU is enabled and the MPU requires power-of-two partition size and address alignment (`CONFIG_NEWLIB_LIBC_ALIGNED_HEAP_SIZE` is set to a non-zero value), the amount of memory space reserved for the Newlib heap is set by the `CONFIG_NEWLIB_LIBC_ALIGNED_HEAP_SIZE`.
- Otherwise, the amount of memory space reserved for the Newlib heap is equal to the amount of free (unallocated) memory in the SRAM region.

The standard dynamic memory management interface functions implemented by the Newlib are thread safe and may be simultaneously called by multiple threads.

Picolibc Picolibc is a complete C library implementation written for the embedded systems, targeting C17 (ISO/IEC 9899:2018) and POSIX 2018 (IEEE Std 1003.1-2017) standards. Picolibc is an external open source project which is provided for Zephyr as a module, and included as part of the *Zephyr SDK* in precompiled form for each supported architecture (`libc.a`).

> **Note**
>
> Picolibc is also available for other 3rd-party toolchains, such as *GNU Arm Embedded*.

Zephyr implements the "API hook" functions that are invoked by the C standard library functions in the Picolibc. These hook functions are implemented in `lib/libc/picolibc/libc-hooks.c` and translate the library internal system calls to the equivalent Zephyr API calls.

Picolibc Module When built as a Zephyr module, there are several configuration knobs available to adjust the feature set in the library, balancing what the library supports versus the code size of the resulting functions. Because the standard C++ library must be compiled for the target C library, the Picolibc module cannot be used with applications which use the standard C++ library. Building the Picolibc module will increase the time it takes to compile the application.

The Picolibc module can be enabled by selecting `CONFIG_PICOLIBC_USE_MODULE` in the application configuration file.

When updating the Picolibc module to a newer version, the *toolchain-bundled Picolibc in the Zephyr SDK* must also be updated to the same version.

Toolchain Picolibc Starting with version 0.16, the Zephyr SDK includes precompiled versions of Picolibc for every target architecture, along with precompiled versions of libstdc++.

The toolchain version of Picolibc can be enabled by de-selecting `CONFIG_PICOLIBC_USE_MODULE` in the application configuration file.

For every release of Zephyr, the toolchain-bundled Picolibc and the *Picolibc module* are guaranteed to be in sync when using the *recommended version of Zephyr SDK*.

Building Without Toolchain bundled Picolibc For toolchain where there is no bundled Picolibc, it is still possible to use Picolibc by building it from source. Note that any restrictions mentioned in *Picolibc Module* still apply.

To build without toolchain bundled Picolibc, the toolchain must enable `CONFIG_PICOLIBC_SUPPORTED`. For example, this needs to be added to the toolchain Kconfig file:

```
config TOOLCHAIN_<name>_PICOLIBC_SUPPORTED
   def_bool y
   select PICOLIBC_SUPPORTED
```

By enabling `CONFIG_PICOLIBC_SUPPORTED`, the build system would automatically build Picolibc from source with its module when there is no toolchain bundled Picolibc.

Formatted Output Picolibc supports all standard C formatted input and output functions, including `printf()`, `fprintf()`, `sprintf()` and `sscanf()`.

Picolibc formatted input and output function implementation supports all format specifiers defined by the C17 and POSIX 2018 standards with the following exceptions:

- Floating point format specifiers (e.g. `%f`) require `CONFIG_PICOLIBC_IO_FLOAT`.
- Long long format specifiers (e.g. `%lld`) require `CONFIG_PICOLIBC_IO_LONG_LONG`. This option is automatically enabled with `CONFIG_PICOLIBC_IO_FLOAT`.

Printk, cbprintf and friends When using Picolibc, Zephyr formatted output functions are implemented in terms of stdio calls. This includes:

- printk, snprintk and vsnprintk
- cbprintf and cbvprintf
- fprintfcb, vfprintfcb, printfcb, vprintfcb, snprintfcb and vsnprintfcb

When using tagged args (`CONFIG_CBPRINTF_PACKAGE_SUPPORT_TAGGED_ARGUMENTS` and `CBPRINTF_PACKAGE_ARGS_ARE_TAGGED`), calls to cbpprintf will not use Picolibc, so formatting of output using those code will differ from Picolibc results as the cbprintf functions are not completely C/POSIX compliant.

Math Functions Picolibc provides full C17/IEEE STD 754-2019 support for float, double and long double math operations, except for long double versions of the Bessel functions.

Thread Local Storage Picolibc uses Thread Local Storage (TLS) (where supported) for data which is supposed to remain local to each thread, like *errno*. This means that TLS support is enabled when using Picolibc. As all TLS variables are allocated out of the thread stack area, this can affect stack size requirements by a few bytes.

C Library Local Variables Picolibc uses a few internal variables for things like heap management. These are collected in a dedicated memory partition called `z_libc_partition`. Applications using `CONFIG_USERSPACE` and memory domains must ensure that this partition is included in any domain active during Picolibc calls.

Dynamic Memory Management Picolibc uses the malloc api family implementation provided by the *common C library*, which itself is built upon the *kernel memory heap API*.

Formatted Output

C defines standard formatted output functions such as `printf` and `sprintf` and these functions are implemented by the C standard libraries.

Each C standard library has its own set of requirements and configurations for selecting the formatted output modes and capabilities. Refer to each C standard library documentation for more details.

Dynamic Memory Management

C defines a standard dynamic memory management interface (for example, `malloc()` and `free()`) and these functions are implemented by the C standard libraries.

While the details of the dynamic memory management implementation varies across different C standard libraries, all supported libraries must conform to the following conventions. Every supported C standard library shall:

- manage its own memory heap either internally or by invoking the hook functions (for example, `sbrk()`) implemented in `libc-hooks.c`.
- maintain the architecture- and memory region-specific alignment requirements for the memory blocks allocated by the standard dynamic memory allocation interface (for example, `malloc()`).
- allocate memory blocks inside the `z_malloc_partition` memory partition when userspace is enabled. See *Pre-defined Memory Partitions*.

For more details regarding the C standard library-specific memory management implementation, refer to each C standard library documentation.

> **Note**
>
> Native Zephyr applications should use the *memory management API* supported by the Zephyr kernel such as *`k_malloc()`* in order to take advantage of the advanced features that they offer.
>
> C standard dynamic memory management interface functions such as `malloc()` should be used only by the portable applications and libraries that target multiple operating systems.

2.7.2 C++ Language Support

C++ is a general-purpose object-oriented programming language that is based on the C language.

Enabling C++ Support

Zephyr supports applications written in both C and C++. However, to use C++ in an application you must configure Zephyr to include C++ support by selecting the `CONFIG_CPP` in the application configuration file.

To enable C++ support, the compiler toolchain must also include a C++ compiler and the included compiler must be supported by the Zephyr build system. The *Zephyr SDK*, which includes the GNU C++ Compiler (part of GCC), is supported by Zephyr, and the features and their availability documented here assume the use of the Zephyr SDK.

The default C++ standard level (i.e. the language enforced by the compiler flags passed) for Zephyr apps is C++11. Other standards are available via kconfig choice, for example `CONFIG_STD_CPP98`. The oldest standard supported and tested in Zephyr is C++98.

When compiling a source file, the build system selects the C++ compiler based on the suffix (extension) of the files. Files identified with either a **cpp** or a **cxx** suffix are compiled using the C++ compiler. For example, `myCplusplusApp.cpp` is compiled using C++.

The C++ standard requires the `main()` function to have the return type of `int`. Your `main()` must be defined as `int main(void)`. Zephyr ignores the return value from main, so applications should not return status information and should, instead, return zero.

Note

Do not use C++ for kernel, driver, or system initialization code.

Language Features

Zephyr currently provides only a subset of C++ functionality. The following features are *not* supported:

- Static global object destruction
- OS-specific C++ standard library classes (e.g. `std::thread`, `std::mutex`)

While not an exhaustive list, support for the following functionality is included:

- Inheritance
- Virtual functions
- Virtual tables
- Static global object constructors
- Dynamic object management with the **new** and **delete** operators
- Exceptions
- RTTI (runtime type information)
- Standard Template Library (STL)

Static global object constructors are initialized after the drivers are initialized but before the application `main()` function. Therefore, use of C++ is restricted to application code.

In order to make use of the C++ exceptions, the `CONFIG_CPP_EXCEPTIONS` must be selected in the application configuration file.

Zephyr Minimal C++ Library

Zephyr minimal C++ library (`lib/cpp/minimal`) provides a minimal subset of the C++ standard library and application binary interface (ABI) functions to enable basic C++ language support. This includes:

- `new` and `delete` operators
- virtual function stub and vtables
- static global initializers for global constructors

The scope of the minimal C++ library is strictly limited to providing the basic C++ language support, and it does not implement any Standard Template Library (STL) classes and functions. For

this reason, it is only suitable for use in the applications that implement their own (non-standard) class library and do not rely on the Standard Template Library (STL) components.

Any application that makes use of the Standard Template Library (STL) components, such as `std::string` and `std::vector`, must enable the C++ standard library support.

C++ Standard Library

The C++ Standard Library is a collection of classes and functions that are part of the ISO C++ standard (`std` namespace).

Zephyr does not include any C++ standard library implementation in source code form. Instead, it allows configuring the build system to link against the pre-built C++ standard library included in the C++ compiler toolchain.

To enable C++ standard library, select an applicable toolchain-specific C++ standard library type from the `CONFIG_LIBCPP_IMPLEMENTATION` in the application configuration file.

For instance, when building with the *Zephyr SDK*, the build system can be configured to link against the GNU C++ Library (`libstdc++.a`), which is a fully featured C++ standard library that provides all features required by the ISO C++ standard including the Standard Template Library (STL), by selecting `CONFIG_GLIBCXX_LIBCPP` in the application configuration file.

The following C++ standard libraries are supported by Zephyr:

- GNU C++ Library (`CONFIG_GLIBCXX_LIBCPP`)
- ARC MetaWare C++ Library (`CONFIG_ARCMWDT_LIBCPP`)

A Zephyr subsystem that requires the features from the full C++ standard library can select, from its config, `CONFIG_REQUIRES_FULL_LIBCPP`, which automatically selects a compatible C++ standard library unless the Kconfig symbol for a specific C++ standard library is selected.

Header files and incompatibilities between C and C++

To interact with each other, C and C++ must share code through header files: data structures, macros, static functions,... While C and C++ have a large overlap, they're different languages with known incompatibilities. C is not just a C++ subset. Standard levels (e.g.: "C+11") add another level of complexity as new features are often inspired by and copied from the other language but many years later and with subtle differences. Making things more complex, compilers often offer early prototypes of features before they become standardized. Standards can have ambiguities interpreted differently by different compilers. Compilers can have bugs and these may need workarounds. To help with this, many projects restrict themselves to a limited number of toolchains. Zephyr does not.

These compatibility issues affect header files dis-proportionally. Not just because they have to be compatible between C and C++, but also because they end up being compiled in a surprisingly high number of other source files due to *indirect* inclusion and the lack of structure and headers organization that is typical in real-world projects. So, header files are exposed to a much larger variety of toolchains and project configurations. Adding more constraints, the Zephyr project has demanding policies with respect to code style, compiler warnings, static analyzers and standard compliance (e.g.: MISRA).

Put together, all these constraints can make writing header files very challenging. The purpose of this section is to document some best "header practices" and lessons learned in a Zephyr-specific context. While a lot of the information here is not Zephyr-specific, this section is not a substitute for knowledge of C/C++ standards, textbooks and other references.

Testing Fortunately, the Zephyr project has an extensive test and CI infrastructure that provides coverage baselines, catches issues early, enforces policies and maintains such combinatorial explosions under some control. The `tests/lib/cpp/cxx/` are very useful in this context because their `testcase.yaml` configuration lets `twister` iterate quickly over a range of `-std` parameters: `-std=c++98`, `-std=c++11`, etc.

Keep in mind unused macros are not compiled.

Designated initializers Initialization macros are common in header files as they help reduce boilerplate code. C99 added initialization of `struct` and `union` types by "designated" member names instead of a list of *bare* expressions. Some GCC versions support designated initializers even in their C90 mode.

When used at a simple level, designated initializers are less error-prone, more readable and more flexible. On the other hand, C99 allowed a surprisingly large and lax set of possibilities: designated initializers can be out of order, duplicated, "nested" (`.a.x =`), various other braces can be omitted, designated initializers and not can be mixed, etc.

Twenty years later, C++20 added designated initializers to C++ but in much more restricted way; partly because a C++ `struct` is actually a `class`. As described in the C++ proposal number P0329 (which compares with C) or in any complete C++ reference, a mix is not allowed and initializers must be in order (gaps are allowed).

Interestingly, the new restrictions in C++20 can cause `gcc -std=c++20` to fail to compile code that successfully compiles with `gcc -std=c++17`. For example, `gcc -std=c++17` and older allow the C-style mix of initializers and bare expressions. This fails to compile with using `gcc -std=c++20` *with the same GCC version*.

Recommendation: to maximize compatibility across different C and C++ toolchains and standards, designated initializers in Zephyr header files should follow all C++20 rules and restrictions. Non-designated, pre-C99 initialization offers more compatibility and is also allowed but designated initialization is the more readable and preferred code style. In any case, both styles must never be mixed in the same initializer.

Warning: successful compilation is not the end of the incompatibility story. For instance, the *evaluation order* of initializer expressions is unspecified in C99! It is the (expected) left-to-right order in C++20. Other standard revisions may vary. In doubt, do not rely on evaluation order (here and elsewhere).

Anonymous unions Anonymous unions (a.k.a. "unnamed" unions) seem to have been part of C++ from its very beginning. They were not officially added to C until C11. As usual, there are differences between C and C++. For instance, C supports anonymous unions only as a member of an enclosing `struct` or `union`, empty lists `{ }` have always been allowed in C++ but they require C23, etc.

When initializing anonymous members, the expression can be enclosed in braces or not. It can be either designated or bare. For maximum portability, when initializing *anonymous unions*:

- Do *not* enclose *designated* initializers with braces. This is required by C++20 and above which perceive such braces as mixing bare expressions with (other) designated initializers and fails to compile.
- Do enclose *bare* expressions with braces. This is required by C. Maybe because C is laxer and allows many initialization possibilities and variations, so it may need such braces to disambiguate? Note C does allow omitting most braces in initializer expressions - but not in this particular case of initializing anonymous unions with bare expressions.

Some pre-C11 GCC versions support some form of anonymous unions. They unfortunately require enclosing their designated initializers with braces which conflicts with this recommendation. This can be solved with an `#ifdef __STDC_VERSION__` as demonstrated in Zephyr commit c15f029a7108 and the corresponding code review.

2.8 Optimizations

Guides on how to optimize Zephyr for performance, power and footprint.

2.8.1 Optimizing for Footprint

Stack Sizes

Stack sizes of various system threads are specified generously to allow for usage in different scenarios on as many supported platforms as possible. You should start the optimization process by reviewing all stack sizes and adjusting them for your application:

`CONFIG_ISR_STACK_SIZE`
: Set to 2048 by default

`CONFIG_MAIN_STACK_SIZE`
: Set to 1024 by default

`CONFIG_IDLE_STACK_SIZE`
: Set to 320 by default

`CONFIG_SYSTEM_WORKQUEUE_STACK_SIZE`
: Set to 1024 by default

`CONFIG_PRIVILEGED_STACK_SIZE`
: Set to 1024 by default, depends on userspace feature.

Unused Peripherals

Some peripherals are enabled by default. You can disable unused peripherals in your project configuration, for example:

```
CONFIG_GPIO=n
CONFIG_SPI=n
```

Various Debug/Informational Options

The following options are enabled by default to provide more information about the running application and to provide means for debugging and error handling:

`CONFIG_BOOT_BANNER`
: This option can be disabled to save a few bytes.

`CONFIG_DEBUG`
: This option can be disabled for production builds

MPU/MMU Support

Depending on your application and platform needs, you can disable MPU/MMU support to gain some memory and improve performance. Consider the consequences of this configuration choice though, because you'll lose advanced stack checking and support.

2.8.2 Optimization Tools

The available optimization tools let you analyse *Footprint and Memory Usage* and *Data Structures* using different build system targets.

Footprint and Memory Usage

The build system offers 3 targets to view and analyse RAM, ROM and stack usage in generated images. The tools run on the final image and give information about size of symbols and code being used in both RAM and ROM. Additionally, with features available through the compiler, we can also generate worst-case stack usage analysis.

Some of the tools mentioned in this section are organizing their output based on the physical organization of the symbols. As some symbols might be external to the project's tree structure, or might lack metadata needed to display them by name, the following top-level containers are used to group such symbols:

- Hidden - The RAM and ROM reports list all processing symbols with no matching mapped files in the Hidden category.

  This means that the file for the listed symbol was not added to the metadata file, was empty, or was undefined. The tool was unable to get the name of the function for the given symbol nor identify where it comes from.

- No paths - The RAM and ROM reports list all processing symbols with relative paths in the No paths category.

  This means that the listed symbols cannot be placed in the tree structure of the report at an absolute path under one specific file. The tool was able to get the name of the function, but it was unable to identify where it comes from.

> **Note**
>
> You can have multiple cases of the same function, and the No paths category will list the sum of these in one entry.

Build Target: ram_report List all compiled objects and their RAM usage in a tabular form with bytes per symbol and the percentage it uses. The data is grouped based on the file system location of the object in the tree and the file containing the symbol.

Use the `ram_report` target with your board, as in the following example.

Using west:

```
west build -b reel_board samples/hello_world
west build -t ram_report
```

Using CMake and ninja:

```
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=reel_board samples/hello_world

# Now run the build tool on the generated build system:
ninja -Cbuild ram_report
```

These commands will generate something similar to the output below:

```
Path                                                                  Size    %      Address
=====================================================================================
Root                                                                  4637 100.00%  -
├── (hidden)                                                             4   0.09%  -
├── (no paths)                                                        2748  59.26%  -
|   ├── _cpus_active                                                     4   0.09%  0x20000314
|   ├── _kernel                                                         32   0.69%  0x20000318
|   ├── _sw_isr_table                                                   384   8.28%  0x00006474
|   ├── cli.1                                                           16   0.35%  0x20000254
|   ├── on.2                                                             4   0.09%  0x20000264
|   ├── poll_out_lock.0                                                  4   0.09%  0x200002d4
|   ├── z_idle_threads                                                  128   2.76%  0x20000120
|   ├── z_interrupt_stacks                                             2048  44.17%  0x20000360
|   └── z_main_thread                                                   128   2.76%  0x200001a0
├── WORKSPACE                                                          184   3.97%  -
|   └── modules                                                        184   3.97%  -
|       └── hal                                                        184   3.97%  -
|           └── nordic                                                 184   3.97%  -
|               └── nrfx                                               184   3.97%  -
|                   └── drivers                                        184   3.97%  -
|                       └── src                                        184   3.97%  -
|                           ├── nrfx_clock.c                             8   0.17%  -
|                           |   └── m_clock_cb                           8   0.17%  0x200002e4
|                           ├── nrfx_gpiote.c                          132   2.85%  -
|                           |   └── m_cb                               132   2.85%  0x20000060
|                           ├── nrfx_ppi.c                               4   0.09%  -
|                           |   └── m_channels_allocated                 4   0.09%  0x200000e4
|                           └── nrfx_twim.c                             40   0.86%  -
|                               └── m_cb                                40   0.86%  0x200002ec
└── ZEPHYR_BASE                                                       1701  36.68%  -
    ├── arch                                                             5   0.11%  -
    |   └── arm                                                          5   0.11%  -
    |       └── core                                                     5   0.11%  -
    |           ├── mpu                                                  1   0.02%  -
    |           |   └── arm_mpu.c                                        1   0.02%  -
    |           |       └── static_regions_num                           1   0.02%  0x20000348
    |           └── tls.c                                                4   0.09%  -
    |               └── z_arm_tls_ptr                                    4   0.09%  0x20000240
    ├── drivers                                                        258   5.56%  -
    |   ├── ...                                                        ...    ...%
=====================================================================================
                                                                      4637
```

Build Target: rom_report List all compiled objects and their ROM usage in a tabular form with bytes per symbol and the percentage it uses. The data is grouped based on the file system location of the object in the tree and the file containing the symbol.

Use the `rom_report` target with your board, as in the following example.

Using west:

```
west build -b reel_board samples/hello_world
west build -t rom_report
```

Using CMake and ninja:

```
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=reel_board samples/hello_world

# Now run the build tool on the generated build system:
ninja -Cbuild rom_report
```

These commands will generate something similar to the output below:

```
Path                                                          Size    %      Address
=====================================================================================
Root                                                         27828 100.00%  -
├── ...                                                        ...    ...%
└── ZEPHYR_BASE                                              13558  48.72%  -
    ├── arch                                                  1766   6.35%  -
    │   └── arm                                               1766   6.35%  -
    │       └── core                                          1766   6.35%  -
    │           ├── cortex_m                                  1020   3.67%  -
    │           │   ├── fault.c                                620   2.23%  -
    │           │   │   ├── bus_fault.constprop.0              108   0.39%  0x00000749
    │           │   │   ├── mem_manage_fault.constprop.0       120   0.43%  0x000007b5
    │           │   │   ├── usage_fault.constprop.0             84   0.30%  0x000006f5
    │           │   │   ├── z_arm_fault                        292   1.05%  0x0000082d
    │           │   │   └── z_arm_fault_init                    16   0.06%  0x00000951
    │           │   ├── ...                                    ...    ...%
    ├── boards                                                  32   0.11%  -
    │   └── arm                                                 32   0.11%  -
    │       └── reel_board                                      32   0.11%  -
    │           └── board.c                                     32   0.11%  -
    │               ├── __init_board_reel_board_init             8   0.03%  0x000063e4
    │               └── board_reel_board_init                   24   0.09%  0x00000ed5
    ├── build                                                  194   0.70%  -
    │   └── zephyr                                             194   0.70%  -
    │       ├── isr_tables.c                                   192   0.69%  -
    │       │   └── _irq_vector_table                          192   0.69%  0x00000040
    │       └── misc                                             2   0.01%  -
    │           └── generated                                    2   0.01%  -
    │               └── configs.c                                2   0.01%  -
    │                   └── _ConfigAbsSyms                       2   0.01%  0x00005945
    ├── drivers                                               6282  22.57%  -
    │   ├── ...                                                ...    ...%
=====================================================================================
                                                             21652
```

Build Target: puncover This target uses a third-party tool called puncover which can be found at https://github.com/HBehrens/puncover. When this target is built, it will launch a local web server which will allow you to open a web client and browse the files and view their ROM, RAM, and stack usage.

Before you can use this target, install the puncover Python module:

```
pip3 install git+https://github.com/HBehrens/puncover --user
```

> **Warning**
>
> This is a third-party tool that might or might not be working at any given time. Please check the GitHub issues, and report new problems to the project maintainer.

After you installed the Python module, use `puncover` target with your board, as in the following example.

Using west:

```
west build -b reel_board samples/hello_world
west build -t puncover
```

Using CMake and ninja:

```
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=reel_board samples/hello_world

# Now run the build tool on the generated build system:
ninja -Cbuild puncover
```

To view worst-case stack usage analysis, build this with the `CONFIG_STACK_USAGE` enabled.

Using west:

```
west build -b reel_board samples/hello_world -- -DCONFIG_STACK_USAGE=y
west build -t puncover
```

Using CMake and ninja:

```
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=reel_board -DCONFIG_STACK_USAGE=y samples/hello_world

# Now run the build tool on the generated build system:
ninja -Cbuild puncover
```

Data Structures

Build Target: pahole Poke-a-hole (pahole) is an object-file analysis tool to find the size of the data structures, and the holes caused due to aligning the data elements to the word-size of the CPU by the compiler.

Poke-a-hole (pahole) must be installed prior to using this target. It can be obtained from https://git.kernel.org/pub/scm/devel/pahole/pahole.git and is available in the dwarves package in both fedora and ubuntu:

```
sudo apt-get install dwarves
```

Alternatively, you can get it from fedora:

```
sudo dnf install dwarves
```

After you installed the package, use `pahole` target with your board, as in the following example.

Using west:

```
west build -b reel_board samples/hello_world
west build -t pahole
```

Using CMake and ninja:

```
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=reel_board samples/hello_world

# Now run the build tool on the generated build system:
ninja -Cbuild pahole
```

Pahole will generate something similar to the output below in the console:

```
/* Used at: [...]/build/zephyr/kobject_hash.c */
/* <375> [...]/zephyr/include/zephyr/sys/dlist.h:37 */
union {
        struct _dnode *            head;               /*     0     4 */
        struct _dnode *            next;               /*     0     4 */
};
/* Used at: [...]/build/zephyr/kobject_hash.c */
```

(continues on next page)

(continued from previous page)

```
/* <397> [...]/zephyr/include/zephyr/sys/dlist.h:36 */
struct _dnode {
        union {
                struct _dnode *    head;                /*     0     4 */
                struct _dnode *    next;                /*     0     4 */
        };                                              /*     0     4 */
        union {
                struct _dnode *    tail;                /*     4     4 */
                struct _dnode *    prev;                /*     4     4 */
        };                                              /*     4     4 */

        /* size: 8, cachelines: 1, members: 2 */
        /* last cacheline: 8 bytes */
};
/* Used at: [...]/build/zephyr/kobject_hash.c */
/* <3b7> [...]/zephyr/include/zephyr/sys/dlist.h:41 */
union {
        struct _dnode *            tail;               /*     0     4 */
        struct _dnode *            prev;               /*     0     4 */
};
...
...
```

2.9 Flashing and Hardware Debugging

2.9.1 Flash & Debug Host Tools

This guide describes the software tools you can run on your host workstation to flash and debug Zephyr applications.

Zephyr's west tool has built-in support for all of these in its `flash`, `debug`, `debugserver`, and `attach` commands, provided your board hardware supports them and your Zephyr board directory's `board.cmake` file declares that support properly. See *Building, Flashing and Debugging* for more information on these commands.

SAM Boot Assistant (SAM-BA)

Atmel SAM Boot Assistant (Atmel SAM-BA) allows In-System Programming (ISP) from USB or UART host without any external programming interface. Zephyr allows users to develop and program boards with SAM-BA support using *west*. Zephyr supports devices with/without ROM bootloader and both extensions from Arduino and Adafruit. Full support was introduced in Zephyr SDK 0.12.0.

The typical command to flash the board is:

```
west flash [ -r bossac ] [ -p /dev/ttyX ]
```

Flash configuration for devices:

With ROM bootloader

These devices don't need any special configuration. After building your application, just run `west flash` to flash the board.

Without ROM bootloader

For these devices, the user should:

1. Define flash partitions required to accommodate the bootloader and application image; see *Flash map* for details.
2. Have board `.defconfig` file with the `CONFIG_USE_DT_CODE_PARTITION` Kconfig option set to `y` to instruct the build system to use these partitions for code relocation. This option can also be set in `prj.conf` or any other Kconfig fragment.
3. Build and flash the SAM-BA bootloader on the device.

With compatible SAM-BA bootloader

For these devices, the user should:

1. Define flash partitions required to accommodate the bootloader and application image; see *Flash map* for details.
2. Have board `.defconfig` file with the `CONFIG_BOOTLOADER_BOSSA` Kconfig option set to `y`. This will automatically select the `CONFIG_USE_DT_CODE_PARTITION` Kconfig option which instruct the build system to use these partitions for code relocation. The board `.defconfig` file should have `CONFIG_BOOTLOADER_BOSSA_ARDUINO` , `CONFIG_BOOTLOADER_BOSSA_ADAFRUIT_UF2` or the `CONFIG_BOOTLOADER_BOSSA_LEGACY` Kconfig option set to `y` to select the right compatible SAM-BA bootloader mode. These options can also be set in `prj.conf` or any other Kconfig fragment.
3. Build and flash the SAM-BA bootloader on the device.

Note

The `CONFIG_BOOTLOADER_BOSSA_LEGACY` Kconfig option should be used as last resource. Try configure first with Devices without ROM bootloader.

Typical flash layout and configuration For bootloaders that reside on flash, the devicetree partition layout is mandatory. For devices that have a ROM bootloader, they are mandatory when the application uses a storage or other non-application partition. In this special case, the boot partition should be omitted and code_partition should start from offset 0. It is necessary to define the partitions with sizes that avoid overlaps, always.

A typical flash layout for devices without a ROM bootloader is:

```
/ {
        chosen {
                zephyr,code-partition = &code_partition;
        };
};

&flash0 {
        partitions {
                compatible = "fixed-partitions";
                #address-cells = <1>;
                #size-cells = <1>;

                boot_partition: partition@0 {
                        label = "sam-ba";
                        reg = <0x00000000 0x2000>;
                        read-only;
                };

                code_partition: partition@2000 {
                        label = "code";
                        reg = <0x2000 0x3a000>;
                        read-only;
```

(continues on next page)

(continued from previous page)

```
                };

                /*
                * The final 16 KiB is reserved for the application.
                * Storage partition will be used by FCB/LittleFS/NVS
                * if enabled.
                */
                storage_partition: partition@3c000 {
                        label = "storage";
                        reg = <0x0003c000 0x00004000>;
                };
        };
};
```

A typical flash layout for devices with a ROM bootloader and storage partition is:

```
/ {
        chosen {
                zephyr,code-partition = &code_partition;
        };
};

&flash0 {
        partitions {
                compatible = "fixed-partitions";
                #address-cells = <1>;
                #size-cells = <1>;

                code_partition: partition@0 {
                        label = "code";
                        reg = <0x0 0xF0000>;
                        read-only;
                };

                /*
                * The final 64 KiB is reserved for the application.
                * Storage partition will be used by FCB/LittleFS/NVS
                * if enabled.
                */
                storage_partition: partition@F0000 {
                        label = "storage";
                        reg = <0x000F0000 0x00100000>;
                };
        };
};
```

Enabling SAM-BA runner In order to instruct Zephyr west tool to use the SAM-BA bootloader the `board.cmake` file must have `include(${ZEPHYR_BASE}/boards/common/bossac.board.cmake)` entry. Note that Zephyr tool accept more entries to define multiple runners. By default, the first one will be selected when using `west flash` command. The remaining options are available passing the runner option, for instance `west flash -r bossac`.

More implementation details can be found in the boards documentation. As a quick reference, see these three board documentation pages:

- sam4e_xpro (ROM bootloader)
- adafruit_feather_m0_basic_proto (Adafruit UF2 bootloader)
- arduino_nano_33_iot (Arduino bootloader)
- arduino_nano_33_ble (Arduino legacy bootloader)

Enabling BOSSAC on Windows Native [Experimental] Zephyr SDK´s bossac is currently supported on Linux and macOS only. Windows support can be achieved by using the bossac version from BOSSA official releases. After installing using default options, the `bossac.exe` must be added to Windows PATH. A specific bossac executable can be used by passing the `--bossac` option, as follows:

```
west flash -r bossac --bossac="C:\Program Files (x86)\BOSSA\bossac.exe" --bossac-port="COMx"
```

Note

WSL is not currently supported.

LinkServer Debug Host Tools

Linkserver is a utility for launching and managing GDB servers for NXP debug probes, which also provides a command-line target flash programming capabilities. Linkserver can be used with the NXP MCUXpresso for Visual Studio Code implementation, with custom debug configurations based on GNU tools or as part of a headless solution for continuous integration and test. LinkServer can be used with MCU-Link, LPC-Link2, LPC11U35-based and OpenSDA based standalone or on-board debug probes from NXP.

NXP recommends installing LinkServer by using NXP's MCUXpresso Installer. This method will also install the tools supporting the debug probes below, including NXP's MCU-Link and LPC-Scrypt tools.

LinkServer is compatible with the following debug probes:

- *LPC-LINK2 CMSIS DAP Onboard Debug Probe*
- *MCU-Link CMSIS-DAP Onboard Debug Probe*
- *OpenSDA DAPLink Onboard Debug Probe*

To use LinkServer with West commands, the install folder should be added to the *PATH environment variable*. The default installation path to add is:

Linux

```
/usr/local/LinkServer
```

Windows

```
c:\nxp\LinkServer_<version>
```

Supported west commands:

1. flash
2. debug
3. debugserver
4. attach

Notes:

1. Probes can be listed with LinkServer:

```
LinkServer probes
```

2. With multiple debug probes attached to the host, use the LinkServer west runner `--probe` option to pass the probe index.

```
west flash --runner=linkserver --probe=3
```

3. Device-specific settings can be overridden with the west runner for LinkServer with the option '–override'. May be used multiple times. The format is dictated by LinkServer, e.g.:

```
west flash --runner=linkserver --override /device/memory/5/flash-driver=MIMXRT500_SFDP_MXIC_
↪OSPI_S.cfx
```

4. LinkServer does not install an implicit breakpoint at the reset handler. If you would like to single step from the start of their application, you will need to add a breakpoint at `main` or the reset handler manually.

J-Link Debug Host Tools

Segger provides a suite of debug host tools for Linux, macOS, and Windows operating systems:

- J-Link GDB Server: GDB remote debugging
- J-Link Commander: Command-line control and flash programming
- RTT Viewer: RTT terminal input and output
- SystemView: Real-time event visualization and recording

These debug host tools are compatible with the following debug probes:

- *LPC-Link2 J-Link Onboard Debug Probe*
- *OpenSDA J-Link Onboard Debug Probe*
- *MCU-Link JLink Onboard Debug Probe*
- *J-Link External Debug Probe*
- *ST-LINK/V2-1 Onboard Debug Probe*

Check if your SoC is listed in J-Link Supported Devices.

Download and install the J-Link Software and Documentation Pack to get the J-Link GDB Server and Commander, and to install the associated USB device drivers. RTT Viewer and SystemView can be downloaded separately, but are not required.

Note that the J-Link GDB server does not yet support Zephyr RTOS-awareness.

OpenOCD Debug Host Tools

OpenOCD is a community open source project that provides GDB remote debugging and flash programming support for a wide range of SoCs. A fork that adds Zephyr RTOS-awareness is included in the Zephyr SDK; otherwise see Getting OpenOCD for options to download OpenOCD from official repositories.

These debug host tools are compatible with the following debug probes:

- *OpenSDA DAPLink Onboard Debug Probe*
- *J-Link External Debug Probe*
- *ST-LINK/V2-1 Onboard Debug Probe*

Check if your SoC is listed in OpenOCD Supported Devices.

Note

On Linux, openocd is available though the Zephyr SDK. Windows users should use the following steps to install openocd:

- Download openocd for Windows from here: OpenOCD Windows
- Copy bin and share dirs to `C:\Program Files\OpenOCD\`
- Add `C:\Program Files\OpenOCD\bin` to 'PATH' environment variable

pyOCD Debug Host Tools

pyOCD is an open source project from Arm that provides GDB remote debugging and flash programming support for Arm Cortex-M SoCs. It is distributed on PyPi and installed when you complete the *Get Zephyr and install Python dependencies* step in the Getting Started Guide. pyOCD includes support for Zephyr RTOS-awareness.

These debug host tools are compatible with the following debug probes:

- *LPC-LINK2 CMSIS DAP Onboard Debug Probe*
- *MCU-Link CMSIS-DAP Onboard Debug Probe*
- *OpenSDA DAPLink Onboard Debug Probe*
- *ST-LINK/V2-1 Onboard Debug Probe*

Check if your SoC is listed in pyOCD Supported Devices.

Lauterbach TRACE32 Debug Host Tools

Lauterbach TRACE32 is a product line of microprocessor development tools, debuggers and real-time tracer with support for JTAG, SWD, NEXUS or ETM over multiple core architectures, including Arm Cortex-A/-R/-M, RISC-V, Xtensa, etc. Zephyr allows users to develop and program boards with Lauterbach TRACE32 support using *west*.

The runner consists of a wrapper around TRACE32 software, and allows a Zephyr board to execute a custom start-up script (Practice Script) for the different commands supported, including the ability to pass extra arguments from CMake. Is up to the board using this runner to define the actions performed on each command.

Install Lauterbach TRACE32 Software Download Lauterbach TRACE32 software from the Lauterbach TRACE32 download website (registration required) and follow the installation steps described in Lauterbach TRACE32 Installation Guide.

Flashing and Debugging Set the *environment variable* `T32_DIR` to the TRACE32 system directory. Then execute `west flash` or `west debug` commands to flash or debug the Zephyr application as detailed in *Building, Flashing and Debugging*. The `debug` command launches TRACE32 GUI to allow debug the Zephyr application, while the `flash` command hides the GUI and perform all operations in the background.

By default, the `t32` runner will launch TRACE32 using the default configuration file named `config.t32` located in the TRACE32 system directory. To use a different configuration file, supply the argument `--config CONFIG` to the runner, for example:

```
west flash --config myconfig.t32
```

For more options, run `west flash --context -r t32` to print the usage.

Zephyr RTOS Awareness To enable Zephyr RTOS awareness follow the steps described in Lauterbach TRACE32 Zephyr OS Awareness Manual.

NXP S32 Debug Probe Host Tools

NXP S32 Debug Probe is designed to work in conjunction with NXP S32 Design Studio for S32 Platform.

Download (registration required) NXP S32 Design Studio for S32 Platform and follow the S32 Design Studio for S32 Platform Installation User Guide to get the necessary debug host tools and associated USB device drivers.

Note that Zephyr RTOS-awareness support for the NXP S32 GDB server depends on the target device. Consult the product release notes for more information.

Supported west commands:

1. debug
2. debugserver
3. attach

Basic usage Before starting, add NXP S32 Design Studio installation directory to the system *PATH environment variable*. Alternatively, it can be passed to the runner on each invocation via `--s32ds-path` as shown below:

Linux

```
west debug --s32ds-path=/opt/NXP/S32DS.3.5
```

Windows

```
west debug --s32ds-path=C:\NXP\S32DS.3.5
```

If multiple S32 debug probes are connected to the host via USB, the runner will ask the user to select one via command line prompt before continuing. The connection string for the probe can be also specified when invoking the runner via `--dev-id=<connection-string>`. Consult NXP S32 debug probe user manual for details on how to construct the connection string. For example, if using a probe with serial ID `00:04:9f:00:ca:fe`:

```
west debug --dev-id='s32dbg:00:04:9f:00:ca:fe'
```

It is possible to pass extra options to the debug host tools via `--tool-opt`. When executing `debug` or `attach` commands, the tool options will be passed to the GDB client only. When executing `debugserver`, the tool options will be passed to the GDB server. For example, to load a Zephyr application to SRAM and afterwards detach the debug session:

```
west debug --tool-opt='--batch'
```

2.9.2 Debug Probes

A *debug probe* is special hardware which allows you to control execution of a Zephyr application running on a separate board. Debug probes usually allow reading and writing registers and memory, and support breakpoint debugging of the Zephyr application on your host workstation using tools like GDB. They may also support other debug software and more advanced features such as *tracing program execution*. For details on the related host software supported by Zephyr, see *Flash & Debug Host Tools*.

Debug probes are usually connected to your host workstation via USB; they are sometimes also accessible via an IP network or other means. They usually connect to the device running Zephyr using the JTAG or SWD protocols. Debug probes are either separate hardware devices or circuitry integrated into the same board which runs Zephyr.

Many supported boards in Zephyr include a second microcontroller that serves as an onboard debug probe, usb-to-serial adapter, and sometimes a drag-and-drop flash programmer. This eliminates the need to purchase an external debug probe and provides a variety of debug host tool options.

Several hardware vendors have their own branded onboard debug probe implementations: NXP boards may use *OpenSDA*, *LPC-Link2*, or *MCU-Link*, probes depending on the microcontroller the debug probe firmware runs on. ST boards have the *ST-LINK probe*. Each onboard debug probe microcontroller can support one or more types of firmware that communicate with their respective debug host tools. For example, an OpenSDA microcontroller can be programmed with DAPLink firmware to communicate with pyOCD or OpenOCD debug host tools, or with J-Link firmware to communicate with J-Link debug host tools.

Debug Probes & Host Tools Compatibility Chart		Host Tools				
		J-Link Debug	**OpenOCD**	**pyOCD**	**NXP S32DS**	**NXP LinkServer**
Debug Probes	**J-Link External**	✓	✓			
	LPC-Link2 CMSIS-DAP					✓
	LPC-Link2 J-Link	✓				
	MCU-Link CMSIS-DAP					✓
	MCU-Link J-Link	✓				
	NXP S32 Debug Probe				✓	
	OpenSDA DAPLink		✓	✓		✓
	OpenSDA J-Link	✓				
	ST-LINK/V2-1	✓	✓	*some STM32 boards*		

Some supported boards in Zephyr do not include an onboard debug probe and therefore require an external debug probe. In addition, boards that do include an onboard debug probe often also have an SWD or JTAG header to enable the use of an external debug probe instead. One reason this may be useful is that the onboard debug probe may have limitations, such as lack of support for advanced debuggers or high-speed tracing. You may need to adjust jumpers to prevent the onboard debug probe from interfering with the external debug probe.

NXP Onboard Debug Probes

NXP boards may have one of several onboard debug probes. These probes include the *MCU-Link Onboard Debug Probe*, *LPC-LINK2 Onboard Debug Probe* and *OpenSDA Onboard Debug Probe*.

Each of these probes is implemented as a secondary microcontroller present on the evaluation board. The specific debug probe type present on a given board can be determined based on the debug microcontroller SOC:

- LPC55S69: *MCU-Link Onboard Debug Probe*
- LPC4322: *LPC-LINK2 Onboard Debug Probe*
- MK20: *OpenSDA Onboard Debug Probe*

For example, the frdm_k64f board has an MK20 debug microcontroller, so this board uses the *OpenSDA Onboard Debug Probe*.

MCU-Link Onboard Debug Probe

The MCU-Link onboard debug probe uses an LPC55S69 SOC. This probe supports the following firmwares:

- *MCU-Link CMSIS-DAP Onboard Debug Probe* (default firmware)
- *MCU-Link JLink Onboard Debug Probe*

This probe is programmed using the MCU-Link host tools, which are installed with the *LinkServer Debug Host Tools*. NXP recommends using NXP's MCUXpresso Installer to install the Linkserver tools.

MCU-Link CMSIS-DAP Onboard Debug Probe This is the default firmware installed on MCU-Link debug probes. The CMSIS-DAP debug probes allow debugging from any compatible toolchain, including IAR EWARM, Keil MDK, NXP's MCUXpresso IDE and MCUXpresso extension for VS Code. In addition to debug probe functionality, the MCU-Link probes may also provide:

1. SWO trace end point: this virtual device is used by MCUXpresso to retrieve SWO trace data. See the MCUXpresso IDE documentation for more information.
2. Virtual COM (VCOM) port / UART bridge connected to the target processor
3. USB to UART, SPI and/or I2C interfaces (depending on MCU-Link type/implementation)
4. Energy measurements of the target MCU

This debug probe is compatible with the following debug host tools:

- *LinkServer Debug Host Tools*

Once the MCU-Link host tools are installed, the following steps are required to program the CMSIS-DAP firmware:

1. Make sure the MCU-Link utility is present on your host machine. This can be done by installing *LinkServer Debug Host Tools*.
2. Put the MCU-Link microcontroller into DFU boot mode by attaching the DFU jumper then connecting to the USB debug port on the board. This jumper may also be referred to as the ISP jumper, and will be connected to `PIO0_5` on the LPC55S69.
3. Run the `program_CMSIS` script, found in the installed MCU-Link `scripts` folder.
4. Remove the DFU jumper and power cycle the board.

MCU-Link JLink Onboard Debug Probe This debug probe firmware provides a JLink compatible debug interface, as well as a USB-Serial adapter. It is compatible with the following debug host tools:

- *J-Link Debug Host Tools*

These probes do not have JLink firmware installed by default, and must be updated. Once the MCU-Link host tools are installed, the following steps are required to program the JLink firmware:

1. Make sure the MCU-Link utility is present on your host machine. This can be done by installing *LinkServer Debug Host Tools*.
2. Put the MCU-Link microcontroller into DFU boot mode by attaching the DFU jumper then connecting to the USB debug port on the board. This jumper may also be referred to as the ISP jumper, and will be connected to `PIO0_5` on the LPC55S69.
3. Run the `program_JLINK` script, found in the installed MCU-Link `scripts` folder.
4. Remove the DFU jumper and power cycle the board.

LPC-LINK2 Onboard Debug Probe

The LPC-LINK2 onboard debug probe uses an LPC4322 SOC. This probe supports the following firmwares:

- *LPC-LINK2 CMSIS DAP Onboard Debug Probe*
- *LPC-Link2 J-Link Onboard Debug Probe*
- *LPC-Link2 DAPLink Onboard Debug Probe* (default firmware)

This probe is programmed using the LPCScrypt host tools, which are installed with the *LinkServer Debug Host Tools*. NXP recommends using NXP's MCUXpresso Installer to install the Linkserver tools.

LPC-LINK2 CMSIS DAP Onboard Debug Probe The CMSIS-DAP debug probes allow debugging from any compatible toolchain, including IAR EWARM, Keil MDK, as well as NXP's MCUXpresso IDE and MCUXpresso extension for VS Code. As well as providing debug probe functionality, the LPC-Link2 probes also provide:

1. SWO trace end point: this virtual device is used by MCUXpresso to retrieve SWO trace data. See the MCUXpresso IDE documentation for more information.
2. Virtual COM (VCOM) port / UART bridge connected to the target processor
3. LPCSIO bridge that provides communication to I2C and SPI slave devices

This debug probe firmware is compatible with the following debug host tools:

- *LinkServer Debug Host Tools*

The probe may be updated to use CMSIS-DAP firmware with the following steps:

1. Make sure the LPCScrypt utility is present on your host machine. This can be done by installing *LinkServer Debug Host Tools*.
2. Put the LPC-Link2 microcontroller into DFU boot mode by attaching the DFU jumper, then connecting to the USB debug port on the board. This jumper is connected to `P2_6` on the LPC4322 SOC.
3. Run the `program_CMSIS` script, found in the installed LPCScrypt `scripts` folder.
4. Remove the DFU jumper and power cycle the board.

> **Note**
>
> On some boards, the J-Link probe firmware will no longer power the board via the USB debug port. On these boards, an alternative method of powering the board must be used when this firmware is programmed.

LPC-Link2 J-Link Onboard Debug Probe This debug probe firmware provides a JLink compatible debug interface, as well as a USB-Serial adapter. It is compatible with the following debug host tools:

- *J-Link Debug Host Tools*

The probe may be updated to use the J-Link firmware with the following steps:

> **Note**
>
> Verify the firmware supports your board by visiting Firmware for LPCXpresso

1. Make sure the LPCScrypt utility is present on your host machine. This can be done by installing *LinkServer Debug Host Tools*.
2. Put the LPC-Link2 microcontroller into DFU boot mode by attaching the DFU jumper, then connecting to the USB debug port on the board. This jumper is connected to `P2_6` on the LPC4322 SOC.
3. Run the `program_JLINK` script, found in the installed LPCScrypt `scripts` folder.
4. Remove the DFU jumper and power cycle the board.

LPC-Link2 DAPLink Onboard Debug Probe The LPC-Link2 DAPLink firmware is the default firmware shipped on LPC-Link2 based boards, but is not the recommended firmware. Users should update to the *LPC-LINK2 CMSIS DAP Onboard Debug Probe* firmware following the instructions provided above. For details on programming the DAPLink firmware, see NXP AN13206.

OpenSDA Onboard Debug Probe

The OpenSDA onboard debug probe is based on the NXP MK20 SOC. It features drag and drop programming supports, and supports the following debug firmwares:

- *OpenSDA DAPLink Onboard Debug Probe* (default firmware)
- *OpenSDA J-Link Onboard Debug Probe*

OpenSDA DAPLink Onboard Debug Probe This debug probe firmware is compatible with the following debug host tools:

- *pyOCD Debug Host Tools*
- *OpenOCD Debug Host Tools*
- *LinkServer Debug Host Tools*

This probe is realized by programming the OpenSDA microcontroller with DAPLink OpenSDA firmware. NXP provides OpenSDA DAPLink Board-Specific Firmwares.

Install the debug host tools before you program the firmware.

As with all OpenSDA debug probes, the steps for programming the firmware are:

1. Put the OpenSDA microcontroller into bootloader mode by holding the reset button while you power on the board. Note that "bootloader mode" in this context applies to the OpenSDA microcontroller itself, not the target microcontroller of your Zephyr application.
2. After you power on the board, release the reset button. A USB mass storage device called **BOOTLOADER** or **MAINTENANCE** will enumerate. If the enumerated device is named **BOOTLOADER**, please first update the bootloader to the latest revision by following the instructions for a DAPLink Bootloader Update.
3. Copy the OpenSDA firmware binary to the USB mass storage device.
4. Power cycle the board, this time without holding the reset button. You should see three USB devices enumerate: a CDC device (serial port), a HID device (debug port), and a mass storage device (drag-and-drop flash programming).

OpenSDA J-Link Onboard Debug Probe This debug probe is compatible with the following debug host tools:

- *J-Link Debug Host Tools*

This probe is realized by programming the OpenSDA microcontroller with J-Link OpenSDA firmware. Segger provides OpenSDA J-Link Generic Firmwares and OpenSDA J-Link Board-Specific Firmwares, where the latter is generally recommended when available. Board-specific firmwares are required for i.MX RT boards to support their external flash memories, whereas generic firmwares are compatible with all Kinetis boards.

Install the debug host tools before you program the firmware.

As with all OpenSDA debug probes, the steps for programming the firmware are:

1. Put the OpenSDA microcontroller into bootloader mode by holding the reset button while you plug a USB into the board's USB debug port. Note that "bootloader mode" in this context applies to the OpenSDA microcontroller itself, not the target microcontroller of your Zephyr application.
2. After you power on the board, release the reset button. A USB mass storage device called **BOOTLOADER** or **MAINTENANCE** will enumerate. If the enumerated device is named **BOOTLOADER**, please first update the bootloader to the latest revision by following the instructions for a DAPLink Bootloader Update.
3. Copy the OpenSDA firmware binary to the USB mass storage device.
4. Power cycle the board, this time without holding the reset button. You should see two USB devices enumerate: a CDC device (serial port) and a vendor-specific device (debug port).

J-Link External Debug Probe

Segger J-Link is a family of external debug probes, including J-Link EDU, J-Link PLUS, J-Link ULTRA+, and J-Link PRO, that support a large number of devices from different hardware architectures and vendors.

This debug probe is compatible with the following debug host tools:

- *J-Link Debug Host Tools*
- *OpenOCD Debug Host Tools*

Install the debug host tools before you program the firmware.

ST-LINK/V2-1 Onboard Debug Probe

ST-LINK/V2-1 is a serial and debug adapter built into all Nucleo and Discovery boards. It provides a bridge between your computer (or other USB host) and the embedded target processor, which

can be used for debugging, flash programming, and serial communication, all over a simple USB cable.

It is compatible with the following host debug tools:

- *OpenOCD Debug Host Tools*
- *J-Link Debug Host Tools*

For some STM32 based boards, it is also compatible with:

- *pyOCD Debug Host Tools*

While it works out of the box with OpenOCD, it requires some flashing to work with J-Link. To do this, SEGGER offers a firmware upgrading the ST-LINK/V2-1 on board on the Nucleo and Discovery boards. This firmware makes the ST-LINK/V2-1 compatible with J-LinkOB, allowing users to take advantage of most J-Link features like the ultra fast flash download and debugging speed or the free-to-use GDBServer.

More information about upgrading ST-LINK/V2-1 to JLink or restore ST-Link/V2-1 firmware please visit: Segger over ST-Link

Flash and debug with ST-Link Using OpenOCD

OpenOCD is available by default on ST-Link and configured as the default flash and debug tool. Flash and debug can be done as follows:

```
# From the root of the zephyr repository
west build -b None samples/hello_world
west flash
```

```
# From the root of the zephyr repository
west build -b None samples/hello_world
west debug
```

Using Segger J-Link

Once STLink is flashed with SEGGER FW and J-Link GDB server is installed on your host computer, you can flash and debug as follows:

Use CMake with `-DBOARD_FLASH_RUNNER=jlink` to change the default OpenOCD runner to J-Link. Alternatively, you might add the following line to your application `CMakeList.txt` file.

```
set(BOARD_FLASH_RUNNER jlink)
```

If you use West (Zephyr's meta-tool) you can modify the default runner using the `--runner` (or `-r`) option.

```
west flash --runner jlink
```

To attach a debugger to your board and open up a debug console with `jlink`.

```
west debug --runner jlink
```

For more information about West and available options, see *West (Zephyr's meta-tool)*.

If you configured your Zephyr application to use Segger RTT console instead, open telnet:

```
$ telnet localhost 19021
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
SEGGER J-Link V6.30f - Real time terminal output
```

(continues on next page)

(continued from previous page)

```
J-Link STLink V21 compiled Jun 26 2017 10:35:16 V1.0, SN=773895351
Process: JLinkGDBServerCLExe
Zephyr Shell, Zephyr version: 1.12.99
Type 'help' for a list of available commands
shell>
```

If you get no RTT output you might need to disable other consoles which conflict with the RTT one if they are enabled by default in the particular sample or application you are running, such as disable UART_CONSOLE in menuconfig

Updating or restoring ST-Link firmware ST-Link firmware can be updated using STM32CubeProgrammer Tool. It is usually useful when facing flashing issues, for instance when using twister's device-testing option.

Once installed, you can update attached board ST-Link firmware with the following command

```
s java -jar ~/STMicroelectronics/STM32Cube/STM32CubeProgrammer/Drivers/
↪FirmwareUpgrade/STLinkUpgrade.jar -sn <board_uid>
```

Where board_uid can be obtained using twister's generate-hardware-map option. For more information about twister and available options, see *Test Runner (Twister)*.

NXP S32 Debug Probe

NXP S32 Debug Probe enables NXP S32 target system debugging via a standard debug port while connected to a developer's workstation via USB or remotely via Ethernet.

NXP S32 Debug Probe is designed to work in conjunction with NXP S32 Design Studio (S32DS) and NXP Automotive microcontrollers and processors. Install the debug host tools as in indicated in *NXP S32 Debug Probe Host Tools* before you program the firmware.

2.10 Modules (External projects)

Zephyr relies on the source code of several externally maintained projects in order to avoid reinventing the wheel and to reuse as much well-established, mature code as possible when it makes sense. In the context of Zephyr's build system those are called *modules*. These modules must be integrated with the Zephyr build system, as described in more detail in other sections on this page.

To be classified as a candidate for being included in the default list of modules, an external project is required to have its own life-cycle outside the Zephyr Project, that is, reside in its own repository, and have its own contribution and maintenance workflow and release process. Zephyr modules should not contain code that is written exclusively for Zephyr. Instead, such code should be contributed to the main zephyr tree.

Modules to be included in the default manifest of the Zephyr project need to provide functionality or features endorsed and approved by the project Technical Steering Committee and should comply with the *module licensing requirements* and *contribution guidelines*. They should also have a Zephyr developer that is committed to maintain the module codebase.

Zephyr depends on several categories of modules, including but not limited to:

- Debugger integration
- Silicon vendor Hardware Abstraction Layers (HALs)
- Cryptography libraries

- File Systems
- Inter-Process Communication (IPC) libraries

Additionally, in some cases modules (particularly vendor HALs) can contain references to optional *binary blobs*.

This page summarizes a list of policies and best practices which aim at better organizing the workflow in Zephyr modules.

2.10.1 Modules vs west projects

Zephyr modules, described in this page, are not the same as *west projects*. In fact, modules *do not require west* at all. However, when using modules *with west*, then the build system uses west in order to find modules.

In summary:

Modules are repositories that contain a `zephyr/module.yml` file, so that the Zephyr build system can pull in the source code from the repository. *West projects* are entries in the *projects:* section in the `west.yml` manifest file. West projects are often also modules, but not always. There are west projects that are not included in the final firmware image (eg. tools) and thus do not need to be modules. Modules are found by the Zephyr build system either via *west itself*, or via the *ZEPHYR_MODULES CMake variable*.

The contents of this page only apply to modules, and not to west projects in general (unless they are a module themselves).

2.10.2 Module Repositories

- All modules included in the default manifest shall be hosted in repositories under the zephyrproject-rtos GitHub organization.
- The module repository codebase shall include a *module.yml* file in a `zephyr/` folder at the root of the repository.
- Module repository names should follow the convention of using lowercase letters and dashes instead of underscores. This rule will apply to all new module repositories, except for repositories that are directly tracking external projects (hosted in Git repositories); such modules may be named as their external project counterparts.

> **Note**
>
> Existing module repositories that do not conform to the above convention do not need to be renamed to comply with the above convention.

- Module repositories names should be explicitly set in the `zephyr/module.yml` file.
- Modules should use “zephyr” as the default name for the repository main branch. Branches for specific purposes, for example, a module branch for an LTS Zephyr version, shall have names starting with the ‘zephyr_’ prefix.
- If the module has an external (upstream) project repository, the module repository should preserve the upstream repository folder structure.

> **Note**
>
> It is not required in module repositories to maintain a 'master' branch mirroring the master branch of the external repository. It is not recommended as this may generate confusion around the module's main branch, which should be 'zephyr'.

- Modules should expose all provided header files with an include pathname beginning with the module-name. (E.g., mcuboot should expose its `bootutil/bootutil.h` as "mcuboot/bootutil/bootutil.h".)

Synchronizing with upstream

It is preferred to synchronize a module repository with the latest stable release of the corresponding external project. It is permitted, however, to update a Zephyr module repository with the latest development branch tip, if this is required to get important updates in the module codebase. When synchronizing a module with upstream it is mandatory to document the rationale for performing the particular update.

Requirements for allowed practices Changes to the main branch of a module repository, including synchronization with upstream code base, may only be applied via pull requests. These pull requests shall be *verifiable* by Zephyr CI and *mergeable* (e.g. with the *Rebase and merge*, or *Create a merge commit* option using Github UI). This ensures that the incoming changes are always **reviewable**, and the *downstream* module repository history is incremental (that is, existing commits, tags, etc. are always preserved). This policy also allows to run Zephyr CI, git lint, identity, and license checks directly on the set of changes that are to be brought into the module repository.

> **Note**
>
> Force-pushing to a module's main branch is not allowed.

Allowed practices The following practices conform to the above requirements and should be followed in all modules repositories. It is up to the module code owner to select the preferred synchronization practice, however, it is required that the selected practice is consistently followed in the respective module repository.

Updating modules with a diff from upstream: Upstream changes brought as a single *snapshot* commit (manual diff) in a pull request against the module's main branch, which may be merged using the *Rebase & merge* operation. This approach is simple and should be applicable to all modules with the downside of suppressing the upstream history in the module repository.

> **Note**
>
> The above practice is the only allowed practice in modules where the external project is not hosted in an upstream Git repository.

The commit message is expected to identify the upstream project URL, the version to which the module is updated (upstream version, tag, commit SHA, if applicable, etc.), and the reason for the doing the update.

Updating modules by merging the upstream branch: Upstream changes brought in by performing a Git merge of the intended upstream branch (e.g. main branch, latest release branch,

etc.) submitting the result in pull request against the module main branch, and merging the pull request using the *Create a merge commit* operation. This approach is applicable to modules with an upstream project Git repository. The main advantages of this approach is that the upstream repository history (that is, the original commit SHAs) is preserved in the module repository. The downside of this approach is that two additional merge commits are generated in the downstream main branch.

2.10.3 Contributing to Zephyr modules

Individual Roles & Responsibilities

To facilitate management of Zephyr module repositories, the following individual roles are defined.

Administrator: Each Zephyr module shall have an administrator who is responsible for managing access to the module repository, for example, for adding individuals as Collaborators in the repository at the request of the module owner. Module administrators are members of the Administrators team, that is a group of project members with admin rights to module GitHub repositories.

Module owner: Each module shall have a module code owner. Module owners will have the overall responsibility of the contents of a Zephyr module repository. In particular, a module owner will:

- coordinate code reviewing in the module repository
- be the default assignee in pull-requests against the repository's main branch
- request additional collaborators to be added to the repository, as they see fit
- regularly synchronize the module repository with its upstream counterpart following the policies described in *Synchronizing with upstream*
- be aware of security vulnerability issues in the external project and update the module repository to include security fixes, as soon as the fixes are available in the upstream code base
- list any known security vulnerability issues, present in the module codebase, in Zephyr release notes.

> **Note**
>
> Module owners are not required to be Zephyr *Maintainers*.

Merger: The Zephyr Release Engineering team has the right and the responsibility to merge approved pull requests in the main branch of a module repository.

Maintaining the module codebase

Updates in the zephyr main tree, for example, in public Zephyr APIs, may require patching a module's codebase. The responsibility for keeping the module codebase up to date is shared between the **contributor** of such updates in Zephyr and the module **owner**. In particular:

- the contributor of the original changes in Zephyr is required to submit the corresponding changes that are required in module repositories, to ensure that Zephyr CI on the pull request with the original changes, as well as the module integration testing are successful.

- the module owner has the overall responsibility for synchronizing and testing the module codebase with the zephyr main tree. This includes occasional advanced testing of the module's codebase in addition to the testing performed by Zephyr's CI. The module owner is required to fix issues in the module's codebase that have not been caught by Zephyr pull request CI runs.

Contributing changes to modules

Submitting and merging changes directly to a module's codebase, that is, before they have been merged in the corresponding external project repository, should be limited to:

- changes required due to updates in the zephyr main tree
- urgent changes that should not wait to be merged in the external project first, such as fixes to security vulnerabilities.

Non-trivial changes to a module's codebase, including changes in the module design or functionality should be discouraged, if the module has an upstream project repository. In that case, such changes shall be submitted to the upstream project, directly.

Submitting changes to modules describes in detail the process of contributing changes to module repositories.

Contribution guidelines Contributing to Zephyr modules shall follow the generic project *Contribution guidelines*.

Pull Requests: may be merged with minimum of 2 approvals, including an approval by the PR assignee. In addition to this, pull requests in module repositories may only be merged if the introduced changes are verified with Zephyr CI tools, as described in more detail in other sections on this page.

The merging of pull requests in the main branch of a module repository must be coupled with the corresponding manifest file update in the zephyr main tree.

Issue Reporting: GitHub issues are intentionally disabled in module repositories, in favor of a centralized policy for issue reporting. Tickets concerning, for example, bugs or enhancements in modules shall be opened in the main zephyr repository. Issues should be appropriately labeled using GitHub labels corresponding to each module, where applicable.

> **Note**
>
> It is allowed to file bug reports for zephyr modules to track the corresponding upstream project bugs in Zephyr. These bug reports shall not affect the *Release Quality Criteria*.

2.10.4 Licensing requirements and policies

All source files in a module's codebase shall include a license header, unless the module repository has **main license file** that covers source files that do not include license headers.

Main license files shall be added in the module's codebase by Zephyr developers, only if they exist as part of the external project, and they contain a permissive OSI-compliant license. Main license files should preferably contain the full license text instead of including an SPDX license identifier. If multiple main license files are present it shall be made clear which license applies to each source file in a module's codebase.

Individual license headers in module source files supersede the main license.

Any new content to be added in a module repository will require to have license coverage.

> **Note**
>
> Zephyr recommends conveying module licensing via individual license headers and main license files. This not a hard requirement; should an external project have its own practice of conveying how licensing applies in the module's codebase (for example, by having a single or multiple main license files), this practice may be accepted by and be referred to in the Zephyr module, as long as licensing requirements, for example OSI compliance, are satisfied.

License policies

When creating a module repository a developer shall:

- import the main license files, if they exist in the external project, and
- document (for example in the module README or .yml file) the default license that covers the module's codebase.

License checks License checks (via CI tools) shall be enabled on every pull request that adds new content in module repositories.

2.10.5 Documentation requirements

All Zephyr module repositories shall include an .rst file documenting:

- the scope and the purpose of the module
- how the module integrates with Zephyr
- the owner of the module repository
- synchronization information with the external project (commit, SHA, version etc.)
- licensing information as described in *Licensing requirements and policies*.

The file shall be required for the inclusion of the module and the contained information should be kept up to date.

2.10.6 Testing requirements

All Zephyr modules should provide some level of **integration** testing, ensuring that the integration with Zephyr works correctly. Integration tests:

- may be in the form of a minimal set of samples and tests that reside in the zephyr main tree
- should verify basic usage of the module (configuration, functional APIs, etc.) that is integrated with Zephyr.
- shall be built and executed (for example in QEMU) as part of twister runs in pull requests that introduce changes in module repositories.

> **Note**
>
> New modules, that are candidates for being included in the Zephyr default manifest, shall provide some level of integration testing.

> **Note**
>
> Vendor HALs are implicitly tested via Zephyr tests built or executed on target platforms, so they do not need to provide integration tests.

The purpose of integration testing is not to provide functional verification of the module; this should be part of the testing framework of the external project.

Certain external projects provide test suites that reside in the upstream testing infrastructure but are written explicitly for Zephyr. These tests may (but are not required to) be part of the Zephyr test framework.

2.10.7 Deprecating and removing modules

Modules may be deprecated for reasons including, but not limited to:

- Lack of maintainership in the module
- Licensing changes in the external project
- Codebase becoming obsolete

The module information shall indicate whether a module is deprecated and the build system shall issue a warning when trying to build Zephyr using a deprecated module.

Deprecated modules may be removed from the Zephyr default manifest after 2 Zephyr releases.

> **Note**
>
> Repositories of removed modules shall remain accessible via their original URL, as they are required by older Zephyr versions.

2.10.8 Integrate modules in Zephyr build system

The build system variable **ZEPHYR_MODULES** is a CMake list of absolute paths to the directories containing Zephyr modules. These modules contain `CMakeLists.txt` and `Kconfig` files describing how to build and configure them, respectively. Module `CMakeLists.txt` files are added to the build using CMake's add_subdirectory() command, and the `Kconfig` files are included in the build's Kconfig menu tree.

If you have *west* installed, you don't need to worry about how this variable is defined unless you are adding a new module. The build system knows how to use west to set **ZEPHYR_MODULES**. You can add additional modules to this list by setting the **EXTRA_ZEPHYR_MODULES** CMake variable or by adding a **EXTRA_ZEPHYR_MODULES** line to `.zephyrrc` (See the section on *Environment Variables* for more details). This can be useful if you want to keep the list of modules found with west and also add your own.

> **Note**
>
> If the module `FOO` is provided by *west* but also given with `-DEXTRA_ZEPHYR_MODULES=/<path>/foo` then the module given by the command line variable **EXTRA_ZEPHYR_MODULES** will take precedence. This allows you to use a custom version of `FOO` when building and still use other Zephyr modules provided by *west*. This can for example be useful for special test purposes.

If you want to permanently add modules to the zephyr workspace and you are using zephyr as your manifest repository, you can also add a west manifest file into the submanifests directory. See submanifests/README.txt for more details.

See *Basics* for more on west workspaces.

Finally, you can also specify the list of modules yourself in various ways, or not use modules at all if your application doesn't need them.

2.10.9 Module yaml file description

A module can be described using a file named `zephyr/module.yml`. The format of `zephyr/module.yml` is described in the following:

Module name

Each Zephyr module is given a name by which it can be referred to in the build system.

The name should be specified in the `zephyr/module.yml` file. This will ensure the module name is not changeable through user-defined directory names or `west` manifest files:

```
name: <name>
```

In CMake the location of the Zephyr module can then be referred to using the CMake variable `ZEPHYR_<MODULE_NAME>_MODULE_DIR` and the variable `ZEPHYR_<MODULE_NAME>_CMAKE_DIR` holds the location of the directory containing the module's `CMakeLists.txt` file.

> **Note**
>
> When used for CMake and Kconfig variables, all letters in module names are converted to uppercase and all non-alphanumeric characters are converted to underscores (_). As example, the module `foo-bar` must be referred to as `ZEPHYR_FOO_BAR_MODULE_DIR` in CMake and Kconfig.

Here is an example for the Zephyr module `foo`:

```
name: foo
```

> **Note**
>
> If the `name` field is not specified then the Zephyr module name will be set to the name of the module folder. As example, the Zephyr module located in `<workspace>/modules/bar` will use `bar` as its module name if nothing is specified in `zephyr/module.yml`.

Module integration files (in-module)

Inclusion of build files, `CMakeLists.txt` and `Kconfig`, can be described as:

```
build:
  cmake: <cmake-directory>
  kconfig: <directory>/Kconfig
```

The `cmake: <cmake-directory>` part specifies that `<cmake-directory>` contains the `CMakeLists.txt` to use. The `kconfig: <directory>/Kconfig` part specifies the Kconfig file to use. Neither is required: `cmake` defaults to `zephyr`, and `kconfig` defaults to `zephyr/Kconfig`.

Here is an example `module.yml` file referring to `CMakeLists.txt` and `Kconfig` files in the root directory of the module:

```
build:
  cmake: .
  kconfig: Kconfig
```

Sysbuild integration

Sysbuild is the Zephyr build system that allows for building multiple images as part of a single application, the sysbuild build process can be extended externally with modules as needed, for example to add custom build steps or add additional targets to a build. Inclusion of sysbuild-specific build files, `CMakeLists.txt` and `Kconfig`, can be described as:

```
build:
  sysbuild-cmake: <cmake-directory>
  sysbuild-kconfig: <directory>/Kconfig
```

The `sysbuild-cmake: <cmake-directory>` part specifies that `<cmake-directory>` contains the `CMakeLists.txt` to use. The `sysbuild-kconfig: <directory>/Kconfig` part specifies the Kconfig file to use.

Here is an example `module.yml` file referring to `CMakeLists.txt` and `Kconfig` files in the *sysbuild* directory of the module:

```
build:
  sysbuild-cmake: sysbuild
  sysbuild-kconfig: sysbuild/Kconfig
```

The module description file `zephyr/module.yml` can also be used to specify that the build files, `CMakeLists.txt` and `Kconfig`, are located in a *Module integration files (external)*.

Build files located in a `MODULE_EXT_ROOT` can be described as:

```
build:
  sysbuild-cmake-ext: True
  sysbuild-kconfig-ext: True
```

This allows control of the build inclusion to be described externally to the Zephyr module.

Vulnerability monitoring

The module description file `zephyr/module.yml` can be used to improve vulnerability monitoring.

If your module needs to track vulnerabilities using an external reference (e.g your module is forked from another repository), you can use the `security` section. It contains the field `external-references` that contains a list of references that needs to be monitored for your module. The supported formats are:

- CPE (Common Platform Enumeration)
- PURL (Package URL)

```
security:
  external-references:
    - <module-related-cpe>
    - <an-other-module-related-cpe>
    - <module-related-purl>
```

A real life example for *mbedTLS* module could look like this:

```
security:
  external-references:
    - cpe:2.3:a:arm:mbed_tls:3.5.2:*:*:*:*:*:*:*
    - pkg:github/Mbed-TLS/mbedtls@V3.5.2
```

Note

CPE field must follow the CPE 2.3 schema provided by NVD. PURL field must follow the PURL specification provided by Github.

Build system integration

When a module has a `module.yml` file, it will automatically be included into the Zephyr build system. The path to the module is then accessible through Kconfig and CMake variables.

Zephyr modules In both Kconfig and CMake, the variable `ZEPHYR_<MODULE_NAME>_MODULE_DIR` contains the absolute path to the module.

In CMake, `ZEPHYR_<MODULE_NAME>_CMAKE_DIR` contains the absolute path to the directory containing the `CMakeLists.txt` file that is included into CMake build system. This variable's value is empty if the module.yml file does not specify a CMakeLists.txt.

To read these variables for a Zephyr module named `foo`:

- In CMake: use `${ZEPHYR_FOO_MODULE_DIR}` for the module's top level directory, and `${ZEPHYR_FOO_CMAKE_DIR}` for the directory containing its `CMakeLists.txt`
- In Kconfig: use `$(ZEPHYR_FOO_MODULE_DIR)` for the module's top level directory

Notice how a lowercase module name `foo` is capitalized to `FOO` in both CMake and Kconfig.

These variables can also be used to test whether a given module exists. For example, to verify that `foo` is the name of a Zephyr module:

```
if(ZEPHYR_FOO_MODULE_DIR)
  # Do something if FOO exists.
endif()
```

In Kconfig, the variable may be used to find additional files to include. For example, to include the file `some/Kconfig` in module `foo`:

```
source "$(ZEPHYR_FOO_MODULE_DIR)/some/Kconfig"
```

During CMake processing of each Zephyr module, the following variables are also available:

- the current module's name: `${ZEPHYR_CURRENT_MODULE_NAME}`
- the current module's top level directory: `${ZEPHYR_CURRENT_MODULE_DIR}`
- the current module's `CMakeLists.txt` directory: `${ZEPHYR_CURRENT_CMAKE_DIR}`

This removes the need for a Zephyr module to know its own name during CMake processing. The module can source additional CMake files using these `CURRENT` variables. For example:

```
include(${ZEPHYR_CURRENT_MODULE_DIR}/cmake/code.cmake)
```

It is possible to append values to a Zephyr CMake list variable from the module's first CMakeLists.txt file. To do so, append the value to the list and then set the list in the PARENT_SCOPE of the CMakeLists.txt file. For example, to append `bar` to the `FOO_LIST` variable in the Zephyr CMakeLists.txt scope:

```
list(APPEND FOO_LIST bar)
set(FOO_LIST ${FOO_LIST} PARENT_SCOPE)
```

An example of a Zephyr list where this is useful is when adding additional directories to the SYSCALL_INCLUDE_DIRS list.

Sysbuild modules In both Kconfig and CMake, the variable SYSBUILD_CURRENT_MODULE_DIR contains the absolute path to the sysbuild module. In CMake, SYSBUILD_CURRENT_CMAKE_DIR contains the absolute path to the directory containing the CMakeLists.txt file that is included into CMake build system. This variable's value is empty if the module.yml file does not specify a CMakeLists.txt.

To read these variables for a sysbuild module:

- In CMake: use ${SYSBUILD_CURRENT_MODULE_DIR} for the module's top level directory, and ${SYSBUILD_CURRENT_CMAKE_DIR} for the directory containing its CMakeLists.txt
- In Kconfig: use $(SYSBUILD_CURRENT_MODULE_DIR) for the module's top level directory

In Kconfig, the variable may be used to find additional files to include. For example, to include the file some/Kconfig:

```
source "$(SYSBUILD_CURRENT_MODULE_DIR)/some/Kconfig"
```

The module can source additional CMake files using these variables. For example:

```
include(${SYSBUILD_CURRENT_MODULE_DIR}/cmake/code.cmake)
```

It is possible to append values to a Zephyr CMake list variable from the module's first CMakeLists.txt file. To do so, append the value to the list and then set the list in the PARENT_SCOPE of the CMakeLists.txt file. For example, to append bar to the FOO_LIST variable in the Zephyr CMakeLists.txt scope:

```
list(APPEND FOO_LIST bar)
set(FOO_LIST ${FOO_LIST} PARENT_SCOPE)
```

Sysbuild modules hooks Sysbuild provides an infrastructure which allows a sysbuild module to define a function which will be invoked by sysbuild at a pre-defined point in the CMake flow.

Functions invoked by sysbuild:

- <module-name>_pre_cmake(IMAGES <images>): This function is called for each sysbuild module before CMake configure is invoked for all images.
- <module-name>_post_cmake(IMAGES <images>): This function is called for each sysbuild module after CMake configure has completed for all images.
- <module-name>_pre_domains(IMAGES <images>): This function is called for each sysbuild module before domains yaml is created by sysbuild.
- <module-name>_post_domains(IMAGES <images>): This function is called for each sysbuild module after domains yaml has been created by sysbuild.

arguments passed from sysbuild to the function defined by a module:

- <images> is the list of Zephyr images that will be created by the build system.

If a module foo want to provide a post CMake configure function, then the module's sysbuild CMakeLists.txt file must define function foo_post_cmake().

To facilitate naming of functions, the module name is provided by sysbuild CMake through the SYSBUILD_CURRENT_MODULE_NAME CMake variable when loading the module's sysbuild CMakeLists.txt file.

Example of how the foo sysbuild module can define foo_post_cmake():

```
function(${SYSBUILD_CURRENT_MODULE_NAME}_post_cmake)
  cmake_parse_arguments(POST_CMAKE "" "" "IMAGES" ${ARGN})

  message("Invoking ${CMAKE_CURRENT_FUNCTION}. Images: ${POST_CMAKE_IMAGES}")
endfunction()
```

Zephyr module dependencies

A Zephyr module may be dependent on other Zephyr modules to be present in order to function correctly. Or it might be that a given Zephyr module must be processed after another Zephyr module, due to dependencies of certain CMake targets.

Such a dependency can be described using the depends field.

```
build:
  depends:
    - <module>
```

Here is an example for the Zephyr module foo that is dependent on the Zephyr module bar to be present in the build system:

```
name: foo
build:
  depends:
    - bar
```

This example will ensure that bar is present when foo is included into the build system, and it will also ensure that bar is processed before foo.

Module integration files (external)

Module integration files can be located externally to the Zephyr module itself. The MODULE_EXT_ROOT variable holds a list of roots containing integration files located externally to Zephyr modules.

Module integration files in Zephyr The Zephyr repository contain CMakeLists.txt and Kconfig build files for certain known Zephyr modules.

Those files are located under

```
<ZEPHYR_BASE>
└── modules
    └── <module_name>
        ├── CMakeLists.txt
        └── Kconfig
```

Module integration files in a custom location You can create a similar MODULE_EXT_ROOT for additional modules, and make those modules known to Zephyr build system.

Create a MODULE_EXT_ROOT with the following structure

```
<MODULE_EXT_ROOT>
└── modules
    ├── modules.cmake
    └── <module_name>
```

(continues on next page)

(continued from previous page)

```
        ├── CMakeLists.txt
        └── Kconfig
```

and then build your application by specifying `-DMODULE_EXT_ROOT` parameter to the CMake build system. The `MODULE_EXT_ROOT` accepts a CMake list of roots as argument.

A Zephyr module can automatically be added to the `MODULE_EXT_ROOT` list using the module description file `zephyr/module.yml`, see *Build settings*.

Note

`ZEPHYR_BASE` is always added as a `MODULE_EXT_ROOT` with the lowest priority. This allows you to overrule any integration files under `<ZEPHYR_BASE>/modules/<module_name>` with your own implementation your own `MODULE_EXT_ROOT`.

The `modules.cmake` file must contain the logic that specifies the integration files for Zephyr modules via specifically named CMake variables.

To include a module's CMake file, set the variable `ZEPHYR_<MODULE_NAME>_CMAKE_DIR` to the path containing the CMake file.

To include a module's Kconfig file, set the variable `ZEPHYR_<MODULE_NAME>_KCONFIG` to the path to the Kconfig file.

The following is an example on how to add support the `FOO` module.

Create the following structure

```
<MODULE_EXT_ROOT>
└── modules
    ├── modules.cmake
    └── foo
        ├── CMakeLists.txt
        └── Kconfig
```

and inside the `modules.cmake` file, add the following content

```
set(ZEPHYR_FOO_CMAKE_DIR ${CMAKE_CURRENT_LIST_DIR}/foo)
set(ZEPHYR_FOO_KCONFIG   ${CMAKE_CURRENT_LIST_DIR}/foo/Kconfig)
```

Module integration files (zephyr/module.yml) The module description file `zephyr/module.yml` can be used to specify that the build files, `CMakeLists.txt` and `Kconfig`, are located in a *Module integration files (external)*.

Build files located in a `MODULE_EXT_ROOT` can be described as:

```
build:
  cmake-ext: True
  kconfig-ext: True
```

This allows control of the build inclusion to be described externally to the Zephyr module.

The Zephyr repository itself is always added as a Zephyr module ext root.

Build settings

It is possible to specify additional build settings that must be used when including the module into the build system.

All `root` settings are relative to the root of the module.

Build settings supported in the `module.yml` file are:

- `board_root`: Contains additional boards that are available to the build system. Additional boards must be located in a `<board_root>/boards` folder.
- `dts_root`: Contains additional dts files related to the architecture/soc families. Additional dts files must be located in a `<dts_root>/dts` folder.
- `snippet_root`: Contains additional snippets that are available for use. These snippets must be defined in `snippet.yml` files underneath the `<snippet_root>/snippets` folder. For example, if you have `snippet_root: foo`, then you should place your module's `snippet.yml` files in `<your-module>/foo/snippets` or any nested subdirectory.
- `soc_root`: Contains additional SoCs that are available to the build system. Additional SoCs must be located in a `<soc_root>/soc` folder.
- `arch_root`: Contains additional architectures that are available to the build system. Additional architectures must be located in a `<arch_root>/arch` folder.
- `module_ext_root`: Contains `CMakeLists.txt` and `Kconfig` files for Zephyr modules, see also *Module integration files (external)*.
- `sca_root`: Contains additional *SCA* tool implementations available to the build system. Each tool must be located in `<sca_root>/sca/<tool>` folder. The folder must contain a `sca.cmake`.

Example of a `module.yaml` file containing additional roots, and the corresponding file system layout.

```
build:
  settings:
    board_root: .
    dts_root: .
    soc_root: .
    arch_root: .
    module_ext_root: .
```

requires the following folder structure:

```
<zephyr-module-root>
├── arch
├── boards
├── dts
├── modules
└── soc
```

Twister (Test Runner)

To execute both tests and samples available in modules, the Zephyr test runner (twister) should be pointed to the directories containing those samples and tests. This can be done by specifying the path to both samples and tests in the `zephyr/module.yml` file. Additionally, if a module defines out of tree boards, the module file can point twister to the path where those files are maintained in the module. For example:

```
build:
  cmake: .
samples:
  - samples
tests:
  - tests
```

(continues on next page)

(continued from previous page)

```
boards:
  - boards
```

Binary Blobs

Zephyr supports fetching and using *binary blobs*, and their metadata is contained entirely in `zephyr/module.yml`. This is because a binary blob must always be associated with a Zephyr module, and thus the blob metadata belongs in the module's description itself.

Binary blobs are fetched using *west blobs*. If `west` is *not used*, they must be downloaded and verified manually.

The `blobs` section in `zephyr/module.yml` consists of a sequence of maps, each of which has the following entries:

- `path`: The path to the binary blob, relative to the `zephyr/blobs/` folder in the module repository
- `sha256`: SHA-256 checksum of the binary blob file
- `type`: The *type of binary blob*. Currently limited to `img` or `lib`
- `version`: A version string
- `license-path`: Path to the license file for this blob, relative to the root of the module repository
- `url`: URL that identifies the location the blob will be fetched from, as well as the fetching scheme to use
- `description`: Human-readable description of the binary blob
- `doc-url`: A URL pointing to the location of the official documentation for this blob

Module Inclusion

Using West If west is installed and `ZEPHYR_MODULES` is not already set, the build system finds all the modules in your *west installation* and uses those. It does this by running *west list* to get the paths of all the projects in the installation, then filters the results to just those projects which have the necessary module metadata files.

Each project in the `west list` output is tested like this:

- If the project contains a file named `zephyr/module.yml`, then the content of that file will be used to determine which files should be added to the build, as described in the previous section.
- Otherwise (i.e. if the project has no `zephyr/module.yml`), the build system looks for `zephyr/CMakeLists.txt` and `zephyr/Kconfig` files in the project. If both are present, the project is considered a module, and those files will be added to the build.
- If neither of those checks succeed, the project is not considered a module, and is not added to `ZEPHYR_MODULES`.

Without West If you don't have west installed or don't want the build system to use it to find Zephyr modules, you can set `ZEPHYR_MODULES` yourself using one of the following options. Each of the directories in the list must contain either a `zephyr/module.yml` file or the files `zephyr/CMakeLists.txt` and `Kconfig`, as described in the previous section.

1. At the CMake command line, like this:

```
cmake -DZEPHYR_MODULES=<path-to-module1>[;<path-to-module2>[...]] ...
```

2. At the top of your application's top level `CMakeLists.txt`, like this:

```
set(ZEPHYR_MODULES <path-to-module1> <path-to-module2> [...])
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
```

   If you choose this option, make sure to set the variable **before** calling `find_package(Zephyr ...)`, as shown above.

3. In a separate CMake script which is pre-loaded to populate the CMake cache, like this:

```
# Put this in a file with a name like "zephyr-modules.cmake"
set(ZEPHYR_MODULES <path-to-module1> <path-to-module2>
  CACHE STRING "pre-cached modules")
```

   You can tell the build system to use this file by adding `-C zephyr-modules.cmake` to your CMake command line.

Not using modules If you don't have west installed and don't specify `ZEPHYR_MODULES` yourself, then no additional modules are added to the build. You will still be able to build any applications that don't require code or Kconfig options defined in an external repository.

2.10.10 Submitting changes to modules

When submitting new or making changes to existing modules the main repository Zephyr needs a reference to the changes to be able to verify the changes. In the main tree this is done using revisions. For code that is already merged and part of the tree we use the commit hash, a tag, or a branch name. For pull requests however, we require specifying the pull request number in the revision field to allow building the zephyr main tree with the changes submitted to the module.

To avoid merging changes to master with pull request information, the pull request should be marked as `DNM` (Do Not Merge) or preferably a draft pull request to make sure it is not merged by mistake and to allow for the module to be merged first and be assigned a permanent commit hash. Drafts reduce noise by not automatically notifying anyone until marked as "Ready for review". Once the module is merged, the revision will need to be changed either by the submitter or by the maintainer to the commit hash of the module which reflects the changes.

Note that multiple and dependent changes to different modules can be submitted using exactly the same process. In this case you will change multiple entries of all modules that have a pull request against them.

Process for submitting a new module

Please follow the process in *Submission and review process* and obtain the TSC approval to integrate the external source code as a module

If the request is approved, a new repository will created by the project team and initialized with basic information that would allow submitting code to the module project following the project contribution guidelines.

If a module is maintained as a fork of another project on Github, the Zephyr module related files and changes in relation to upstream need to be maintained in a special branch named `zephyr`.

Maintainers from the Zephyr project will create the repository and initialize it. You will be added as a collaborator in the new repository. Submit the module content (code) to the new repository following the guidelines described *here*, and then add a new entry to the west.yml with the following information:

```
- name: <name of repository>
  path: <path to where the repository should be cloned>
  revision: <ref pointer to module pull request>
```

For example, to add *my_module* to the manifest:

```
- name: my_module
  path: modules/lib/my_module
  revision: pull/23/head
```

Where 23 in the example above indicated the pull request number submitted to the *my_module* repository. Once the module changes are reviewed and merged, the revision needs to be changed to the commit hash from the module repository.

Process for submitting changes to existing modules

1. Submit the changes using a pull request to an existing repository following the *contribution guidelines* and *expectations*.
2. Submit a pull request changing the entry referencing the module into the west.yml of the main Zephyr tree with the following information:

```
- name: <name of repository>
  path: <path to where the repository should be cloned>
  revision: <ref pointer to module pull request>
```

For example, to add *my_module* to the manifest:

```
- name: my_module
  path: modules/lib/my_module
  revision: pull/23/head
```

Where 23 in the example above indicated the pull request number submitted to the *my_module* repository. Once the module changes are reviewed and merged, the revision needs to be changed to the commit hash from the module repository.

2.11 West (Zephyr's meta-tool)

The Zephyr project includes a swiss-army knife command line tool named `west`[1]. West is developed in its own repository.

West's built-in commands provide a multiple repository management system with features inspired by Google's Repo tool and Git submodules. West is also "pluggable": you can write your own west extension commands which add additional features to west. Zephyr uses this to provide conveniences for building applications, flashing and debugging them, and more.

Like `git` and `docker`, the top-level `west` command takes some common options, a sub-command to run, and then options and arguments for that sub-command:

```
west [common-opts] <command> [opts] <args>
```

Since west v0.8, you can also run west like this:

```
python3 -m west [common-opts] <command> [opts] <args>
```

You can run `west --help` (or `west -h` for short) to get top-level help for available west commands, and `west <command> -h` for detailed help on each command.

[1] Zephyr is an English name for the Latin Zephyrus, the ancient Greek god of the west wind.

2.11.1 Installing west

West is written in Python 3 and distributed through PyPI. Use `pip3` to install or upgrade west:

On Linux:

```
pip3 install --user -U west
```

On Windows and macOS:

```
pip3 install -U west
```

Note

See *Python and pip* for additional clarification on using the `--user` switch.

Afterwards, you can run `pip3 show -f west` for information on where the west binary and related files were installed.

Once west is installed, you can use it to *clone the Zephyr repositories*.

Structure

West's code is distributed via PyPI in a Python package named `west`. This distribution includes a launcher executable, which is also named `west` (or `west.exe` on Windows).

When west is installed, the launcher is placed by `pip3` somewhere in the user's filesystem (exactly where depends on the operating system, but should be on the `PATH` *environment variable*). This launcher is the command-line entry point to running both built-in commands like `west init`, `west update`, along with any extensions discovered in the workspace.

In addition to its command-line interface, you can also use west's Python APIs directly. See west-apis for details.

Enabling shell completion

West currently supports shell completion in the following shells:

- bash
- zsh
- fish

In order to enable shell completion, you will need to obtain the corresponding completion script and have it sourced. Using the completion scripts:

bash

One-time setup:

```
source <(west completion bash)
```

Permanent setup:

```
west completion bash > ~/west-completion.bash; echo "source ~/west-completion.bash" >> ~/.
↪bashrc
```

zsh

One-time setup:

```
source <(west completion zsh)
```

Permanent setup:

```
west completion zsh > "${fpath[1]}/_west"
```

fish

One-time setup:

```
west completion fish | source
```

Permanent setup:

```
west completion fish > $HOME/.config/fish/completions/west.fish
```

2.11.2 West Release Notes

v1.2.0

Major changes:

- New `west grep` command for running a "grep tool" in your west workspace's repositories. Currently, `git grep`, ripgrep, and standard `grep` are supported grep tools.

  To run this command to get `git grep foo` results from all cloned, active repositories, run:

  ```
  west grep foo
  ```

  Here are some other examples for running different grep commands with `west grep`:

`git grep --untracked`	`west grep --untracked foo`
`ripgrep`	`west grep --tool ripgrep foo`
`grep --recursive`	`west grep --tool grep foo`

  To switch the default grep tool in your workspace, run the appropriate command in this table:

`ripgrep`	`west config grep.tool ripgrep`
`grep`	`west config grep.tool grep`

  For more details, run `west help grep`.

Other changes:

- The manifest file format now supports a `description` field in each `projects:` element. See *Projects* for examples.
- `west list --format` now accepts `{description}` in the format string, which prints the project's `description:` value.
- `west compare` now always prints information about *The manifest-rev branch*.

Bug fixes:

- `west init` aborts if the destination directory already exists.

API changes:

- `west.commands.WestCommand` methods `check_call()` and `check_output()` now take any kwargs that can be passed on to the underlying subprocess function.

- `west.commands.WestCommand.run_subprocess()`: new wrapper around `subprocess.run()`. This could not be named `run()` because `WestCommand` already had a method by this name.
- `west.commands.WestCommand` methods `dbg()`, `inf()`, `wrn()`, and `err()` now all take an `end` kwarg, which is passed on to the call to `print()`.
- `west.manifest.Project` now has a `description` attribute, which contains the parsed value of the `description:` field in the manifest data.

v1.1.0

Major changes:

- `west compare`: new command that compares the state of the workspace against the manifest.
- Support for a new `manifest.project-filter` configuration option. See *Built-in Configuration Options* for details. The `west manifest --freeze` and `west manifest --resolve` commands currently cannot be used when this option is set. This restriction can be removed in a later release.
- Project names which contain comma (`,`) or whitespace now generate warnings. These warnings are errors if the new `manifest.project-filter` configuration option is set. The warnings may be promoted to errors in a future major version of west.

Other changes:

- `west forall` now takese a `--group` argument that can be used to restrict the command to only run in one or more groups. Run `west help forall` for details.
- All west commands will now output log messages from west API modules at warning level or higher. In addition, the `--verbose` argument to west can be used once to include informational messages, or twice to include debug messages, from all commands.

Bug fixes:

- Various improvements to error messages, debug logging, and error handling.

API changes:

- `west.manifest.Manifest.is_active()` now respects the `manifest.project-filter` configuration option's value.

v1.0.1

Major changes:

- Manifest schema version "1.0" is now available for use in this release. This is identical to the "0.13" schema version in terms of features, but can be used by applications that do not wish to use a "0.x" manifest "version:" field. See *Version* for details on this feature.

Bug fixes:

- West no longer exits with a successful error code when sent an interrupt signal. Instead, it exits with a platform-specific error code and signals to the calling environment that the process was interrupted.

v1.0.0

Major changes in this release:

- The west-apis are now declared stable. Any breaking changes will be communicated by a major version bump from v1.x.y to v2.x.y.

- West v1.0 no longer works with the Zephyr v1.14 LTS releases. This LTS has long been obsoleted by Zephyr v2.7 LTS. If you need to use Zephyr v1.14, you must use west v0.14 or earlier.
- Like the rest of Zephyr, west now requires Python v3.8 or later
- West commands no longer accept abbreviated command line arguments. For example, you must now specify `west update --keep-descendants` instead of using an abbreviation like `west update --keep-d`. This is part of a change applied to all of Zephyr's Python scripts' command-line interfaces. The abbreviations were causing problems in practice when commands were updated to add new options with similar names but different behavior to existing ones.

Other changes:

- All built-in west functions have stopped using `west.log`
- `west update`: new `--submodule-init-config` option. See commit 9ba92b05 for details.

Bug fixes:

- West extension commands that failed to load properly sometimes dumped stack. This has been fixed and west now prints a sensible error message in this case.
- `west config` now fails on malformed configuration option arguments which lack a `.` in the option name

API changes:

- The west package now contains the metadata files necessary for some static analyzers (such as mypy) to auto-detect its type annotations. See commit d9f00e24 for details.
- the deprecated `west.build` module used for Zephyr v1.14 LTS compatibility was removed
- the deprecated `west.cmake` module used for Zephyr v1.14 LTS compatibility was removed
- the `west.log` module is now deprecated. This module uses global state, which can make it awkward to use it as an API which multiple different python modules may rely on.
- The west-apis-commands module got some new APIs which lay groundwork for a future change to add a global verbosity control to a command's output, and work to remove global state from the `west` package's API:
  - New `west.commands.WestCommand.__init__()` keyword argument: `verbosity`
  - New `west.commands.WestCommand` property: `color_ui`
  - New `west.commands.WestCommand` methods, which should be used to print output from extension commands instead of writing directly to sys.stdout or sys.stderr: `inf()`, `wrn()`, `err()`, `die()`, `banner()`, `small_banner()`
  - New `west.commands.VERBOSITY` enum

v0.14.0

Bug fixes:

- West commands that were run with a bad local configuration file dumped stack in a confusing way. This has been fixed and west now prints a sensible error message in this case.
- A bug in the way west looks for the zephyr repository was fixed. The bug itself usually appeared when running an extension command like `west build` in a new workspace for the first time; this used to fail (just for the first time, not on subsequent command invocations) unless you ran the command in the workspace's top level directory.
- West now prints sensible error messages when the user lacks permission to open the manifest file instead of dumping stack traces.

API changes:

- The `west.manifest.MalformedConfig` exception type has been moved to the `west.configuration` module
- The `west.manifest.MalformedConfig` exception type has been moved to the west.configuration module
- The `west.configuration.Configuration` class now raises `MalformedConfig` instead of `RuntimeError` in some cases

v0.13.1

Bug fix:

- When calling west.manifest.Manifest.from_file() when outside of a workspace, west again falls back on the ZEPHYR_BASE environment variable to locate the workspace.

v0.13.0

New features:

- You can now associate arbitrary user data with the manifest repository itself in the `manifest: self: userdata:` value, like so:

```
manifest:
  self:
    userdata: <any YAML value can go here>
```

Bug fixes:

- The path to the manifest repository reported by west could be incorrect in certain circumstances detailed in [issue #572](https://github.com/zephyrproject-rtos/west/issues/572). This has been fixed as part of a larger overhaul of path handling support in the `west.manifest` API module.
- The `west.Manifest.ManifestProject.__repr__` return value was fixed

API changes:

- `west.configuration.Configuration`: new object-oriented interface to the current configuration. This reflects the system, global, and workspace-local configuration values, and allows you to read, write, and delete configuration options from any or all of these locations.
- `west.commands.WestCommand`:
  - `config`: new attribute, returns a `Configuration` object or aborts the program if none is set. This is always usable from within extension command `do_run()` implementations.
  - `has_config`: new boolean attribute, which is `True` if and only if reading `self.config` will abort the program.
- The path handling in the `west.manifest` package has been overhauled in a backwards-incompatible way. For more details, see commit [56cfe8d1d1](https://github.com/zephyrproject-rtos/west/commit/56cfe8d1d1f3c9b45de3e793c738acd62db52aca).
- `west.manifest.Manifest.validate()`: this now returns the validated data as a Python dict. This can be useful if the value passed to this function was a str, and the dict is desired.
- `west.manifest.Manifest`: new:
  - path attributes `abspath`, `posixpath`, `relative_path`, `yaml_path`, `repo_path`, `repo_posixpath`

  - `userdata` attribute, which contains the parsed value from `manifest: self: userdata:`, or is None
  - `from_topdir()` factory method
- `west.manifest.ManifestProject`: new `userdata` attribute, which also contains the parsed value from `manifest: self: userdata:`, or is None
- `west.manifest.ManifestImportFailed`: the constructor can now take any value; this can be used to reflect failed imports from a *map* or other compound value.
- Deprecated configuration APIs:

  The following APIs are now deprecated in favor of using a `Configuration` object. Usually this will be done via `self.config` from a `WestCommand` instance, but this can be done directly by instantiating a `Configuration` object for other usages.

  - `west.configuration.config`
  - `west.configuration.read_config`
  - `west.configuration.update_config`
  - `west.configuration.delete_config`

v0.12.0

New features:

- West now works on the MSYS2 platform.
- West manifest files can now contain arbitrary user data associated with each project. See *Repository user data* for details.

Bug fixes:

- The `west list` command's `{sha}` format key has been fixed for the manifest repository; it now prints `N/A` ("not applicable") as expected.

API changes:

- The `west.manifest.Project.userdata` attribute was added to support project user data.

v0.11.1

New features:

- `west status` now only prints output for projects which have a nonempty status.

Bug fixes:

- The manifest file parser was incorrectly allowing project names which contain the path separator characters `/` and `\`. These invalid characters are now rejected.

  Note: if you need to place a project within a subdirectory of the workspace topdir, use the `path:` key. If you need to customize a project's fetch URL relative to its remote `url-base:`, use `repo-path:`. See *Projects* for examples.
- The changes made in west v0.10.1 to the `west init --manifest-rev` option which selected the default branch name were leaving the manifest repository in a detached HEAD state. This has been fixed by using `git clone` internally instead of `git init` and `git fetch`. See issue #522 for details.
- The `WEST_CONFIG_LOCAL` environment variable now correctly overrides the default location, `<workspace topdir>/.west/config`.

- `west update --fetch=smart` (`smart` is the default) now correctly skips fetches for project revisions which are lightweight tags (it already worked correctly for annotated tags; only lightweight tags were unnecessarily fetched).

Other changes:

- The fix for issue #522 mentioned above introduces a new restriction. The `west init --manifest-rev` option value, if given, must now be either a branch or a tag. In particular, "pseudo-branches" like GitHub's `pull/1234/head` references which could previously be used to fetch a pull request can no longer be passed to `--manifest-rev`. Users must now fetch and check out such revisions manually after running `west init`.

API changes:

- `west.manifest.Manifest.get_projects()` avoids incorrect results in some edge cases described in issue #523.
- `west.manifest.Project.sha()` now works correctly for tag revisions. (This applies to both lightweight and annotated tags.)

v0.11.0

New features:

- `west update` now supports `--narrow`, `--name-cache`, and `--path-cache` options. These can be influenced by the `update.narrow`, `update.name-cache`, and `update.path-cache` *Configuration* options. These can be used to optimize the speed of the update.
- `west update` now supports a `--fetch-opt` option that will be passed to the `git fetch` command used to fetch remote revisions when updating each project.

Bug fixes:

- `west update` now synchronizes Git submodules in projects by default. This avoids issues if the URL changes in the manifest file from when the submodule was first initialized. This behavior can be disabled by setting the `update.sync-submodules` configuration option to `false`.

Other changes:

- the west-apis-manifest module has fixed docstrings for the Project class

v0.10.1

New features:

- The *west init* command's `--manifest-rev` (`--mr`) option no longer defaults to `master`. Instead, the command will query the repository for its default branch name and use that instead. This allows users to move from `master` to `main` without breaking scripts that do not provide this option.

v0.10.0

New features:

- The `name` key in a project's *submodules list* is now optional.

Bug fixes:

- West now checks that the manifest schema version is one of the explicitly allowed values documented in *Version*. The old behavior was just to check that the schema version was

newer than the west version where the `manifest: version:` key was introduced. This incorrectly allowed invalid schema versions, like `0.8.2`.

Other changes:

- A manifest file's `group-filter` is now propagated through an `import`. This is a change from how west v0.9.x handled this. In west v0.9.x, only the top level manifest file's `group-filter` had any effect; the group filter lists from any imported manifests were ignored.

  Starting with west v0.10.0, the group filter lists from imported manifests are also imported. For details, see *Group Filters and Imports*.

  The new behavior will take effect if `manifest: version:` is not given or is at least `0.10`. The old behavior is still available in the top level manifest file only with an explicit `manifest: version: 0.9`. See *Version* for more information on schema versions.

  See west pull request #482 for the motivation for this change and additional context.

v0.9.1

Bug fixes:

- Commands like `west manifest --resolve` now correctly include group and group filter information.

Other changes:

- West now warns if you combine `import` with `group-filter`. Semantics for this combination have changed starting with v0.10.x. See the v0.10.0 release notes above for more information.

v0.9.0

Warning

The `west config` fix described below comes at a cost: any comments or other manual edits in configuration files will be removed when setting a configuration option via that command or the `west.configuration` API.

Warning

Combining the `group-filter` feature introduced in this release with manifest imports is discouraged. The resulting behavior has changed in west v0.10.

New features:

- West manifests now support *Git Submodules in Projects*. This allows you to clone Git submodules into a west project repository in addition to the project repository itself.
- West manifests now support *Project Groups*. Project groups can be enabled and disabled to determine what projects are "active", and therefore will be acted upon by the following commands: `west update`, `west list`, `west diff`, `west status`, `west forall`.
- `west update` no longer updates inactive projects by default. It now supports a `--group-filter` option which allows for one-time modifications to the set of enabled and disabled project groups.

- Running `west list`, `west diff`, `west status`, or `west forall` with no arguments does not print information for inactive projects by default. If the user specifies a list of projects explicitly at the command line, output for them is included regardless of whether they are active.

  These commands also now support `--all` arguments to include all projects, even inactive ones.

- `west list` now supports a `{groups}` format string key in its `--format` argument.

Bug fixes:

- The `west config` command and `west.configuration` API did not correctly store some configuration values, such as strings which contain commas. This has been fixed; see commit 36f3f91e for details.
- A manifest file with an empty `manifest: self: path:` value is invalid, but west used to let it pass silently. West now rejects such manifests.
- A bug affecting the behavior of the `west init -l .` command was fixed; see issue #435.

API changes:

- added `west.manifest.Manifest.is_active()`
- added `west.manifest.Manifest.group_filter`
- added `submodules` attribute to `west.manifest.Project`, which has newly added type `west.manifest.Submodule`

Other changes:

- The *Manifest Imports* feature now supports the terms `allowlist` and `blocklist` instead of `whitelist` and `blacklist`, respectively.

  The old terms are still supported for compatibility, but the documentation has been updated to use the new ones exclusively.

v0.8.0

This is a feature release which changes the manifest schema by adding support for a `path-prefix:` key in an `import:` mapping, along with some other features and fixes.

- Manifest import mappings now support a `path-prefix:` key, which places the project and its imported repositories in a subdirectory of the workspace. See *Example 3.4: Import into a subdirectory* for an example.
- The west command line application can now also be run using `python3 -m west`. This makes it easier to run west under a particular Python interpreter without modifying the *PATH* environment variable.
- *west manifest –path* prints the absolute path to west.yml
- `west init` now supports an `--mf foo.yml` option, which initializes the workspace using `foo.yml` instead of `west.yml`.
- `west list` now prints the manifest repository's path using the `manifest.path` *configuration option*, which may differ from the `self: path:` value in the manifest data. The old behavior is still available, but requires passing a new `--manifest-path-from-yaml` option.
- Various Python API changes; see west-apis for details.

v0.7.3

This is a bugfix release.

- Fix an error where a failed import could leave the workspace in an unusable state (see [PR #415](https://github.com/zephyrproject-rtos/west/pull/415) for details)

v0.7.2

This is a bugfix and minor feature release.

- Filter out duplicate extension commands brought in by manifest imports
- Fix `west.Manifest.get_projects()` when finding the manifest repository by path

v0.7.1

This is a bugfix and minor feature release.

- `west update --stats` now prints timing for operations which invoke a subprocess, time spent in west's Python process for each project, and total time updating each project.
- `west topdir` always prints a POSIX style path
- minor console output changes

v0.7.0

The main user-visible feature in west 0.7 is the *Manifest Imports* feature. This allows users to load west manifest data from multiple different files, resolving the results into a single logical manifest.

Additional user-visible changes:

- The idea of a "west installation" has been renamed to "west workspace" in this documentation and in the west API documentation. The new term seems to be easier for most people to work with than the old one.
- West manifests now support a *schema version*.
- The "west config" command can now be run outside of a workspace, e.g. to run `west config --global section.key value` to set a configuration option's value globally.
- There is a new *west topdir* command, which prints the root directory of the current west workspace.
- The `west -vv init` command now prints the git operations being performed, and their results.
- The restriction that no project can be named "manifest" is now enforced; the name "manifest" is reserved for the manifest repository, and is usable as such in commands like `west list manifest`, instead of `west list path-to-manifest-repository` being the only way to say that
- It's no longer an error if there is no project named "zephyr". This is part of an effort to make west generally usable for non-Zephyr use cases.
- Various bug fixes.

The developer-visible changes to the west-apis are:

- west.build and west.cmake: deprecated; this is Zephyr-specific functionality and should never have been part of west. Since Zephyr v1.14 LTS relies on it, it will continue to be included in the distribution, but will be removed when that version of Zephyr is obsoleted.
- west.commands:
  - WestCommand.requires_installation: deprecated; use requires_workspace instead

  - WestCommand.requires_workspace: new
  - WestCommand.has_manifest: new
  - WestCommand.manifest: this is now settable
- west.configuration: callers can now identify the workspace directory when reading and writing configuration files
- west.log:
  - msg(): new
- west.manifest:
  - The module now uses the standard logging module instead of west.log
  - QUAL_REFS_WEST: new
  - SCHEMA_VERSION: new
  - Defaults: removed
  - Manifest.as_dict(): new
  - Manifest.as_frozen_yaml(): new
  - Manifest.as_yaml(): new
  - Manifest.from_file() and from_data(): these factory methods are more flexible to use and less reliant on global state
  - Manifest.validate(): new
  - ManifestImportFailed: new
  - ManifestProject: semi-deprecated and will likely be removed later.
  - Project: the constructor now takes a topdir argument
  - Project.format() and its callers are removed. Use f-strings instead.
  - Project.name_and_path: new
  - Project.remote_name: new
  - Project.sha() now captures stderr
  - Remote: removed

West now requires Python 3.6 or later. Additionally, some features may rely on Python dictionaries being insertion-ordered; this is only an implementation detail in CPython 3.6, but it is part of the language specification as of Python 3.7.

v0.6.3

This point release fixes an error in the behavior of the deprecated `west.cmake` module.

v0.6.2

This point release fixes an error in the behavior of `west update --fetch=smart`, introduced in v0.6.1.

All v0.6.1 users must upgrade.

v0.6.1

Warning

Do not use this point release. Make sure to use v0.6.2 instead.

The user-visible features in this point release are:

- The *west update* command has a new `--fetch` command line flag and `update.fetch` *configuration option*. The default value, "smart", skips fetching SHAs and tags which are available locally.
- Better and more consistent error-handling in the `west diff`, `west status`, `west forall`, and `west update` commands. Each of these commands can operate on multiple projects; if a subprocess related to one project fails, these commands now continue to operate on the rest of the projects. All of them also now report a nonzero error code from the west process if any of these subprocesses fails (this was previously not true of `west forall` in particular).
- The *west manifest* command also handles errors better.
- The *west list* command now works even when the projects are not cloned, as long as its format string only requires information which can be read from the manifest file. It still fails if the format string requires data stored in the project repository, e.g. if it includes the `{sha}` format string key.
- Commands and options which operate on git revisions now accept abbreviated SHAs. For example, `west init --mr SHA_PREFIX` now works. Previously, the `--mr` argument needed to be the entire 40 character SHA if it wasn't a branch or a tag.

The developer-visible changes to the west-apis are:

- west.log.banner(): new
- west.log.small_banner(): new
- west.manifest.Manifest.get_projects(): new
- west.manifest.Project.is_cloned(): new
- west.commands.WestCommand instances can now access the parsed Manifest object via a new self.manifest property during the do_run() call. If read, it returns the Manifest object or aborts the command if it could not be parsed.
- west.manifest.Project.git() now has a capture_stderr kwarg

v0.6.0

- No separate bootstrapper

  In west v0.5.x, the program was split into two components, a bootstrapper and a per-installation clone. See Multiple Repository Management in the v1.14 documentation for more details.

  This is similar to how Google's Repo tool works, and lets west iterate quickly at first. It caused confusion, however, and west is now stable enough to be distributed entirely as one piece via PyPI.

  From v0.6.x onwards, all of the core west commands and helper classes are part of the west package distributed via PyPI. This eliminates complexity and makes it possible to import west modules from anywhere in the system, not just extension commands.

- The `selfupdate` command still exists for backwards compatibility, but now simply exits after printing an error message.
- Manifest syntax changes
  - A west manifest file's `projects` elements can now specify their fetch URLs directly, like so:

```
manifest:
  projects:
    - name: example-project-name
      url: https://github.com/example/example-project
```

    Project elements with `url` attributes set in this way may not also have `remote` attributes.
  - Project names must be unique: this restriction is needed to support future work, but was not possible in west v0.5.x because distinct projects may have URLs with the same final pathname component, like so:

```
manifest:
  remotes:
    - name: remote-1
      url-base: https://github.com/remote-1
    - name: remote-2
      url-base: https://github.com/remote-2
  projects:
    - name: project
      remote: remote-1
      path: remote-1-project
    - name: project
      remote: remote-2
      path: remote-2-project
```

    These manifests can now be written with projects that use `url` instead of `remote`, like so:

```
manifest:
  projects:
    - name: remote-1-project
      url: https://github.com/remote-1/project
    - name: remote-2-project
      url: https://github.com/remote-2/project
```

- The `west list` command now supports a `{sha}` format string key
- The default format string for `west list` was changed to `"{name:12} {path:28} {revision:40} {url}"`.
- The command `west manifest --validate` can now be run to load and validate the current manifest file, among other error-handling fixes related to manifest parsing.
- Incompatible API changes were made to west's APIs. Further changes are expected until API stability is declared in west v1.0.
  - The `west.manifest.Project` constructor's `remote` and `defaults` positional arguments are now kwargs. A new `url` kwarg was also added; if given, the `Project` URL is set to that value, and the `remote` kwarg is ignored.
  - `west.manifest.MANIFEST_SECTIONS` was removed. There is only one section now, namely `manifest`. The *sections* kwargs in the `west.manifest.Manifest` factory methods and constructor were also removed.
  - The `west.manifest.SpecialProject` class was removed. Use `west.manifest.ManifestProject` instead.

v0.5.x

West v0.5.x is the first version used widely by the Zephyr Project as part of its v1.14 Long-Term Support (LTS) release. The west v0.5.x documentation is available as part of the Zephyr's v1.14 documentation.

West's main features in v0.5.x are:

- Multiple repository management using Git repositories, including self-update of west itself
- Hierarchical configuration files
- Extension commands

Versions Before v0.5.x

Tags in the west repository before v0.5.x are prototypes which are of historical interest only.

2.11.3 Troubleshooting West

This page covers common issues with west and how to solve them.

`west update` fetching failures

One good way to troubleshoot fetching issues is to run `west update` in verbose mode, like this:

```
west -v update
```

The output includes Git commands run by west and their outputs. Look for something like this:

```
=== updating your_project (path/to/your/project):
west.manifest: your_project: checking if cloned
[...other west.manifest logs...]
--- your_project: fetching, need revision SOME_SHA
west.manifest: running 'git fetch ... https://github.com/your-username/your_project ...' in␣
↪/some/directory
```

The `git fetch` command example in the last line above is what needs to succeed.

One strategy is to go to `/path/to/your/project`, copy/paste and run the entire `git fetch` command, then debug from there using the documentation for your credential storage helper.

If you're behind a corporate firewall and may have proxy or other issues, `curl -v FETCH_URL` (for HTTPS URLs) or `ssh -v FETCH_URL` (for SSH URLs) may be helpful.

If you can get the `git fetch` command to run successfully without prompting for a password when you run it directly, you will be able to run `west update` without entering your password in that same shell.

"'west' is not recognized as an internal or external command, operable program or batch file."

On Windows, this means that either west is not installed, or your *PATH* environment variable does not contain the directory where pip installed `west.exe`.

First, make sure you've installed west; see *Installing west*. Then try running `west` from a new `cmd.exe` window. If that still doesn't work, keep reading.

You need to find the directory containing `west.exe`, then add it to your *PATH*. (This *PATH* change should have been done for you when you installed Python and pip, so ordinarily you should not need to follow these steps.)

Run this command in `cmd.exe`:

```
pip3 show west
```

Then:

1. Look for a line in the output that looks like `Location: C:\foo\python\python38\lib\site-packages`. The exact location will be different on your computer.
2. Look for a file named `west.exe` in the `scripts` directory `C:\foo\python\python38\scripts`.

   Important

   Notice how `lib\site-packages` in the `pip3 show` output was changed to `scripts`!

3. If you see `west.exe` in the `scripts` directory, add the full path to `scripts` to your *PATH* using a command like this:

   ```
   setx PATH "%PATH%;C:\foo\python\python38\scripts"
   ```

   Do not just copy/paste this command. The `scripts` directory location will be different on your system.

4. Close your `cmd.exe` window and open a new one. You should be able to run `west`.

"Error: unexpected keyword argument 'requires_workspace'"

This error occurs on some Linux distributions after upgrading to west 0.7.0 or later from 0.6.x. For example:

```
$ west update
[... stack trace ...]
TypeError: __init__() got an unexpected keyword argument 'requires_workspace'
```

This appears to be a problem with the distribution's pip; see this comment in west issue 373 for details. Some versions of **Ubuntu** and **Linux Mint** are known to have this problem. Some users report issues on Fedora as well.

Neither macOS nor Windows users have reported this issue. There have been no reports of this issue on other Linux distributions, like Arch Linux, either.

Workaround 1: remove the old version, then upgrade:

```
$ pip3 show west | grep Location: | cut -f 2 -d ' '
/home/foo/.local/lib/python3.6/site-packages
$ rm -r /home/foo/.local/lib/python3.6/site-packages/west
$ pip3 install --user west==0.7.0
```

Workaround 2: install west in a Python virtual environment

One option is to use the venv module that's part of the Python 3 standard library. Some distributions remove this module from their base Python 3 packages, so you may need to do some additional work to get it installed on your system.

"invalid choice: 'build'" (or 'flash', etc.)

If you see an unexpected error like this when trying to run a Zephyr extension command (like *west flash*, *west build*, etc.):

```
$ west build [...]
west: error: argument <command>: invalid choice: 'build' (choose from 'init', [...])

$ west flash [...]
west: error: argument <command>: invalid choice: 'flash' (choose from 'init', [...])
```

The most likely cause is that you're running the command outside of a *west workspace*. West needs to know where your workspace is to find *Extensions*.

To fix this, you have two choices:

1. Run the command from inside a workspace (e.g. the `zephyrproject` directory you created when you *got started*).

   For example, create your build directory inside the workspace, or run `west flash --build-dir YOUR_BUILD_DIR` from inside the workspace.

2. Set the *ZEPHYR_BASE environment variable* and re-run the west extension command. If set, west will use *ZEPHYR_BASE* to find your workspace.

If you're unsure whether a command is built-in or an extension, run `west help` from inside your workspace. The output prints extension commands separately, and looks like this for mainline Zephyr:

```
$ west help

built-in commands for managing git repositories:
  init:                 create a west workspace
  [...]

other built-in commands:
  help:                 get help for west or a command
  [...]

extension commands from project manifest (path: zephyr):
  build:                compile a Zephyr application
  flash:                flash and run a binary on a board
  [...]
```

"invalid choice: 'post-init'"

If you see this error when running `west init`:

```
west: error: argument <command>: invalid choice: 'post-init'
(choose from 'init', 'update', 'list', 'manifest', 'diff',
'status', 'forall', 'config', 'selfupdate', 'help')
```

Then you have an old version of west installed, and are trying to use it in a workspace that requires a more recent version.

The easiest way to resolve this issue is to upgrade west and retry as follows:

1. Install the latest west with the `-U` option for `pip3 install` as shown in *Installing west*.
2. Back up any contents of `zephyrproject/.west/config` that you want to save. (If you don't have any configuration options set, it's safe to skip this step.)
3. Completely remove the `zephyrproject/.west` directory (if you don't, you will get the "already in a workspace" error message discussed next).

4. Run `west init` again.

"already in an installation"

You may see this error when running `west init` with west 0.6:

```
FATAL ERROR: already in an installation (<some directory>), aborting
```

If this is unexpected and you're really trying to create a new west workspace, then it's likely that west is using the *ZEPHYR_BASE environment variable* to locate a workspace elsewhere on your system.

This is intentional; it allows you to put your Zephyr applications in any directory and still use west to build, flash, and debug them, for example.

To resolve this issue, unset *ZEPHYR_BASE* and try again.

2.11.4 Basics

This page introduces west's basic concepts and provides references to further reading.

West's built-in commands allow you to work with *projects* (Git repositories) under a common *workspace* directory.

West works in the following manner: the `west init` command creates the *west workspace*, and clones the *manifest repo*, while the `west update` command initially clones, and later updates, the *projects* listed in the manifest in the workspace.

Example workspace

If you've followed the *Getting Started Guide*, your local *west workspace*, which in this case is the folder named `zephyrproject` as well as all its subfolders, looks like this:

```
zephyrproject/                # west topdir
├── .west/                    # marks the location of the topdir
│   └── config                # per-workspace local configuration file
│
│   # The manifest repository, never modified by west after creation:
├── zephyr/                   # .git/ repo
│   ├── west.yml              # manifest file
│   └── [... other files ...]
│
│   # Projects managed by west:
├── modules/
│   └── lib/
│       └── zcbor/            # .git/ project
├── net-tools/                # .git/ project
└── [ ... other projects ...]
```

Workspace concepts

Here are the basic concepts you should understand about this structure. Additional details are in *Workspaces*.

topdir
: Above, `zephyrproject` is the name of the workspace's top level directory, or *topdir*. (The

name `zephyrproject` is just an example – it could be anything, like `z`, `my-zephyr-workspace`, etc.)

You'll typically create the topdir and a few other files and directories using *west init*.

.west directory
The topdir contains the `.west` directory. When west needs to find the topdir, it searches for `.west`, and uses its parent directory. The search starts from the current working directory (and starts again from the location in the *ZEPHYR_BASE* environment variable as a fallback if that fails).

configuration file
The file `.west/config` is the workspace's *local configuration file*.

manifest repository
Every west workspace contains exactly one *manifest repository*, which is a Git repository containing a *manifest file*. The location of the manifest repository is given by the *manifest.path configuration option* in the local configuration file.

For upstream Zephyr, `zephyr` is the manifest repository, but you can configure west to use any Git repository in the workspace as the manifest repository. The only requirement is that it contains a valid manifest file. See *Topologies supported* for information on other options, and *West Manifests* for details on the manifest file format.

manifest file
The manifest file is a YAML file that defines *projects*, which are the additional Git repositories in the workspace managed by west. The manifest file is named `west.yml` by default; this can be overridden using the `manifest.file` local configuration option.

You use the *west update* command to update the workspace's projects based on the contents of the manifest file.

projects
Projects are Git repositories managed by west. Projects are defined in the manifest file and can be located anywhere inside the workspace. In the above example workspace, `zcbor` and `net-tools` are projects.

By default, the Zephyr *build system* uses west to get the locations of all the projects in the workspace, so any code they contain can be used as *Modules (External projects)*. Note however that modules and projects *are conceptually different*.

extensions
Any repository known to west (either the manifest repository or any project repository) can define *Extensions*. Extensions are extra west commands you can run when using that workspace.

The zephyr repository uses this feature to provide Zephyr-specific commands like *west build*. Defining these as extensions keeps west's core agnostic to the specifics of any workspace's Zephyr version, etc.

ignored files
A workspace can contain additional Git repositories or other files and directories not managed by west. West basically ignores anything in the workspace except `.west`, the manifest repository, and the projects specified in the manifest file.

west init and west update

The two most important workspace-related commands are `west init` and `west update`.

`west init` basics This command creates a west workspace.

Important

West doesn't change your manifest repository contents after `west init` is run. Use ordinary Git commands to pull new versions, etc.

You will typically run it once, like this:

```
west init -m https://github.com/zephyrproject-rtos/zephyr --mr v2.5.0 zephyrproject
```

This will:

1. Create the topdir, `zephyrproject`, along with `.west` and `.west/config` inside it
2. Clone the manifest repository from https://github.com/zephyrproject-rtos/zephyr, placing it into `zephyrproject/zephyr`
3. Check out the `v2.5.0` git tag in your local zephyr clone
4. Set `manifest.path` to `zephyr` in `.west/config`
5. Set `manifest.file` to `west.yml`

Your workspace is now almost ready to use; you just need to run `west update` to clone the rest of the projects into the workspace to finish.

For more details, see *west init*.

`west update` basics This command makes sure your workspace contains Git repositories matching the projects in the manifest file.

Important

Whenever you check out a different revision in your manifest repository, you should run `west update` to make sure your workspace contains the project repositories the new revision expects.

The `west update` command reads the manifest file's contents by:

1. Finding the topdir. In the `west init` example above, that means finding `zephyrproject`.
2. Loading `.west/config` in the topdir to read the `manifest.path` (e.g. `zephyr`) and `manifest.file` (e.g. `west.yml`) options.
3. Loading the manifest file given by these options (e.g. `zephyrproject/zephyr/west.yml`).

It then uses the manifest file to decide where missing projects should be placed within the workspace, what URLs to clone them from, and what Git revisions should be checked out locally. Project repositories which already exist are updated in place by fetching and checking out their respective Git revisions in the manifest file.

For more details, see *west update*.

Other built-in commands

See *Built-in commands*.

Zephyr Extensions

See the following pages for information on Zephyr's extension commands:

- *Building, Flashing and Debugging*
- *Signing Binaries*
- *Additional Zephyr extension commands*
- *Enabling shell completion*

Troubleshooting

See *Troubleshooting West*.

2.11.5 Built-in commands

This page describes west's built-in commands, some of which were introduced in *Basics*, in more detail.

Some commands are related to Git commands with the same name, but operate on the entire workspace. For example, `west diff` shows local changes in multiple Git repositories in the workspace.

Some commands take projects as arguments. These arguments can be project names as specified in the manifest file, or (as a fallback) paths to them on the local file system. Omitting project arguments to commands which accept them (such as `west list`, `west forall`, etc.) usually defaults to using all projects in the manifest file plus the manifest repository itself.

For additional help, run `west <command> -h` (e.g. `west init -h`).

west init

This command creates a west workspace. It can be used in two ways:

1. Cloning a new manifest repository from a remote URL
2. Creating a workspace around an existing local manifest repository

Option 1: to clone a new manifest repository from a remote URL, use:

```
west init [-m URL] [--mr REVISION] [--mf FILE] [directory]
```

The new workspace is created in the given `directory`, creating a new `.west` inside this directory. You can give the manifest URL using the `-m` switch, the initial revision to check out using `--mr`, and the location of the manifest file within the repository using `--mf`.

For example, running:

```
west init -m https://github.com/zephyrproject-rtos/zephyr --mr v1.14.0 zp
```

would clone the upstream official zephyr repository into `zp/zephyr`, and check out the `v1.14.0` release. This command creates `zp/.west`, and set the `manifest.path` *configuration option* to `zephyr` to record the location of the manifest repository in the workspace. The default manifest file location is used.

The `-m` option defaults to `https://github.com/zephyrproject-rtos/zephyr`. The `--mf` option defaults to `west.yml`. Since west v0.10.1, west will use the default branch in the manifest repository unless the `--mr` option is used to override it. (In prior versions, `--mr` defaulted to `master`.)

If no `directory` is given, the current working directory is used.

Option 2: to create a workspace around an existing local manifest repository, use:

```
west init -l [--mf FILE] directory
```

This creates `.west` **next to** `directory` in the file system, and sets `manifest.path` to `directory`.

As above, `--mf` defaults to `west.yml`.

Reconfiguring the workspace:

If you change your mind later, you are free to change `manifest.path` and `manifest.file` using *west config* after running `west init`. Just be sure to run `west update` afterwards to update your workspace to match the new manifest file.

west update

```
west update [-f {always,smart}] [-k] [-r]
            [--group-filter FILTER] [--stats] [PROJECT ...]
```

Which projects are updated:

By default, this command parses the manifest file, usually `west.yml`, and updates each project specified there. If your manifest uses *project groups*, then only the active projects are updated.

To operate on a subset of projects only, give `PROJECT` argument(s). Each `PROJECT` is either a project name as given in the manifest file, or a path that points to the project within the workspace. If you specify projects explicitly, they are updated regardless of whether they are active.

Project update procedure:

For each project that is updated, this command:

1. Initializes a local Git repository for the project in the workspace, if it does not already exist
2. Inspects the project's `revision` field in the manifest, and fetches it from the remote if it is not already available locally
3. Sets the project's *manifest-rev* branch to the commit specified by the revision in the previous step
4. Checks out `manifest-rev` in the local working copy as a detached HEAD
5. If the manifest file specifies a *submodules* key for the project, recursively updates the project's submodules as described below.

To avoid unnecessary fetches, `west update` will not fetch project `revision` values which are Git SHAs or tags that are already available locally. This is the behavior when the `-f` (`--fetch`) option has its default value, `smart`. To force this command to fetch from project remotes even if the revisions appear to be available locally, either use `-f always` or set the `update.fetch` *configuration option* to `always`. SHAs may be given as unique prefixes as long as they are acceptable to Git[1].

If the project `revision` is a Git ref that is neither a tag nor a SHA (i.e. if the project is tracking a branch), `west update` always fetches, regardless of `-f` and `update.fetch`.

Some branch names might look like short SHAs, like `deadbeef`. West treats these like SHAs. You can disambiguate by prefixing the `revision` value with `refs/heads/`, e.g. `revision: refs/heads/deadbeef`.

For safety, `west update` uses `git checkout --detach` to check out a detached `HEAD` at the manifest revision for each updated project, leaving behind any branches which were already checked out. This is typically a safe operation that will not modify any of your local branches.

However, if you had added some local commits onto a previously detached `HEAD` checked out by west, then git will warn you that you've left behind some commits which are no longer referred

[1] West may fetch all refs from the Git server when given a SHA as a revision. This is because some Git servers have historically not allowed fetching SHAs directly.

to by any branch. These may be garbage-collected and lost at some point in the future. To avoid this if you have local commits in the project, make sure you have a local branch checked out before running `west update`.

If you would rather rebase any locally checked out branches instead, use the `-r` (`--rebase`) option.

If you would like `west update` to keep local branches checked out as long as they point to commits that are descendants of the new `manifest-rev`, use the `-k` (`--keep-descendants`) option.

Note

`west update --rebase` will fail in projects that have git conflicts between your branch and new commits brought in by the manifest. You should immediately resolve these conflicts as you usually do with `git`, or you can use `git -C <project_path> rebase --abort` to ignore incoming changes for the moment.

With a clean working tree, a plain `west update` never fails because it does not try to hold on to your commits and simply leaves them aside.

`west update --keep-descendants` offers an intermediate option that never fails either but does not treat all projects the same:

- in projects where your branch diverged from the incoming commits, it does not even try to rebase and leaves your branches behind just like a plain `west update` does;
- in all other projects where no rebase or merge is needed it keeps your branches in place.

One-time project group manipulation:

The `--group-filter` option can be used to change which project groups are enabled or disabled for the duration of a single `west update` command. See *Project Groups* for details on the project group feature.

The `west update` command behaves as if the `--group-filter` option's value were appended to the `manifest.group-filter` *configuration option*.

For example, running `west update --group-filter=+foo,-bar` would behave the same way as if you had temporarily appended the string `"+foo,-bar"` to the value of `manifest.group-filter`, run `west update`, then restored `manifest.group-filter` to its original value.

Note that using the syntax `--group-filter=VALUE` instead of `--group-filter VALUE` avoids issues parsing command line options if you just want to disable a single group, e.g. `--group-filter=-bar`.

Submodule update procedure:

If a project in the manifest has a `submodules` key, the submodules are updated as follows, depending on the value of the `submodules` key.

If the project has `submodules: true`, west first synchronizes the project's submodules with:

```
git submodule sync --recursive
```

West then runs one of the following in the project repository, depending on whether you run `west update` with the `--rebase` option or without it:

```
# without --rebase, e.g. "west update":
git submodule update --init --checkout --recursive

# with --rebase, e.g. "west update --rebase":
git submodule update --init --rebase --recursive
```

Otherwise, the project has `submodules: <list-of-submodules>`. In this case, west synchronizes the project's submodules with:

```
git submodule sync --recursive -- <submodule-path>
```

Then it updates each submodule in the list as follows, depending on whether you run `west update` with the `--rebase` option or without it:

```
# without --rebase, e.g. "west update":
git submodule update --init --checkout --recursive <submodule-path>

# with --rebase, e.g. "west update --rebase":
git submodule update --init --rebase --recursive <submodule-path>
```

The `git submodule sync` commands are skipped if the `update.sync-submodules` *Configuration* option is false.

Other project commands

West has a few more commands for managing the projects in the workspace, which are summarized here. Run `west <command> -h` for detailed help.

- `west compare`: compare the state of the workspace against the manifest
- `west diff`: run `git diff` in local project repositories
- `west forall`: run an arbitrary command in local project repositories
- `west grep`: search for patterns in local project repositories
- `west list`: print a line of information about each project in the manifest, according to a format string
- `west manifest`: manage the manifest file. See *Manifest Command*.
- `west status`: run `git status` in local project repositories

Other built-in commands

Finally, here is a summary of other built-in commands.

- `west config`: get or set *configuration options*
- `west topdir`: print the top level directory of the west workspace
- `west help`: get help about a command, or print information about all commands in the workspace, including *Extensions*

2.11.6 Workspaces

This page describes the *west workspace* concept introduced in *Basics* in more detail.

The `manifest-rev` branch

West creates and controls a Git branch named `manifest-rev` in each project. This branch points to the revision that the manifest file specified for the project at the time *west update* was last run. Other workspace management commands may use `manifest-rev` as a reference point for the upstream revision as of this latest update. Among other purposes, the `manifest-rev` branch allows the manifest file to use SHAs as project revisions.

Although `manifest-rev` is a normal Git branch, west will recreate and/or reset it on the next update. For this reason, it is **dangerous** to check it out or otherwise modify it yourself. For

instance, any commits you manually add to this branch may be lost the next time you run `west update`. Instead, check out a local branch with another name, and either rebase it on top of a new `manifest-rev`, or merge `manifest-rev` into it.

Note

West does not create a `manifest-rev` branch in the manifest repository, since west does not manage the manifest repository's branches or revisions.

The `refs/west/*` Git refs

West also reserves all Git refs that begin with `refs/west/` (such as `refs/west/foo`) for itself in local project repositories. Unlike `manifest-rev`, these refs are not regular branches. West's behavior here is an implementation detail; users should not rely on these refs' existence or behavior.

Private repositories

You can use west to fetch from private repositories. There is nothing west-specific about this.

The `west update` command essentially runs `git fetch YOUR_PROJECT_URL` when a project's `manifest-rev` branch must be updated to a newly fetched commit. It's up to your environment to make sure the fetch succeeds.

You can either enter the password manually or use any of the credential helpers built in to Git. Since Git has credential storage built in, there is no need for a west-specific feature.

The following sections cover common cases for running `west update` without having to enter your password, as well as how to troubleshoot issues.

Fetching via HTTPS On Windows when fetching from GitHub, recent versions of Git prompt you for your GitHub password in a graphical window once, then store it for future use (in a default installation). Passwordless fetching from GitHub should therefore work "out of the box" on Windows after you have done it once.

In general, you can store your credentials on disk using the "store" git credential helper. See the git-credential-store manual page for details.

To use this helper for all the repositories in your workspace, run:

```
west forall -c "git config credential.helper store"
```

To use this helper on just the projects `foo` and `bar`, run:

```
west forall -c "git config credential.helper store" foo bar
```

To use this helper by default on your computer, run:

```
git config --global credential.helper store
```

On GitHub, you can set up a personal access token to use in place of your account password. (This may be required if your account has two-factor authentication enabled, and may be preferable to storing your account password in plain text even if two-factor authentication is disabled.)

You can use the Git credential store to authenticate with a GitHub PAT (Personal Access Token) like so:

```
echo "https://x-access-token:$GH_TOKEN@github.com" >> ~/.git-credentials
```

If you don't want to store any credentials on the file system, you can store them in memory temporarily using git-credential-cache instead.

If you setup fetching via SSH, you can use Git URL rewrite feature. The following command instructs Git to use SSH URLs for GitHub instead of HTTPS ones:

```
git config --global url."git@github.com:".insteadOf "https://github.com/"
```

Fetching via SSH If your SSH key has no password, fetching should just work. If it does have a password, you can avoid entering it manually every time using ssh-agent.

On GitHub, see Connecting to GitHub with SSH for details on configuration and key creation.

Project locations

Projects can be located anywhere inside the workspace, but they may not "escape" it.

In other words, project repositories need not be located in subdirectories of the manifest repository or as immediate subdirectories of the topdir. However, projects must have paths inside the workspace.

You may replace a project's repository directory within the workspace with a symbolic link to elsewhere on your computer, but west will not do this for you.

Topologies supported

The following are example source code topologies supported by west.

- T1: star topology, zephyr is the manifest repository
- T2: star topology, a Zephyr application is the manifest repository
- T3: forest topology, freestanding manifest repository

T1: Star topology, zephyr is the manifest repository

- The zephyr repository acts as the central repository and specifies its *Modules (External projects)* in its `west.yml`
- Analogy with existing mechanisms: Git submodules with zephyr as the super-project

This is the default. See *Workspace concepts* for how mainline Zephyr is an example of this topology.

T2: Star topology, application is the manifest repository

- Useful for those focused on a single application
- A repository containing a Zephyr application acts as the central repository and names other projects required to build it in its `west.yml`. This includes the zephyr repository and any modules.
- Analogy with existing mechanisms: Git submodules with the application as the super-project, zephyr and other projects as submodules

A workspace using this topology looks like this:

```
west-workspace/
|
├── application/         # .git/     |
|   ├── CMakeLists.txt                |
|   ├── prj.conf                      |  never modified by west
|   ├── src/                          |
|   |   └── main.c                    |
|   └── west.yml         # main manifest with optional import(s) and override(s)
|                                     |
├── modules/
|   └── lib/
|       └── zcbor/       # .git/ project from either the main manifest or some import.
|
└── zephyr/              # .git/ project
    └── west.yml         # This can be partially imported with lower precedence or ignored.
                         # Only the 'manifest-rev' version can be imported.
```

Here is an example `application/west.yml` which uses *Manifest Imports*, available since west 0.7, to import Zephyr v2.5.0 and its modules into the application manifest file:

```
# Example T2 west.yml, using manifest imports.
manifest:
  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos
  projects:
    - name: zephyr
      remote: zephyrproject-rtos
      revision: v2.5.0
      import: true
  self:
    path: application
```

You can still selectively "override" individual Zephyr modules if you use `import:` in this way; see *Example 1.3: Downstream of a Zephyr release, with module fork* for an example.

Another way to do the same thing is to copy/paste `zephyr/west.yml` to `application/west.yml`, adding an entry for the zephyr project itself, like this:

```
# Equivalent to the above, but with manually maintained Zephyr modules.
manifest:
  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos
  defaults:
    remote: zephyrproject-rtos
  projects:
    - name: zephyr
      revision: v2.5.0
      west-commands: scripts/west-commands.yml
    - name: net-tools
      revision: some-sha-goes-here
      path: tools/net-tools
    # ... other Zephyr modules go here ...
  self:
    path: application
```

(The `west-commands` is there for *Building, Flashing and Debugging* and other Zephyr-specific *Extensions*. It's not necessary when using `import`.)

The main advantage to using `import` is not having to track the revisions of imported projects separately. In the above example, using `import` means Zephyr's *module* versions are automatically determined from the `zephyr/west.yml` revision, instead of having to be copy/pasted (and

maintained) on their own.

T3: Forest topology

- Useful for those supporting multiple independent applications or downstream distributions with no "central" repository
- A dedicated manifest repository which contains no Zephyr source code, and specifies a list of projects all at the same "level"
- Analogy with existing mechanisms: Google repo-based source distribution

A workspace using this topology looks like this:

```
west-workspace/
├── app1/               # .git/ project
│   ├── CMakeLists.txt
│   ├── prj.conf
│   └── src/
│       └── main.c
├── app2/               # .git/ project
│   ├── CMakeLists.txt
│   ├── prj.conf
│   └── src/
│       └── main.c
├── manifest-repo/      # .git/ never modified by west
│   └── west.yml        # main manifest with optional import(s) and override(s)
├── modules/
│   └── lib/
│       └── zcbor/      # .git/ project from either the main manifest or
│                       #       from some import
│
└── zephyr/             # .git/ project
    └── west.yml        # This can be partially imported with lower precedence or ignored.
                        # Only the 'manifest-rev' version can be imported.
```

Here is an example T3 `manifest-repo/west.yml` which uses *Manifest Imports*, available since west 0.7, to import Zephyr v2.5.0 and its modules, then add the `app1` and `app2` projects:

```
manifest:
  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos
    - name: your-git-server
      url-base: https://git.example.com/your-company
  defaults:
    remote: your-git-server
  projects:
    - name: zephyr
      remote: zephyrproject-rtos
      revision: v2.5.0
      import: true
    - name: app1
      revision: SOME_SHA_OR_BRANCH_OR_TAG
    - name: app2
      revision: ANOTHER_SHA_OR_BRANCH_OR_TAG
  self:
    path: manifest-repo
```

You can also do this "by hand" by copy/pasting `zephyr/west.yml` as shown *above* for the T2 topology, with the same caveats.

2.11.7 West Manifests

This page contains detailed information about west's multiple repository model, manifest files, and the `west manifest` command. For API documentation on the `west.manifest` module, see west-apis-manifest. For a more general introduction and command overview, see *Basics*.

Multiple Repository Model

West's view of the repositories in a *west workspace*, and their history, looks like the following figure (though some parts of this example are specific to upstream Zephyr's use of west):

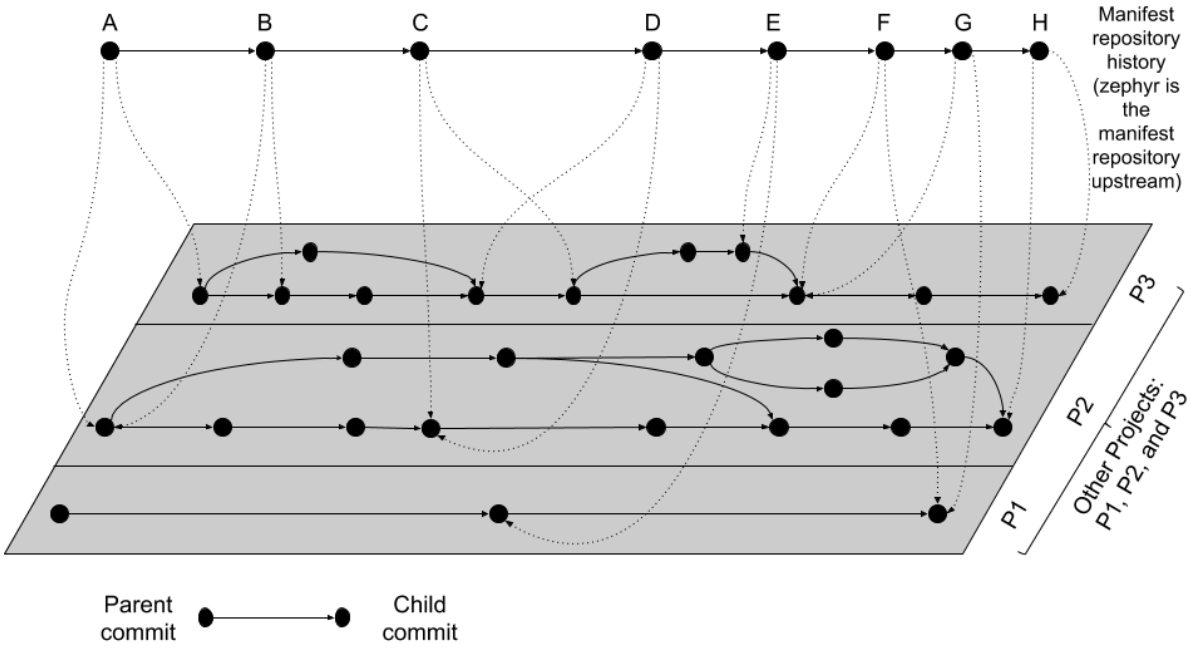


Fig. 3: West multi-repo history

The history of the manifest repository is the line of Git commits which is "floating" on top of the gray plane. Parent commits point to child commits using solid arrows. The plane below contains the Git commit history of the repositories in the workspace, with each project repository boxed in by a rectangle. Parent/child commit relationships in each repository are also shown with solid arrows.

The commits in the manifest repository (again, for upstream Zephyr this is the zephyr repository itself) each have a manifest file. The manifest file in each commit specifies the corresponding commits which it expects in each of the project repositories. This relationship is shown using dotted line arrows in the diagram. Each dotted line arrow points from a commit in the manifest repository to a corresponding commit in a project repository.

Notice the following important details:

- Projects can be added (like `P1` between manifest repository commits `D` and `E`) and removed (`P2` between the same manifest repository commits)
- Project and manifest repository histories don't have to move forwards or backwards together:
  - `P2` stays the same from `A` → `B`, as do `P1` and `P3` from `F` → `G`.
  - `P3` moves forward from `A` → `B`.
  - `P3` moves backward from `C` → `D`.

One use for moving backward in project history is to "revert" a regression by going back to a revision before it was introduced.

- Project repository commits can be "skipped": `P3` moves forward multiple commits in its history from `B → C`.
- In the above diagram, no project repository has two revisions "at the same time": every manifest file refers to exactly one commit in the projects it cares about. This can be relaxed by using a branch name as a manifest revision, at the cost of being able to bisect manifest repository history.

Manifest Files

West manifests are YAML files. Manifests have a top-level `manifest` section with some subsections, like this:

```
manifest:
  remotes:
    # short names for project URLs
  projects:
    # a list of projects managed by west
  defaults:
    # default project attributes
  self:
    # configuration related to the manifest repository itself,
    # i.e. the repository containing west.yml
  version: "<schema-version>"
  group-filter:
    # a list of project groups to enable or disable
```

In YAML terms, the manifest file contains a mapping, with a `manifest` key. Any other keys and their contents are ignored (west v0.5 also required a `west` key, but this is ignored starting with v0.6).

The manifest contains subsections, like `defaults`, `remotes`, `projects`, and `self`. In YAML terms, the value of the `manifest` key is also a mapping, with these "subsections" as keys. As of west v0.10, all of these "subsection" keys are optional.

The `projects` value is a list of repositories managed by west and associated metadata. We'll discuss it soon, but first we will describe the `remotes` section, which can be used to save typing in the `projects` list.

Remotes The `remotes` subsection contains a sequence which specifies the base URLs where projects can be fetched from.

Each `remotes` element has a name and a "URL base". These are used to form the complete Git fetch URL for each project. A project's fetch URL can be set by appending a project-specific path onto a remote URL base. (As we'll see below, projects can also specify their complete fetch URLs.)

For example:

```
manifest:
  # ...
  remotes:
    - name: remote1
      url-base: https://git.example.com/base1
    - name: remote2
      url-base: https://git.example.com/base2
```

The `remotes` keys and their usage are in the following table.

Table 2: remotes keys

Key	Description
name	Mandatory; a unique name for the remote.
url-base	A prefix that is prepended to the fetch URL for each project with this remote.

Above, two remotes are given, with names remote1 and remote2. Their URL bases are respectively https://git.example.com/base1 and https://git.example.com/base2. You can use SSH URL bases as well; for example, you might use git@example.com:base1 if remote1 supported Git over SSH as well. Anything acceptable to Git will work.

Projects The projects subsection contains a sequence describing the project repositories in the west workspace. Every project has a unique name. You can specify what Git remote URLs to use when cloning and fetching the projects, what revisions to track, and where the project should be stored on the local file system. Note that west projects *are different from modules*.

Here is an example. We'll assume the remotes given above.

```
manifest:
  # [... same remotes as above...]
  projects:
    - name: proj1
      description: the first example project
      remote: remote1
      path: extra/project-1
    - name: proj2
      description: |
        A multi-line description of the second example
        project.
      repo-path: my-path
      remote: remote2
      revision: v1.3
    - name: proj3
      url: https://github.com/user/project-three
      revision: abcde413a111
```

In this manifest:

- proj1 has remote remote1, so its Git fetch URL is https://git.example.com/base1/proj1. The remote url-base is appended with a / and the project name to form the URL.

  Locally, this project will be cloned at path extra/project-1 relative to the west workspace's root directory, since it has an explicit path attribute with this value.

  Since the project has no revision specified, master is used by default. The current tip of this branch will be fetched and checked out as a detached HEAD when west next updates this project.

- proj2 has a remote and a repo-path, so its fetch URL is https://git.example.com/base2/my-path. The repo-path attribute, if present, overrides the default name when forming the fetch URL.

  Since the project has no path attribute, its name is used by default. It will be cloned into a directory named proj2. The commit pointed to by the v1.3 tag will be checked out when west updates the project.

- proj3 has an explicit url, so it will be fetched from https://github.com/user/project-three.

  Its local path defaults to its name, proj3. Commit abcde413a111 will be checked out when it is next updated.

The available project keys and their usage are in the following table. Sometimes we'll refer to the `defaults` subsection; it will be described next.

Table 3: projects elements keys

Key(s)	Description
`name`	Mandatory; a unique name for the project. The name cannot be one of the reserved values "west" or "manifest". The name must be unique in the manifest file.
`description`	Optional, an informational description of the project. Added in west v1.2.0.
`remote`, `url`	Mandatory (one of the two, but not both). If the project has a `remote`, that remote's `url-base` will be combined with the project's `name` (or `repo-path`, if it has one) to form the fetch URL instead. If the project has a `url`, that's the complete fetch URL for the remote Git repository. If the project has neither, the `defaults` section must specify a `remote`, which will be used as the project's remote. Otherwise, the manifest is invalid.
`repo-path`	Optional. If given, this is concatenated on to the remote's `url-base` instead of the project's `name` to form its fetch URL. Projects may not have both `url` and `repo-path` attributes.
`revision`	Optional. The Git revision that `west update` should check out. This will be checked out as a detached HEAD by default, to avoid conflicting with local branch names. If not given, the `revision` value from the `defaults` subsection will be used if present. A project revision can be a branch, tag, or SHA. The default `revision` is `master` if not otherwise specified. Using `HEAD~0`[1] as the `revision` will cause west to keep the current state of the project.
`path`	Optional. Relative path specifying where to clone the repository locally, relative to the top directory in the west workspace. If missing, the project's `name` is used as a directory name.
`clone-depth`	Optional. If given, a positive integer which creates a shallow history in the cloned repository limited to the given number of commits. This can only be used if the `revision` is a branch or tag.
`west-commands`	Optional. If given, a relative path to a YAML file within the project which describes additional west commands provided by that project. This file is named `west-commands.yml` by convention. See *Extensions* for details.
`import`	Optional. If `true`, imports projects from manifest files in the given repository into the current manifest. See *Manifest Imports* for details.
`groups`	Optional, a list of groups the project belongs to. See *Project Groups* for details.
`submodules`	Optional. You can use this to make `west update` also update Git submodules defined by the project. See *Git Submodules in Projects* for details.
`userdata`	Optional. The value is an arbitrary YAML value. See *Repository user data*.

Defaults The `defaults` subsection can provide default values for project attributes. In particular, the default remote name and revision can be specified here. Another way to write the same manifest we have been describing so far using `defaults` is:

```
manifest:
  defaults:
    remote: remote1
    revision: v1.3
```

(continues on next page)

[1] In git, HEAD is a reference, whereas HEAD~<n> is a valid revision but not a reference. West fetches references, such as refs/heads/main or HEAD, and commits not available locally, but will not fetch commits if they are already available. HEAD~0 is resolved to a specific commit that is locally available, and therefore west will simply checkout the locally available commit, identified by HEAD~0.

(continued from previous page)

```
remotes:
  - name: remote1
    url-base: https://git.example.com/base1
  - name: remote2
    url-base: https://git.example.com/base2

projects:
  - name: proj1
    description: the first example project
    path: extra/project-1
    revision: master
  - name: proj2
    description: |
      A multi-line description of the second example
      project.
    repo-path: my-path
    remote: remote2
  - name: proj3
    url: https://github.com/user/project-three
    revision: abcde413a111
```

The available `defaults` keys and their usage are in the following table.

Table 4: defaults keys

Key	Description
`remote`	Optional. This will be used for a project's `remote` if it does not have a `url` or `remote` key set.
`revision`	Optional. This will be used for a project's `revision` if it does not have one set. If not given, the default is `master`.

Self The `self` subsection can be used to control the manifest repository itself.

As an example, let's consider this snippet from the zephyr repository's `west.yml`:

```
manifest:
  # ...
  self:
    path: zephyr
    west-commands: scripts/west-commands.yml
```

This ensures that the zephyr repository is cloned into path `zephyr`, though as explained above that would have happened anyway if cloning from the default manifest URL, `https://github.com/zephyrproject-rtos/zephyr`. Since the zephyr repository does contain extension commands, its `self` entry declares the location of the corresponding `west-commands.yml` relative to the repository root.

The available `self` keys and their usage are in the following table.

Table 5: self keys

Key	Description
path	Optional. The path `west init` should clone the manifest repository into, relative to the west workspace topdir. If not given, the basename of the path component in the manifest repository URL will be used by default. For example, if the URL is `https://git.example.com/project-repo`, the manifest repository would be cloned to the directory `project-repo`.
west-commands	Optional. This is analogous to the same key in a project sequence element.
import	Optional. This is also analogous to the `projects` key, but allows importing projects from other files in the manifest repository. See *Manifest Imports*.

Version The `version` subsection declares that the manifest file uses features which were introduced in some version of west. Attempts to load the manifest with older versions of west will fail with an error message that explains the minimum required version of west which is needed.

Here is an example:

```
manifest:
  # Marks that this file uses version 0.10 of the west manifest
  # file format.
  #
  # An attempt to load this manifest file with west v0.8.0 will
  # fail with an error message saying that west v0.10.0 or
  # later is required.
  version: "0.10"
```

The pykwalify schema `manifest-schema.yml` in the west source code repository is used to validate the manifest section.

Here is a table with the valid `version` values, along with information about the manifest file features that were introduced in that version.

version	New features
"0.7"	Initial support for the version feature. All manifest file features that are not otherwise mentioned in this table were introduced in west v0.7.0 or earlier.
"0.8"	Support for import: path-prefix: (*Option 3: Mapping*)
"0.9"	**Use of west v0.9.x is discouraged.** This schema version is provided to allow users to explicitly request compatibility with west *v0.9.0*. However, west *v0.10.0* and later have incompatible behavior for features that were introduced in west v0.9.0. You should ignore version "0.9" if possible.
"0.10"	Support for: • submodules: in projects: (*Git Submodules in Projects*) • manifest: group-filter:, and groups: in projects: (*Project Groups*) • The import: feature now supports allowlist: and blocklist:; these are respectively recommended as replacements for older names as part of a general Zephyr-wide inclusive language change. The older key names are still supported for backwards compatibility. (*Manifest Imports*, *Option 3: Mapping*)
"0.12"	Support for userdata: in projects: (*Repository user data*)
"0.13"	Support for self: userdata: (*Repository user data*)
"1.0"	Identical to "0.13", but available for use by users that do not wish to use a "0.x" version field.
"1.2"	Support for description: in projects: (*Projects*)

Note

Versions of west without any new features in the manifest file format do not change the list of valid version values. For example, version: "0.11" is **not** valid, because west v0.11.x did not introduce new manifest file format features.

Quoting the version value as shown above forces the YAML parser to treat it as a string. Without quotes, 0.10 in YAML is just the floating point value 0.1. You can omit the quotes if the value is the same when cast to string, but it's best to include them. Always use quotes if you're not sure.

If you do not include a version in your manifest, each new release of west assumes that it should try to load it using the features that were available in that release. This may result in error messages that are harder to understand if that version of west is too old to load the manifest.

Group-filter See *Project Groups*.

Active and Inactive Projects

Projects defined in the west manifest can be *inactive* or *active*. The difference is that an inactive project is generally ignored by west. For example, west update will not update inactive projects, and west list will not print information about them by default. As another example, any *Manifest Imports* in an inactive project will be ignored by west.

There are two ways to make a project inactive:

1. Using the manifest.project-filter configuration option. If a project is made active or inactive using this option, then the rules related to making a project inactive using its

`groups:` are ignored. That is, if a regular expression in `manifest.project-filter` applies to a project, the project's groups have no effect on whether it is active or inactive.

See the entry for this option in *Built-in Configuration Options* for details.

2. Otherwise, if a project has groups, and they are all disabled, then the project is inactive.

   See the following section for details.

Project Groups

You can use the `groups` and `group-filter` keys briefly described *above* to place projects into groups, and to enable or disable groups.

For example, this lets you run a `west forall` command only on the projects in the group by using `west forall --group`. This can also let you make projects inactive; see the previous section for more information on inactive projects.

The next section introduces project groups. The following section describes *Enabled and Disabled Project Groups*. There are some basic examples in *Project Group Examples*. Finally, *Group Filters and Imports* provides a simplified overview of how `group-filter` interacts with the *Manifest Imports* feature.

Groups Basics The `groups:` and `group-filter:` keys appear in the manifest like this:

```
manifest:
  projects:
    - name: some-project
      groups: ...
  group-filter: ...
```

The `groups` key's value is a list of group names. Group names are strings.

You can enable or disable project groups using `group-filter`. Projects whose groups are all disabled, and which are not otherwise made active by a `manifest.project-filter` configuration option, are inactive.

For example, in this manifest fragment:

```
manifest:
  projects:
    - name: project-1
      groups:
        - groupA
    - name: project-2
      groups:
        - groupB
        - groupC
    - name: project-3
```

The projects are in these groups:

- `project-1`: one group, named `groupA`
- `project-2`: two groups, named `groupB` and `groupC`
- `project-3`: no groups

Project group names must not contain commas (,), colons (:), or whitespace.

Group names must not begin with a dash (-) or the plus sign (+), but they may contain these characters elsewhere in their names. For example, `foo-bar` and `foo+bar` are valid groups, but `-foobar` and `+foobar` are not.

Group names are otherwise arbitrary strings. Group names are case sensitive.

As a restriction, no project may use both `import:` and `groups:`. (This is necessary to avoid some pathological edge cases.)

Enabled and Disabled Project Groups All project groups are enabled by default. You can enable or disable groups in both your manifest file and *Configuration*.

Within a manifest file, `manifest: group-filter:` is a YAML list of groups to enable and disable.

To enable a group, prefix its name with a plus sign (+). For example, `groupA` is enabled in this manifest fragment:

```
manifest:
  group-filter: [+groupA]
```

Although this is redundant for groups that are already enabled by default, it can be used to override settings in an imported manifest file. See *Group Filters and Imports* for more information.

To disable a group, prefix its name with a dash (-). For example, `groupA` and `groupB` are disabled in this manifest fragment:

```
manifest:
  group-filter: [-groupA,-groupB]
```

Note

Since `group-filter` is a YAML list, you could have written this fragment as follows:

```
manifest:
  group-filter:
    - -groupA
    - -groupB
```

However, this syntax is harder to read and therefore discouraged.

In addition to the manifest file, you can control which groups are enabled and disabled using the `manifest.group-filter` configuration option. This option is a comma-separated list of groups to enable and/or disable.

To enable a group, add its name to the list prefixed with `+`. To disable a group, add its name prefixed with `-`. For example, setting `manifest.group-filter` to `+groupA,-groupB` enables `groupA`, and disables `groupB`.

The value of the configuration option overrides any data in the manifest file. You can think of this as if the `manifest.group-filter` configuration option is appended to the `manifest: group-filter:` list from YAML, with "last entry wins" semantics.

Project Group Examples This section contains example situations involving project groups and active projects. The examples use both `manifest: group-filter:` YAML lists and `manifest.group-filter` configuration lists, to show how they work together.

Note that the `defaults` and `remotes` data in the following manifests isn't relevant except to make the examples complete and self-contained.

Note

In all of the examples that follow, the `manifest.project-filter` option is assumed to be unset.

Example 1: no disabled groups The entire manifest file is:

```
manifest:
  projects:
    - name: foo
      groups:
        - groupA
    - name: bar
      groups:
        - groupA
        - groupB
    - name: baz

  defaults:
    remote: example-remote
  remotes:
    - name: example-remote
      url-base: https://git.example.com
```

The `manifest.group-filter` configuration option is not set (you can ensure this by running `west config -D manifest.group-filter`).

No groups are disabled, because all groups are enabled by default. Therefore, all three projects (`foo`, `bar`, and `baz`) are active. Note that there is no way to make project `baz` inactive, since it has no groups.

Example 2: Disabling one group via manifest The entire manifest file is:

```
manifest:
  projects:
    - name: foo
      groups:
        - groupA
    - name: bar
      groups:
        - groupA
        - groupB

  group-filter: [-groupA]

  defaults:
    remote: example-remote
  remotes:
    - name: example-remote
      url-base: https://git.example.com
```

The `manifest.group-filter` configuration option is not set (you can ensure this by running `west config -D manifest.group-filter`).

Since `groupA` is disabled, project `foo` is inactive. Project `bar` is active, because `groupB` is enabled.

Example 3: Disabling multiple groups via manifest The entire manifest file is:

```
manifest:
  projects:
    - name: foo
      groups:
        - groupA
    - name: bar
      groups:
        - groupA
```

(continues on next page)

(continued from previous page)

```
      - groupB

  group-filter: [-groupA,-groupB]

  defaults:
    remote: example-remote
  remotes:
    - name: example-remote
      url-base: https://git.example.com
```

The `manifest.group-filter` configuration option is not set (you can ensure this by running `west config -D manifest.group-filter`).

Both `foo` and `bar` are inactive, because all of their groups are disabled.

Example 4: Disabling a group via configuration The entire manifest file is:

```
manifest:
  projects:
    - name: foo
      groups:
        - groupA
    - name: bar
      groups:
        - groupA
        - groupB

  defaults:
    remote: example-remote
  remotes:
    - name: example-remote
      url-base: https://git.example.com
```

The `manifest.group-filter` configuration option is set to `-groupA` (you can ensure this by running `west config manifest.group-filter -- -groupA`; the extra `--` is required so the argument parser does not treat `-groupA` as a command line option `-g` with value `roupA`).

Project `foo` is inactive because `groupA` has been disabled by the `manifest.group-filter` configuration option. Project `bar` is active because `groupB` is enabled.

Example 5: Overriding a disabled group via configuration The entire manifest file is:

```
manifest:
  projects:
    - name: foo
    - name: bar
      groups:
        - groupA
    - name: baz
      groups:
        - groupA
        - groupB

  group-filter: [-groupA]

  defaults:
    remote: example-remote
  remotes:
    - name: example-remote
      url-base: https://git.example.com
```

The `manifest.group-filter` configuration option is set to `+groupA` (you can ensure this by running `west config manifest.group-filter +groupA`).

In this case, `groupA` is enabled: the `manifest.group-filter` configuration option has higher precedence than the `manifest: group-filter: [-groupA]` content in the manifest file.

Therefore, projects `foo` and `bar` are both active.

Example 6: Overriding multiple disabled groups via configuration The entire manifest file is:

```
manifest:
  projects:
    - name: foo
    - name: bar
      groups:
        - groupA
    - name: baz
      groups:
        - groupA
        - groupB

  group-filter: [-groupA,-groupB]

  defaults:
    remote: example-remote
  remotes:
    - name: example-remote
      url-base: https://git.example.com
```

The `manifest.group-filter` configuration option is set to `+groupA,+groupB` (you can ensure this by running `west config manifest.group-filter "+groupA,+groupB"`).

In this case, both `groupA` and `groupB` are enabled, because the configuration value overrides the manifest file for both groups.

Therefore, projects `foo` and `bar` are both active.

Example 7: Disabling multiple groups via configuration The entire manifest file is:

```
manifest:
  projects:
    - name: foo
    - name: bar
      groups:
        - groupA
    - name: baz
      groups:
        - groupA
        - groupB

  defaults:
    remote: example-remote
  remotes:
    - name: example-remote
      url-base: https://git.example.com
```

The `manifest.group-filter` configuration option is set to `-groupA,-groupB` (you can ensure this by running `west config manifest.group-filter -- "-groupA,-groupB"`).

In this case, both `groupA` and `groupB` are disabled.

Therefore, projects `foo` and `bar` are both inactive.

Group Filters and Imports This section provides a simplified description of how the `manifest: group-filter:` value behaves when combined with *Manifest Imports*. For complete details, see *Manifest Import Details*.

Warning

The below semantics apply to west v0.10.0 and later. West v0.9.x semantics are different, and combining `group-filter` with `import` in west v0.9.x is discouraged.

In short:

- if you only import one manifest, any groups it disables in its `group-filter` are also disabled in your manifest
- you can override this in your manifest file's `manifest: group-filter:` value, your workspace's `manifest.group-filter` configuration option, or both

Here are some examples.

Example 1: no overrides You are using this `parent/west.yml` manifest:

```
# parent/west.yml:
manifest:
  projects:
    - name: child
      url: https://git.example.com/child
      import: true
    - name: project-1
      url: https://git.example.com/project-1
      groups:
        - unstable
```

And `child/west.yml` contains:

```
# child/west.yml:
manifest:
  group-filter: [-unstable]
  projects:
    - name: project-2
      url: https://git.example.com/project-2
    - name: project-3
      url: https://git.example.com/project-3
      groups:
        - unstable
```

Only `child` and `project-2` are active in the resolved manifest.

The `unstable` group is disabled in `child/west.yml`, and that is not overridden in `parent/west.yml`. Therefore, the final `group-filter` for the resolved manifest is `[-unstable]`.

Since `project-1` and `project-3` are in the `unstable` group and are not in any other group, they are inactive.

Example 2: overriding an imported `group-filter` via manifest You are using this `parent/west.yml` manifest:

```
# parent/west.yml:
manifest:
  group-filter: [+unstable,-optional]
  projects:
```

(continues on next page)

(continued from previous page)

```
    - name: child
      url: https://git.example.com/child
      import: true
    - name: project-1
      url: https://git.example.com/project-1
      groups:
        - unstable
```

And `child/west.yml` contains:

```
# child/west.yml:
manifest:
  group-filter: [-unstable]
  projects:
    - name: project-2
      url: https://git.example.com/project-2
      groups:
        - optional
    - name: project-3
      url: https://git.example.com/project-3
      groups:
        - unstable
```

Only the `child`, `project-1`, and `project-3` projects are active.

The `[-unstable]` group filter in `child/west.yml` is overridden in `parent/west.yml`, so the `unstable` group is enabled. Since `project-1` and `project-3` are in the `unstable` group, they are active.

The same `parent/west.yml` file disables the `optional` group, so `project-2` is inactive.

The final group filter specified by `parent/west.yml` is `[+unstable,-optional]`.

Example 3: overriding an imported `group-filter` via configuration You are using this `parent/west.yml` manifest:

```
# parent/west.yml:
manifest:
  projects:
    - name: child
      url: https://git.example.com/child
      import: true
    - name: project-1
      url: https://git.example.com/project-1
      groups:
        - unstable
```

And `child/west.yml` contains:

```
# child/west.yml:
manifest:
  group-filter: [-unstable]
  projects:
    - name: project-2
      url: https://git.example.com/project-2
      groups:
        - optional
    - name: project-3
      url: https://git.example.com/project-3
      groups:
        - unstable
```

If you run:

```
west config manifest.group-filter +unstable,-optional
```

Then only the child, project-1, and project-3 projects are active.

The -unstable group filter in child/west.yml is overridden in the manifest.group-filter configuration option, so the unstable group is enabled. Since project-1 and project-3 are in the unstable group, they are active.

The same configuration option disables the optional group, so project-2 is inactive.

The final group filter specified by parent/west.yml and the manifest.group-filter configuration option is [+unstable,-optional].

Git Submodules in Projects

You can use the submodules keys briefly described *above* to force west update to also handle any Git submodules configured in project's git repository. The submodules key can appear inside projects, like this:

```
manifest:
  projects:
    - name: some-project
      submodules: ...
```

The submodules key can be a boolean or a list of mappings. We'll describe these in order.

Option 1: Boolean This is the easiest way to use submodules.

If submodules is true as a projects attribute, west update will recursively update the project's Git submodules whenever it updates the project itself. If it's false or missing, it has no effect.

For example, let's say you have a source code repository foo, which has some submodules, and you want west update to keep all of them in sync, along with another project named bar in the same workspace.

You can do that with this manifest file:

```
manifest:
  projects:
    - name: foo
      submodules: true
    - name: bar
```

Here, west update will initialize and update all submodules in foo. If bar has any submodules, they are ignored, because bar does not have a submodules value.

Option 2: List of mappings The submodules key may be a list of mappings, one list element for each desired submodule. Each submodule listed is updated recursively. You can still track and update unlisted submodules with git commands manually; present or not they will be completely ignored by west.

The path key must match exactly the path of one submodule relative to its parent west project, as shown in the output of git submodule status. The name key is optional and not used by west for now; it's not passed to git submodule commands either. The name key was briefly mandatory in west version 0.9.0, but was made optional in 0.9.1.

For example, let's say you have a source code repository foo, which has many submodules, and you want west update to keep some but not all of them in sync, along with another project named bar in the same workspace.

You can do that with this manifest file:

```
manifest:
  projects:
    - name: foo
      submodules:
        - path: path/to/foo-first-sub
        - name: foo-second-sub
          path: path/to/foo-second-sub
    - name: bar
```

Here, `west update` will recursively initialize and update just the submodules in `foo` with paths `path/to/foo-first-sub` and `path/to/foo-second-sub`. Any submodules in `bar` are still ignored.

Repository user data

West versions v0.12 and later support an optional `userdata` key in projects.

West versions v0.13 and later supports this key in the `manifest: self:` section.

It is meant for consumption by programs that require user-specific project metadata. Beyond parsing it as YAML, west itself ignores the value completely.

The key's value is arbitrary YAML. West parses the value and makes it accessible to programs using west-apis as the `userdata` attribute of the corresponding `west.manifest.Project` object.

Example manifest fragment:

```
manifest:
  projects:
    - name: foo
    - name: bar
      userdata: a-string
    - name: baz
      userdata:
        key: value
  self:
    userdata: blub
```

Example Python usage:

```
manifest = west.manifest.Manifest.from_file()

foo, bar, baz = manifest.get_projects(['foo', 'bar', 'baz'])

foo.userdata # None
bar.userdata # 'a-string'
baz.userdata # {'key': 'value'}
manifest.userdata # 'blub'
```

Manifest Imports

You can use the `import` key briefly described above to include projects from other manifest files in your `west.yml`. This key can be either a `project` or `self` section attribute:

```
manifest:
  projects:
    - name: some-project
      import: ...
  self:
    import: ...
```

You can use a "self: import:" to load additional files from the repository containing your `west.yml`. You can use a "project: ... import:" to load additional files defined in that project's Git history.

West resolves the final manifest from individual manifest files in this order:

1. imported files in `self`
2. your `west.yml` file
3. imported files in `projects`

During resolution, west ignores projects which have already been defined in other files. For example, a project named `foo` in your `west.yml` makes west ignore other projects named `foo` imported from your `projects` list.

The `import` key can be a boolean, path, mapping, or sequence. We'll describe these in order, using examples:

- ***Boolean***
  - *Example 1.1: Downstream of a Zephyr release*
  - *Example 1.2: "Rolling release" Zephyr downstream*
  - *Example 1.3: Downstream of a Zephyr release, with module fork*
- ***Relative path***
  - *Example 2.1: Downstream of a Zephyr release with explicit path*
  - *Example 2.2: Downstream with directory of manifest files*
  - *Example 2.3: Continuous Integration overrides*
- ***Mapping with additional configuration***
  - *Example 3.1: Downstream with name allowlist*
  - *Example 3.2: Downstream with path allowlist*
  - *Example 3.3: Downstream with path blocklist*
  - *Example 3.4: Import into a subdirectory*
- ***Sequence of paths and mappings***
  - *Example 4.1: Downstream with sequence of manifest files*
  - *Example 4.2: Import order illustration*

A more *formal description* of how this works is last, after the examples.

Troubleshooting Note If you're using this feature and find west's behavior confusing, try *resolving your manifest* to see the final results after imports are done.

Option 1: Boolean This is the easiest way to use `import`.

If `import` is `true` as a `projects` attribute, west imports projects from the `west.yml` file in that project's root directory. If it's `false` or missing, it has no effect. For example, this manifest would import `west.yml` from the `p1` git repository at revision `v1.0`:

```
manifest:
  # ...
  projects:
    - name: p1
      revision: v1.0
      import: true    # Import west.yml from p1's v1.0 git tag
    - name: p2
```

(continues on next page)

(continued from previous page)

```
      import: false   # Nothing is imported from p2.
    - name: p3        # Nothing is imported from p3 either.
```

It's an error to set `import` to either `true` or `false` inside `self`, like this:

```
manifest:
  # ...
  self:
    import: true  # Error
```

Example 1.1: Downstream of a Zephyr release You have a source code repository you want to use with Zephyr v1.14.1 LTS. You want to maintain the whole thing using west. You don't want to modify any of the mainline repositories.

In other words, the west workspace you want looks like this:

```
my-downstream/
├── .west/                    # west directory
├── zephyr/                   # mainline zephyr repository
│   └── west.yml              # the v1.14.1 version of this file is imported
├── modules/                  # modules from mainline zephyr
│   ├── hal/
│   └── [...other directories..]
├── [ ... other projects ...]  # other mainline repositories
└── my-repo/                  # your downstream repository
    ├── west.yml              # main manifest importing zephyr/west.yml v1.14.1
    └── [...other files..]
```

You can do this with the following `my-repo/west.yml`:

```
# my-repo/west.yml:
manifest:
  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos
  projects:
    - name: zephyr
      remote: zephyrproject-rtos
      revision: v1.14.1
      import: true
```

You can then create the workspace on your computer like this, assuming `my-repo` is hosted at `https://git.example.com/my-repo`:

```
west init -m https://git.example.com/my-repo my-downstream
cd my-downstream
west update
```

After `west init`, `my-downstream/my-repo` will be cloned.

After `west update`, all of the projects defined in the `zephyr` repository's `west.yml` at revision `v1.14.1` will be cloned into `my-downstream` as well.

You can add and commit any code to `my-repo` you please at this point, including your own Zephyr applications, drivers, etc. See *Application Development*.

Example 1.2: "Rolling release" Zephyr downstream This is similar to *Example 1.1: Downstream of a Zephyr release*, except we'll use `revision: main` for the zephyr repository:

```
# my-repo/west.yml:
manifest:
  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos
  projects:
    - name: zephyr
      remote: zephyrproject-rtos
      revision: main
      import: true
```

You can create the workspace in the same way:

```
west init -m https://git.example.com/my-repo my-downstream
cd my-downstream
west update
```

This time, whenever you run `west update`, the special *manifest-rev* branch in the `zephyr` repository will be updated to point at a newly fetched `main` branch tip from the URL https://github.com/zephyrproject-rtos/zephyr.

The contents of `zephyr/west.yml` at the new `manifest-rev` will then be used to import projects from Zephyr. This lets you stay up to date with the latest changes in the Zephyr project. The cost is that running `west update` will not produce reproducible results, since the remote `main` branch can change every time you run it.

It's also important to understand that west **ignores your working tree's** `zephyr/west.yml` entirely when resolving imports. West always uses the contents of imported manifests as they were committed to the latest `manifest-rev` when importing from a project.

You can only import manifest from the file system if they are in your manifest repository's working tree. See *Example 2.2: Downstream with directory of manifest files* for an example.

Example 1.3: Downstream of a Zephyr release, with module fork This manifest is similar to the one in *Example 1.1: Downstream of a Zephyr release*, except it:

- is a downstream of Zephyr 2.0
- includes a downstream fork of the `modules/hal/nordic` *module* which was included in that release

```
# my-repo/west.yml:
manifest:
  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos
    - name: my-remote
      url-base: https://git.example.com
  projects:
    - name: hal_nordic         # higher precedence
      remote: my-remote
      revision: my-sha
      path: modules/hal/nordic
    - name: zephyr
      remote: zephyrproject-rtos
      revision: v2.0.0
      import: true             # imported projects have lower precedence

# subset of zephyr/west.yml contents at v2.0.0:
manifest:
  defaults:
    remote: zephyrproject-rtos
```

(continues on next page)

(continued from previous page)

```
  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos
  projects:
  # ...
  - name: hal_nordic           # lower precedence, values ignored
    path: modules/hal/nordic
    revision: another-sha
```

With this manifest file, the project named `hal_nordic`:

- is cloned from `https://git.example.com/hal_nordic` instead of `https://github.com/zephyrproject-rtos/hal_nordic`.
- is updated to commit `my-sha` by `west update`, instead of the mainline commit `another-sha`

In other words, when your top-level manifest defines a project, like `hal_nordic`, west will ignore any other definition it finds later on while resolving imports.

This does mean you have to copy the `path: modules/hal/nordic` value into `my-repo/west.yml` when defining `hal_nordic` there. The value from `zephyr/west.yml` is ignored entirely. See *Resolving Manifests* for troubleshooting advice if this gets confusing in practice.

When you run `west update`, west will:

- update zephyr's `manifest-rev` to point at the `v2.0.0` tag
- import `zephyr/west.yml` at that `manifest-rev`
- locally check out the `v2.0.0` revisions for all zephyr projects except `hal_nordic`
- update `hal_nordic` to `my-sha` instead of `another-sha`

Option 2: Relative path The `import` value can also be a relative path to a manifest file or a directory containing manifest files. The path is relative to the root directory of the `projects` or `self` repository the `import` key appears in.

Here is an example:

```
manifest:
  projects:
    - name: project-1
      revision: v1.0
      import: west.yml
    - name: project-2
      revision: main
      import: p2-manifests
  self:
    import: submanifests
```

This will import the following:

- the contents of `project-1/west.yml` at `manifest-rev`, which points at tag `v1.0` after running `west update`
- any YAML files in the directory tree `project-2/p2-manifests` at the latest commit in the `main` branch, as fetched by `west update`, sorted by file name
- YAML files in `submanifests` in your manifest repository, as they appear on your file system, sorted by file name

Notice how `projects` imports get data from Git using `manifest-rev`, while `self` imports get data from your file system. This is because as usual, west leaves version control for your manifest repository up to you.

Example 2.1: Downstream of a Zephyr release with explicit path This is an explicit way to write an equivalent manifest to the one in *Example 1.1: Downstream of a Zephyr release*.

```
manifest:
  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos
  projects:
    - name: zephyr
      remote: zephyrproject-rtos
      revision: v1.14.1
      import: west.yml
```

The setting `import: west.yml` means to use the file `west.yml` inside the `zephyr` project. This example is contrived, but shows the idea.

This can be useful in practice when the name of the manifest file you want to import is not `west.yml`.

Example 2.2: Downstream with directory of manifest files Your Zephyr downstream has a lot of additional repositories. So many, in fact, that you want to split them up into multiple manifest files, but keep track of them all in a single manifest repository, like this:

```
my-repo/
├── submanifests
│   ├── 01-libraries.yml
│   ├── 02-vendor-hals.yml
│   └── 03-applications.yml
└── west.yml
```

You want to add all the files in `my-repo/submanifests` to the main manifest file, `my-repo/west.yml`, in addition to projects in `zephyr/west.yml`. You want to track the latest development code in the Zephyr repository's `main` branch instead of using a fixed revision.

Here's how:

```
# my-repo/west.yml:
manifest:
  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos
  projects:
    - name: zephyr
      remote: zephyrproject-rtos
      revision: main
      import: true
  self:
    import: submanifests
```

Manifest files are imported in this order during resolution:

1. `my-repo/submanifests/01-libraries.yml`
2. `my-repo/submanifests/02-vendor-hals.yml`
3. `my-repo/submanifests/03-applications.yml`
4. `my-repo/west.yml`
5. `zephyr/west.yml`

Note

The `.yml` file names are prefixed with numbers in this example to make sure they are imported in the specified order.

You can pick arbitrary names. West sorts files in a directory by name before importing.

Notice how the manifests in `submanifests` are imported *before* `my-repo/west.yml` and `zephyr/west.yml`. In general, an `import` in the `self` section is processed before the manifest files in `projects` and the main manifest file.

This means projects defined in `my-repo/submanifests` take highest precedence. For example, if `01-libraries.yml` defines `hal_nordic`, the project by the same name in `zephyr/west.yml` is simply ignored. As usual, see *Resolving Manifests* for troubleshooting advice.

This may seem strange, but it allows you to redefine projects "after the fact", as we'll see in the next example.

Example 2.3: Continuous Integration overrides Your continuous integration system needs to fetch and test multiple repositories in your west workspace from a developer's forks instead of your mainline development trees, to see if the changes all work well together.

Starting with *Example 2.2: Downstream with directory of manifest files*, the CI scripts add a file `00-ci.yml` in `my-repo/submanifests`, with these contents:

```
# my-repo/submanifests/00-ci.yml:
manifest:
  projects:
    - name: a-vendor-hal
      url: https://github.com/a-developer/hal
      revision: a-pull-request-branch
    - name: an-application
      url: https://github.com/a-developer/application
      revision: another-pull-request-branch
```

The CI scripts run `west update` after generating this file in `my-repo/submanifests`. The projects defined in `00-ci.yml` have higher precedence than other definitions in `my-repo/submanifests`, because the name `00-ci.yml` comes before the other file names.

Thus, `west update` always checks out the developer's branches in the projects named `a-vendor-hal` and `an-application`, even if those same projects are also defined elsewhere.

Option 3: Mapping The `import` key can also contain a mapping with the following keys:

- `file`: Optional. The name of the manifest file or directory to import. This defaults to `west.yml` if not present.
- `name-allowlist`: Optional. If present, a name or sequence of project names to include.
- `path-allowlist`: Optional. If present, a path or sequence of project paths to match against. This is a shell-style globbing pattern, currently implemented with pathlib. Note that this means case sensitivity is platform specific.
- `name-blocklist`: Optional. Like `name-allowlist`, but contains project names to exclude rather than include.
- `path-blocklist`: Optional. Like `path-allowlist`, but contains project paths to exclude rather than include.

- `path-prefix`: Optional (new in v0.8.0). If given, this will be prepended to the project's path in the workspace, as well as the paths of any imported projects. This can be used to place these projects in a subdirectory of the workspace.

Allowlists override blocklists if both are given. For example, if a project is blocked by path, then allowed by name, it will still be imported.

Example 3.1: Downstream with name allowlist Here is a pair of manifest files, representing a mainline and a downstream. The downstream doesn't want to use all the mainline projects, however. We'll assume the mainline `west.yml` is hosted at `https://git.example.com/mainline/manifest`.

```
# mainline west.yml:
manifest:
  projects:
    - name: mainline-app                # included
      path: examples/app
      url: https://git.example.com/mainline/app
    - name: lib
      path: libraries/lib
      url: https://git.example.com/mainline/lib
    - name: lib2                        # included
      path: libraries/lib2
      url: https://git.example.com/mainline/lib2

# downstream west.yml:
manifest:
  projects:
    - name: mainline
      url: https://git.example.com/mainline/manifest
      import:
        name-allowlist:
          - mainline-app
          - lib2
    - name: downstream-app
      url: https://git.example.com/downstream/app
    - name: lib3
      path: libraries/lib3
      url: https://git.example.com/downstream/lib3
```

An equivalent manifest in a single file would be:

```
manifest:
  projects:
    - name: mainline
      url: https://git.example.com/mainline/manifest
    - name: downstream-app
      url: https://git.example.com/downstream/app
    - name: lib3
      path: libraries/lib3
      url: https://git.example.com/downstream/lib3
    - name: mainline-app                # imported
      path: examples/app
      url: https://git.example.com/mainline/app
    - name: lib2                        # imported
      path: libraries/lib2
      url: https://git.example.com/mainline/lib2
```

If an allowlist had not been used, the `lib` project from the mainline manifest would have been imported.

Example 3.2: Downstream with path allowlist Here is an example showing how to allowlist mainline's libraries only, using `path-allowlist`.

```
# mainline west.yml:
manifest:
  projects:
    - name: app
      path: examples/app
      url: https://git.example.com/mainline/app
    - name: lib
      path: libraries/lib                  # included
      url: https://git.example.com/mainline/lib
    - name: lib2
      path: libraries/lib2                 # included
      url: https://git.example.com/mainline/lib2

# downstream west.yml:
manifest:
  projects:
    - name: mainline
      url: https://git.example.com/mainline/manifest
      import:
        path-allowlist: libraries/*
    - name: app
      url: https://git.example.com/downstream/app
    - name: lib3
      path: libraries/lib3
      url: https://git.example.com/downstream/lib3
```

An equivalent manifest in a single file would be:

```
manifest:
  projects:
    - name: lib                          # imported
      path: libraries/lib
      url: https://git.example.com/mainline/lib
    - name: lib2                         # imported
      path: libraries/lib2
      url: https://git.example.com/mainline/lib2
    - name: mainline
      url: https://git.example.com/mainline/manifest
    - name: app
      url: https://git.example.com/downstream/app
    - name: lib3
      path: libraries/lib3
      url: https://git.example.com/downstream/lib3
```

Example 3.3: Downstream with path blocklist Here's an example showing how to block all vendor HALs from mainline by common path prefix in the workspace, add your own version for the chip you're targeting, and keep everything else.

```
# mainline west.yml:
manifest:
  defaults:
    remote: mainline
  remotes:
    - name: mainline
      url-base: https://git.example.com/mainline
  projects:
    - name: app
    - name: lib
```

(continues on next page)

(continued from previous page)

```
      path: libraries/lib
    - name: lib2
      path: libraries/lib2
    - name: hal_foo
      path: modules/hals/foo     # excluded
    - name: hal_bar
      path: modules/hals/bar     # excluded
    - name: hal_baz
      path: modules/hals/baz     # excluded

# downstream west.yml:
manifest:
  projects:
    - name: mainline
      url: https://git.example.com/mainline/manifest
      import:
        path-blocklist: modules/hals/*
    - name: hal_foo
      path: modules/hals/foo
      url: https://git.example.com/downstream/hal_foo
```

An equivalent manifest in a single file would be:

```
manifest:
  defaults:
    remote: mainline
  remotes:
    - name: mainline
      url-base: https://git.example.com/mainline
  projects:
    - name: app                  # imported
    - name: lib                  # imported
      path: libraries/lib
    - name: lib2                 # imported
      path: libraries/lib2
    - name: mainline
      repo-path: https://git.example.com/mainline/manifest
    - name: hal_foo
      path: modules/hals/foo
      url: https://git.example.com/downstream/hal_foo
```

Example 3.4: Import into a subdirectory You want to import a manifest and its projects, placing everything into a subdirectory of your *west workspace*.

For example, suppose you want to import this manifest from project `foo`, adding this project and its projects `bar` and `baz` to your workspace:

```
# foo/west.yml:
manifest:
  defaults:
    remote: example
  remotes:
    - name: example
      url-base: https://git.example.com
  projects:
    - name: bar
    - name: baz
```

Instead of importing these into the top level workspace, you want to place all three project repositories in an `external-code` subdirectory, like this:

```
workspace/
└── external-code/
    ├── foo/
    ├── bar/
    └── baz/
```

You can do this using this manifest:

```yaml
manifest:
  projects:
    - name: foo
      url: https://git.example.com/foo
      import:
        path-prefix: external-code
```

An equivalent manifest in a single file would be:

```yaml
# foo/west.yml:
manifest:
  defaults:
    remote: example
  remotes:
    - name: example
      url-base: https://git.example.com
  projects:
    - name: foo
      path: external-code/foo
    - name: bar
      path: external-code/bar
    - name: baz
      path: external-code/baz
```

Option 4: Sequence The `import` key can also contain a sequence of files, directories, and mappings.

Example 4.1: Downstream with sequence of manifest files This example manifest is equivalent to the manifest in *Example 2.2: Downstream with directory of manifest files*, with a sequence of explicitly named files.

```yaml
# my-repo/west.yml:
manifest:
  projects:
    - name: zephyr
      url: https://github.com/zephyrproject-rtos/zephyr
      import: west.yml
  self:
    import:
      - submanifests/01-libraries.yml
      - submanifests/02-vendor-hals.yml
      - submanifests/03-applications.yml
```

Example 4.2: Import order illustration This more complicated example shows the order that west imports manifest files:

```yaml
# my-repo/west.yml
manifest:
  # ...
```

(continues on next page)

(continued from previous page)

```
  projects:
    - name: my-library
    - name: my-app
    - name: zephyr
      import: true
    - name: another-manifest-repo
      import: submanifests
  self:
    import:
      - submanifests/libraries.yml
      - submanifests/vendor-hals.yml
      - submanifests/applications.yml
  defaults:
    remote: my-remote
```

For this example, west resolves imports in this order:

1. the listed files in `my-repo/submanifests` are first, in the order they occur (e.g. `libraries.yml` comes before `applications.yml`, since this is a sequence of files), since the `self: import:` is always imported first
2. `my-repo/west.yml` is next (with projects `my-library` etc. as long as they weren't already defined somewhere in `submanifests`)
3. `zephyr/west.yml` is after that, since that's the first `import` key in the `projects` list in `my-repo/west.yml`
4. files in `another-manifest-repo/submanifests` are last (sorted by file name), since that's the final project `import`

Manifest Import Details This section describes how west resolves a manifest file that uses `import` a bit more formally.

Overview The `import` key can appear in a west manifest's `projects` and `self` sections. The general case looks like this:

```
# Top-level manifest file.
manifest:
  projects:
    - name: foo
      import:
        ... # import-1
    - name: bar
      import:
        ... # import-2
    # ...
    - name: baz
      import:
        ... # import-N
  self:
    import:
      ... # self-import
```

Import keys are optional. If any of `import-1`, `...`, `import-N` are missing, west will not import additional manifest data from that project. If `self-import` is missing, no additional files in the manifest repository (beyond the top-level file) are imported.

The ultimate outcomes of resolving manifest imports are:

- a `projects` list, which is produced by combining the `projects` defined in the top-level file with those defined in imported files

- a set of extension commands, which are drawn from the `west-commands` keys in the top-level file and any imported files
- a `group-filter` list, which is produced by combining the top-level and any imported filters

Importing is done in this order:

1. Manifests from `self-import` are imported first.
2. The top-level manifest file's definitions are handled next.
3. Manifests from `import-1`, ..., `import-N`, are imported in that order.

When an individual `import` key refers to multiple manifest files, they are processed in this order:

- If the value is a relative path naming a directory (or a map whose `file` is a directory), the manifest files it contains are processed in lexicographic order – i.e., sorted by file name.
- If the value is a sequence, its elements are recursively imported in the order they appear.

This process recurses if necessary. E.g., if `import-1` produces a manifest file that contains an `import` key, it is resolved recursively using the same rules before its contents are processed further.

The following sections describe these outcomes.

Projects This section describes how the final `projects` list is created.

Projects are identified by name. If the same name occurs in multiple manifests, the first definition is used, and subsequent definitions are ignored. For example, if `import-1` contains a project named `bar`, that is ignored, because the top-level `west.yml` has already defined a project by that name.

The contents of files named by `import-1` through `import-N` are imported from Git at the latest `manifest-rev` revisions in their projects. These revisions can be updated to the values `rev-1` through `rev-N` by running `west update`. If any `manifest-rev` reference is missing or out of date, `west update` also fetches project data from the remote fetch URL and updates the reference.

Also note that all imported manifests, from the root manifest to the repository which defines a project `P`, must be up to date in order for west to update `P` itself. For example, this means `west update P` would update `manifest-rev` in the `baz` project if `baz/west.yml` defines `P`, as well as updating the `manifest-rev` branch in the local git clone of `P`. Confusingly, updating `baz` may result in the removal of `P` from `baz/west.yml`, which "should" cause `west update P` to fail with an unrecognized project!

For this reason, it's not possible to run `west update P` if `P` is defined in an imported manifest; you must update this project along with all the others with a plain `west update`.

By default, west won't fetch any project data over the network if a project's revision is a SHA or tag which is already available locally, so updating the extra projects shouldn't take too much time unless it's really needed. See the documentation for the *update.fetch* configuration option for more information.

Extensions All extension commands defined using `west-commands` keys discovered while handling imports are available in the resolved manifest.

If an imported manifest file has a `west-commands:` definition in its `self:` section, the extension commands defined there are added to the set of available extensions at the time the manifest is imported. They will thus take precedence over any extension commands with the same names added later on.

Group filters The resolved manifest has a `group-filter` value which is the result of concatenating the `group-filter` values in the top-level manifest and any imported manifests.

Manifest files which appear earlier in the import order have higher precedence and are therefore concatenated later into the final `group-filter`.

In other words, let:

- the submanifest resolved from `self-import` have group filter `self-filter`
- the top-level manifest file have group filter `top-filter`
- the submanifests resolved from `import-1` through `import-N` have group filters `filter-1` through `filter-N` respectively

The final resolved `group-filter` value is then `filterN + ... + filter-2 + filter-1 + top-filter + self-filter`, where + here refers to list concatenation.

Important

The order that filters appear in the above list matters.

The last filter element in the final concatenated list "wins" and determines if the group is enabled or disabled.

For example, in `[-foo] + [+foo]`, group `foo` is *enabled*. However, in `[+foo] + [-foo]`, group `foo` is *disabled*.

For simplicity, west and this documentation may elide concatenated group filter elements which are redundant using these rules. For example, `[+foo] + [-foo]` could be written more simply as `[-foo]`, for the reasons given above. As another example, `[-foo] + [+foo]` could be written as the empty list `[]`, since all groups are enabled by default.

Manifest Command

The `west manifest` command can be used to manipulate manifest files. It takes an action, and action-specific arguments.

The following sections describe each action and provides a basic signature for simple uses. Run `west manifest --help` for full details on all options.

Resolving Manifests The `--resolve` action outputs a single manifest file equivalent to your current manifest and all its *imported manifests*:

```
west manifest --resolve [-o outfile]
```

The main use for this action is to see the "final" manifest contents after performing any `import`s.

To print detailed information about each imported manifest file and how projects are handled during manifest resolution, set the maximum verbosity level using `-v`:

```
west -v manifest --resolve
```

Freezing Manifests The `--freeze` action outputs a frozen manifest:

```
west manifest --freeze [-o outfile]
```

A "frozen" manifest is a manifest file where every project's revision is a SHA. You can use `--freeze` to produce a frozen manifest that's equivalent to your current manifest file. The `-o` option specifies an output file; if not given, standard output is used.

Validating Manifests The `--validate` action either succeeds if the current manifest file is valid, or fails with an error:

```
west manifest --validate
```

The error message can help diagnose errors.

Here, "invalid" means that the syntax of the manifest file doesn't follow the rules documented on this page.

If your manifest is valid but it's not working the way you want it to, turning up the verbosity with `-v` is a good way to get detailed information about what decisions west made about your manifest, and why:

```
west -v manifest --validate
```

Get the manifest path The `--path` action prints the path to the top level manifest file:

```
west manifest --path
```

The output is something like `/path/to/workspace/west.yml`. The path format depends on your operating system.

2.11.8 Configuration

This page documents west's configuration file system, the `west config` command, and configuration options used by built-in commands. For API documentation on the `west.configuration` module, see west-apis-configuration.

West Configuration Files

West's configuration file syntax is INI-like; here is an example file:

```
[manifest]
path = zephyr

[zephyr]
base = zephyr
```

Above, the `manifest` section has option `path` set to `zephyr`. Another way to say the same thing is that `manifest.path` is `zephyr` in this file.

There are three types of configuration file:

1. **System**: Settings in this file affect west's behavior for every user logged in to the computer. Its location depends on the platform:
   - Linux: `/etc/westconfig`
   - macOS: `/usr/local/etc/westconfig`
   - Windows: `%PROGRAMDATA%\west\config`
2. **Global** (per user): Settings in this file affect how west behaves when run by a particular user on the computer.
   - All platforms: the default is `.westconfig` in the user's home directory.
   - Linux note: if the environment variable `XDG_CONFIG_HOME` is set, then `$XDG_CONFIG_HOME/west/config` is used.

- Windows note: the following environment variables are tested to find the home directory: `%HOME%`, then `%USERPROFILE%`, then a combination of `%HOMEDRIVE%` and `%HOMEPATH%`.

3. **Local**: Settings in this file affect west's behavior for the current *west workspace*. The file is `.west/config`, relative to the workspace's root directory.

A setting in a file which appears lower down on this list overrides an earlier setting. For example, if `color.ui` is `true` in the system's configuration file, but `false` in the workspace's, then the final value is `false`. Similarly, settings in the user configuration file override system settings, and so on.

west config

The built-in `config` command can be used to get and set configuration values. You can pass `west config` the options `--system`, `--global`, or `--local` to specify which configuration file to use. Only one of these can be used at a time. If none is given, then writes default to `--local`, and reads show the final value after applying overrides.

Some examples for common uses follow; run `west config -h` for detailed help, and see *Built-in Configuration Options* for more details on built-in options.

To set `manifest.path` to `some-other-manifest`:

```
west config manifest.path some-other-manifest
```

Doing the above means that commands like `west update` will look for the *west manifest* inside the `some-other-manifest` directory (relative to the workspace root directory) instead of the directory given to `west init`, so be careful!

To read `zephyr.base`, the value which will be used as `ZEPHYR_BASE` if it is unset in the calling environment (also relative to the workspace root):

```
west config zephyr.base
```

You can switch to another zephyr repository without changing `manifest.path` – and thus the behavior of commands like `west update` – using:

```
west config zephyr.base some-other-zephyr
```

This can be useful if you use commands like `git worktree` to create your own zephyr directories, and want commands like `west build` to use them instead of the zephyr repository specified in the manifest. (You can go back to using the directory in the upstream manifest by running `west config zephyr.base zephyr`.)

To set `color.ui` to `false` in the global (user-wide) configuration file, so that west will no longer print colored output for that user when run in any workspace:

```
west config --global color.ui false
```

To undo the above change:

```
west config --global color.ui true
```

Built-in Configuration Options

The following table documents configuration options supported by west's built-in commands. Configuration options supported by Zephyr's extension commands are documented in the pages for those commands.

Option	Description
`color.ui`	Boolean. If `true` (the default), then west output is colorized when stdout is a terminal.
`commands.allow_extensions`	Boolean, default `true`, disables *Extensions* if `false`
`grep.color`	String, default empty. Set this to `never` to disable `west grep` color output. If set, `west grep` passes the value to the grep tool's `--color` option.
`grep.tool`	String, one of `"git-grep"` (default), `"ripgrep"`, or `"grep"`. The grep tool that `west grep` should use.
`grep.<TOOL>-args`	String, default empty. The `<TOOL>` part is a pattern that can be any `grep.tool` value, so `grep.ripgrep-args` is an example configuration option. If set, arguments that `west grep` should pass to the corresponding grep tool. Run `west help grep` for details.
`grep.<TOOL>-path`	String, default empty. The `<TOOL>` part is a pattern that can be any `grep.tool` value, so `grep.ripgrep-path` is an example configuration option. The path to the corresponding tool that `west grep` should use instead of searching for the command. Run `west help grep` for details.
`manifest.file`	String, default `west.yml`. Relative path from the manifest repository root directory to the manifest file used by `west init` and other commands which parse the manifest.
`manifest.group-filter`	String, default empty. A comma-separated list of project groups to enable and disable within the workspace. Prefix enabled groups with `+` and disabled groups with `-`. For example, the value `"+foo,-bar"` enables group `foo` and disables `bar`. See *Project Groups*.
`manifest.path`	String, relative path from the *west workspace* root directory to the manifest repository used by `west update` and other commands which parse the manifest. Set locally by `west init`.

`manifest.project-filter`	Comma-separated list of strings.
	The option's value is a comma-separated list of regular expressions, each prefixed with `+` or `-`, like this:
	`+re1,-re2,-re3`
	Project names are matched against each regular expression (`re1`, `re2`, `re3`, ...) in the list, in order. If the entire project name matches the regular expression, that element of the list either deactivates or activates the project. The project is deactivated if the element begins with `-`. The project is activated if the element begins with `+`. (Project names cannot contain `,` if this option is used, so the regular expressions do not need to contain a literal `,` character.)
	If a project's name matches multiple regular expressions in the list, the result from the last regular expression is used. For example, if `manifest.project-filter` is:
	`-hal_.*,+hal_foo`
	Then a project named `hal_bar` is inactive, but a project named `hal_foo` is active.
	If a project is made inactive or active by a list element, the project is active or not regardless of whether any or all of its groups are disabled. (This is currently the only way to make a project that has no groups inactive.)
	Otherwise, i.e. if a project does not match any regular expressions in the list, it is active or inactive according to the usual rules related to its groups (see *Project Group Examples* for examples in that case).
	Within an element of a `manifest.project-filter` list, leading and trailing whitespace are ignored. That means these example values are equivalent:
	`+foo,-bar`
	`+foo , -bar`
	Any empty elements are ignored. That means these example values are equivalent:
	`+foo,,-bar`
	`+foo,-bar`

`update.fetch`	String, one of `"smart"` (the default behavior starting in v0.6.1) or `"al-`

2.11.9 Extensions

West is "pluggable": you can add your own commands to west without editing its source code. These are called **west extension commands**, or just "extensions" for short. Extensions show up in the `west --help` output in a special section for the project which defines them. This page provides general information on west extension commands, and has a tutorial for writing your own.

Some commands you can run when using west with Zephyr, like the ones used to *build, flash, and debug* and the *ones described here* , are extensions. That's why help for them shows up like this in `west --help`:

```
commands from project at "zephyr":
  completion:           display shell completion scripts
  boards:               display information about supported boards
  build:                compile a Zephyr application
  sign:                 sign a Zephyr binary for bootloader chain-loading
  flash:                flash and run a binary on a board
  debug:                flash and interactively debug a Zephyr application
  debugserver:          connect to board and launch a debug server
  attach:               interactively debug a board
```

See `zephyr/scripts/west-commands.yml` and the `zephyr/scripts/west_commands` directory for the implementation details.

Disabling Extension Commands

To disable support for extension commands, set the `commands.allow_extensions` *configuration* option to `false`. To set this globally for whenever you run west, use:

```
west config --global commands.allow_extensions false
```

If you want to, you can then re-enable them in a particular *west workspace* with:

```
west config --local commands.allow_extensions true
```

Note that the files containing extension commands are not imported by west unless the commands are explicitly run. See below for details.

Adding a West Extension

There are three steps to adding your own extension:

1. Write the code implementing the command.
2. Add information about it to a `west-commands.yml` file.
3. Make sure the `west-commands.yml` file is referenced in the *west manifest*.

Note that west ignores extension commands whose names are the same as a built-in command.

Step 1: Implement Your Command Create a Python file to contain your command implementation (see the "Meta > Requires" information on the west PyPI page for details on the currently supported versions of Python). You can put it in anywhere in any project tracked by your *west manifest*, or the manifest repository itself. This file must contain a subclass of the `west.commands.WestCommand` class; this class will be instantiated and used when your extension is run.

Here is a basic skeleton you can use to get started. It contains a subclass of `WestCommand`, with implementations for all the abstract methods. For more details on the west APIs you can use, see west-apis.

```
'''my_west_extension.py

Basic example of a west extension.'''

from textwrap import dedent            # just for nicer code indentation

from west.commands import WestCommand  # your extension must subclass this
from west import log                   # use this for user output

class MyCommand(WestCommand):

    def __init__(self):
        super().__init__(
            'my-command-name',  # gets stored as self.name
            'one-line help for what my-command-name does',  # self.help
            # self.description:
            dedent('''
            A multi-line description of my-command.

            You can split this up into multiple paragraphs and they'll get
            reflowed for you. You can also pass
            formatter_class=argparse.RawDescriptionHelpFormatter when calling
            parser_adder.add_parser() below if you want to keep your line
            endings.'''))

    def do_add_parser(self, parser_adder):
        # This is a bit of boilerplate, which allows you full control over the
        # type of argparse handling you want. The "parser_adder" argument is
        # the return value of an argparse.ArgumentParser.add_subparsers() call.
        parser = parser_adder.add_parser(self.name,
                                         help=self.help,
                                         description=self.description)

        # Add some example options using the standard argparse module API.
        parser.add_argument('-o', '--optional', help='an optional argument')
        parser.add_argument('required', help='a required argument')

        return parser           # gets stored as self.parser

    def do_run(self, args, unknown_args):
        # This gets called when the user runs the command, e.g.:
        #
        #   $ west my-command-name -o FOO BAR
        #   --optional is FOO
        #   required is BAR
        log.inf('--optional is', args.optional)
        log.inf('required is', args.required)
```

You can ignore the second argument to `do_run()` (`unknown_args` above), as `WestCommand` will reject unknown arguments by default. If you want to be passed a list of unknown arguments instead, add `accepts_unknown_args=True` to the `super().__init__()` arguments.

Step 2: Add or Update Your `west-commands.yml` You now need to add a `west-commands.yml` file to your project which describes your extension to west.

Here is an example for the above class definition, assuming it's in `my_west_extension.py` at the project root directory:

```
west-commands:
  - file: my_west_extension.py
    commands:
```

(continues on next page)

(continued from previous page)

```
      - name: my-command-name
        class: MyCommand
        help: one-line help for what my-command-name does
```

The top level of this YAML file is a map with a `west-commands` key. The key's value is a sequence of "command descriptors". Each command descriptor gives the location of a file implementing west extensions, along with the names of those extensions, and optionally the names of the classes which define them (if not given, the `class` value defaults to the same thing as `name`).

Some information in this file is redundant with definitions in the Python code. This is because west won't import `my_west_extension.py` until the user runs `west my-command-name`, since:

- It allows users to run `west update` with a manifest from an untrusted source, then use other west commands without your code being imported along the way. Since importing a Python module is shell-equivalent, this provides some peace of mind.
- It's a small optimization, since your code will only be imported if it is needed.

So, unless your command is explicitly run, west will just load the `west-commands.yml` file to get the basic information it needs to display information about your extension to the user in `west --help` output, etc.

If you have multiple extensions, or want to split your extensions across multiple files, your `west-commands.yml` will look something like this:

```
west-commands:
  - file: my_west_extension.py
    commands:
      - name: my-command-name
        class: MyCommand
        help: one-line help for what my-command-name does
  - file: another_file.py
    commands:
      - name: command2
        help: another cool west extension
      - name: a-third-command
        class: ThirdCommand
        help: a third command in the same file as command2
```

Above:

- `my_west_extension.py` defines extension `my-command-name` with class `MyCommand`
- `another_file.py` defines two extensions:
  1. `command2` with class `command2`
  2. `a-third-command` with class `ThirdCommand`

See the file `west-commands-schema.yml` in the west repository for a schema describing the contents of a `west-commands.yml`.

Step 3: Update Your Manifest Finally, you need to specify the location of the `west-commands.yml` you just edited in your west manifest. If your extension is in a project, add it like this:

```
manifest:
   # [... other contents ...]

   projects:
     - name: your-project
       west-commands: path/to/west-commands.yml
     # [... other projects ...]
```

Where `path/to/west-commands.yml` is relative to the root of the project. Note that the name `west-commands.yml`, while encouraged, is just a convention; you can name the file something else if you need to.

Alternatively, if your extension is in the manifest repository, just do the same thing in the manifest's `self` section, like this:

```
manifest:
  # [... other contents ...]

  self:
    west-commands: path/to/west-commands.yml
```

That's it; you can now run `west my-command-name`. Your command's name, help, and the project which contains its code will now also show up in the `west --help` output. If you share the updated repositories with others, they'll be able to use it, too.

2.11.10 Building, Flashing and Debugging

Zephyr provides several *west extension commands* for building, flashing, and interacting with Zephyr programs running on a board: `build`, `flash`, `debug`, `debugserver` and `attach`.

For information on adding board support for the flashing and debugging commands, see *Flash and debug support* in the board porting guide.

Building: `west build`

> **Tip**
>
> Run `west build -h` for a quick overview.

The `build` command helps you build Zephyr applications from source. You can use *west config* to configure its behavior.

Its default behavior tries to "do what you mean":

- If there is a Zephyr build directory named `build` in your current working directory, it is incrementally re-compiled. The same is true if you run `west build` from a Zephyr build directory.
- Otherwise, if you run `west build` from a Zephyr application's source directory and no build directory is found, a new one is created and the application is compiled in it.

Basics The easiest way to use `west build` is to go to an application's root directory (i.e. the folder containing the application's `CMakeLists.txt`) and then run:

```
west build -b <BOARD>
```

Where `<BOARD>` is the name of the board you want to build for. This is exactly the same name you would supply to CMake if you were to invoke it with: `cmake -DBOARD=<BOARD>`.

> **Tip**
>
> You can use the *west boards* command to list all supported boards.

A build directory named `build` will be created, and the application will be compiled there after `west build` runs CMake to create a build system in that directory. If `west build` finds an existing build directory, the application is incrementally re-compiled there without re-running CMake. You can force CMake to run again with `--cmake`.

You don't need to use the `--board` option if you've already got an existing build directory; `west build` can figure out the board from the CMake cache. For new builds, the `--board` option, *BOARD* environment variable, or `build.board` configuration option are checked (in that order).

Sysbuild (multi-domain builds) *Sysbuild (System build)* can be used to create a multi-domain build system combining multiple images for a single or multiple boards.

Use `--sysbuild` to select the *Sysbuild (System build)* build infrastructure with `west build` to build multiple domains.

More detailed information regarding the use of sysbuild can be found in the *Sysbuild (System build)* guide.

Tip

The `build.sysbuild` configuration option can be enabled to tell `west build` to default build using sysbuild. `--no-sysbuild` can be used to disable sysbuild for a specific build.

`west build` will build all domains through the top-level build folder of the domains specified by sysbuild.

A single domain from a multi-domain project can be built by using `--domain` argument.

Examples Here are some `west build` usage examples, grouped by area.

Forcing CMake to Run Again To force a CMake re-run, use the `--cmake` (or `-c`) option:

```
west build -c
```

Setting a Default Board To configure `west build` to build for the `reel_board` by default:

```
west config build.board reel_board
```

(You can use any other board supported by Zephyr here; it doesn't have to be `reel_board`.)

Setting Source and Build Directories To set the application source directory explicitly, give its path as a positional argument:

```
west build -b <BOARD> path/to/source/directory
```

To set the build directory explicitly, use `--build-dir` (or `-d`):

```
west build -b <BOARD> --build-dir path/to/build/directory
```

To change the default build directory from `build`, use the `build.dir-fmt` configuration option. This lets you name build directories using format strings, like this:

```
west config build.dir-fmt "build/{board}/{app}"
```

With the above, running `west build -b reel_board samples/hello_world` will use build directory `build/reel_board/hello_world`. See *Configuration Options* for more details on this option.

Setting the Build System Target To specify the build system target to run, use `--target` (or `-t`).

For example, on host platforms with QEMU, you can use the `run` target to build and run the hello_world sample for the emulated qemu_x86 board in one command:

```
west build -b qemu_x86 -t run samples/hello_world
```

As another example, to use `-t` to list all build system targets:

```
west build -t help
```

As a final example, to use `-t` to run the `pristine` target, which deletes all the files in the build directory:

```
west build -t pristine
```

Pristine Builds A *pristine* build directory is essentially a new build directory. All byproducts from previous builds have been removed.

To force `west build` make the build directory pristine before re-running CMake to generate a build system, use the `--pristine=always` (or `-p=always`) option.

Giving `--pristine` or `-p` without a value has the same effect as giving it the value `always`. For example, these commands are equivalent:

```
west build -p -b reel_board samples/hello_world
west build -p=always -b reel_board samples/hello_world
```

By default, `west build` makes no attempt to detect if the build directory needs to be made pristine. This can lead to errors if you do something like try to reuse a build directory for a different `--board`.

Using `--pristine=auto` makes `west build` detect some of these situations and make the build directory pristine before trying the build.

Tip

You can run `west config build.pristine always` to always do a pristine build, or `west config build.pristine never` to disable the heuristic. See the `west build` *Configuration Options* for details.

Verbose Builds To print the CMake and compiler commands run by `west build`, use the global west verbosity option, `-v`:

```
west -v build -b reel_board samples/hello_world
```

One-Time CMake Arguments To pass additional arguments to the CMake invocation performed by `west build`, pass them after a `--` at the end of the command line.

Important

Passing additional CMake arguments like this forces `west build` to re-run the CMake build configuration step, even if a build system has already been generated. This will make incremental builds slower (but still much faster than building from scratch).

After using `--` once to generate the build directory, use `west build -d <build-dir>` on subsequent runs to do incremental builds.

Alternatively, make your CMake arguments permanent as described in the next section; it will not slow down incremental builds.

For example, to use the Unix Makefiles CMake generator instead of Ninja (which `west build` uses by default), run:

```
west build -b reel_board -- -G'Unix Makefiles'
```

To use Unix Makefiles and set CMAKE_VERBOSE_MAKEFILE to ON:

```
west build -b reel_board -- -G'Unix Makefiles' -DCMAKE_VERBOSE_MAKEFILE=ON
```

Notice how the `--` only appears once, even though multiple CMake arguments are given. All command-line arguments to `west build` after a `--` are passed to CMake.

To set *DTC_OVERLAY_FILE* to `enable-modem.overlay`, using that file as a *devicetree overlay*:

```
west build -b reel_board -- -DDTC_OVERLAY_FILE=enable-modem.overlay
```

To merge the `file.conf` Kconfig fragment into your build's `.config`:

```
west build -- -DEXTRA_CONF_FILE=file.conf
```

Permanent CMake Arguments The previous section describes how to add CMake arguments for a single `west build` command. If you want to save CMake arguments for `west build` to use every time it generates a new build system instead, you should use the `build.cmake-args` configuration option. Whenever `west build` runs CMake to generate a build system, it splits this option's value according to shell rules and includes the results in the `cmake` command line.

Remember that, by default, `west build` **tries to avoid generating a new build system if one is present** in your build directory. Therefore, you need to either delete any existing build directories or do a *pristine build* after setting `build.cmake-args` to make sure it will take effect.

For example, to always enable **CMAKE_EXPORT_COMPILE_COMMANDS**, you can run:

```
west config build.cmake-args -- -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
```

(The extra `--` is used to force the rest of the command to be treated as a positional argument. Without it, *west config* would treat the `-DVAR=VAL` syntax as a use of its `-D` option.)

To enable **CMAKE_VERBOSE_MAKEFILE**, so CMake always produces a verbose build system:

```
west config build.cmake-args -- -DCMAKE_VERBOSE_MAKEFILE=ON
```

To save more than one argument in `build.cmake-args`, use a single string whose value can be split into distinct arguments (`west build` uses the Python function shlex.split() internally to split the value).

For example, to enable both **CMAKE_EXPORT_COMPILE_COMMANDS** and **CMAKE_VERBOSE_MAKEFILE**:

```
west config build.cmake-args -- "-DCMAKE_EXPORT_COMPILE_COMMANDS=ON -DCMAKE_VERBOSE_
↪MAKEFILE=ON"
```

If you want to save your CMake arguments in a separate file instead, you can combine CMake's `-C <initial-cache>` option with `build.cmake-args`. For instance, another way to set the options used in the previous example is to create a file named `~/my-cache.cmake` with the following contents:

```
set(CMAKE_EXPORT_COMPILE_COMMANDS ON CACHE BOOL "")
set(CMAKE_VERBOSE_MAKEFILE ON CACHE BOOL "")
```

Then run:

```
west config build.cmake-args "-C ~/my-cache.cmake"
```

See the cmake(1) manual page and the set() command documentation for more details.

Build tool arguments Use `-o` to pass options to the underlying build tool.

This works with both `ninja` (*the default*) and `make` based build systems.

For example, to pass `-dexplain` to `ninja`:

```
west build -o=-dexplain
```

As another example, to pass `--keep-going` to `make`:

```
west build -o=--keep-going
```

Note that using `-o=--foo` instead of `-o --foo` is required to prevent `--foo` from being treated as a `west build` option.

Build parallelism By default, `ninja` uses all of your cores to build, while `make` uses only one. You can control this explicitly with the `-j` option supported by both tools.

For example, to build with 4 cores:

```
west build -o=-j4
```

The `-o` option is described further in the previous section.

Build a single domain In a multi-domain build with hello_world and MCUboot, you can use `--domain hello_world` to only build this domain:

```
west build --sysbuild --domain hello_world
```

The `--domain` argument can be combined with the `--target` argument to build the specific target for the target, for example:

```
west build --sysbuild --domain hello_world --target help
```

Use a snippet See *Using Snippets*.

Configuration Options You can *configure* `west build` using these options.

Option	Description
build.board	String. If given, this the board used by *west build* when --board is not given and BOARD is unset in the environment.
build.board_warn	Boolean, default true. If false, disables warnings when west build can't figure out the target board.
build.cmake-args	String. If present, the value will be split according to shell rules and passed to CMake whenever a new build system is generated. See *Permanent CMake Arguments*.
build.dir-fmt	String, default build. The build folder format string, used by west whenever it needs to create or locate a build folder. The currently available arguments are: • board: The board name • source_dir: The relative path from the current working directory to the source directory. If the current working directory is inside the source directory this will be set to an empty string. • app: The name of the source directory.
build.generator	String, default Ninja. The CMake Generator to use to create a build system. (To set a generator for a single build, see the *above example*)
build.guess-dir	String, instructs west whether to try to guess what build folder to use when build.dir-fmt is in use and not enough information is available to resolve the build folder name. Can take these values: • never (default): Never try to guess, bail out instead and require the user to provide a build folder with -d. • runners: Try to guess the folder when using any of the 'runner' commands. These are typically all commands that invoke an external tool, such as flash and debug.
build.pristine	String. Controls the way in which west build may clean the build folder before building. Can take the following values: • never (default): Never automatically make the build folder pristine. • auto: west build will automatically make the build folder pristine before building, if a build system is present and the build would fail otherwise (e.g. the user has specified a different board or application from the one previously used to make the build directory). • always: Always make the build folder pristine before building, if a build system is present.
build.sysbuild	Boolean, default false. If true, build application using the sysbuild infrastructure.

Flashing: west flash

Tip

Run west flash -h for additional help.

Basics From a Zephyr build directory, re-build the binary and flash it to your board:

```
west flash
```

Without options, the behavior is the same as ninja flash (or make flash, etc.).

To specify the build directory, use `--build-dir` (or `-d`):

```
west flash --build-dir path/to/build/directory
```

If you don't specify the build directory, `west flash` searches for one in `build`, then the current working directory. If you set the `build.dir-fmt` configuration option (see *Setting Source and Build Directories*), `west flash` searches there instead of `build`.

Choosing a Runner If your board's Zephyr integration supports flashing with multiple programs, you can specify which one to use using the `--runner` (or `-r`) option. For example, if West flashes your board with `nrfjprog` by default, but it also supports JLink, you can override the default with:

```
west flash --runner jlink
```

You can override the default flash runner at build time by using the `BOARD_FLASH_RUNNER` CMake variable, and the debug runner with `BOARD_DEBUG_RUNNER`.

For example:

```
# Set the default runner to "jlink", overriding the board's
# usual default.
west build [...] -- -DBOARD_FLASH_RUNNER=jlink
```

See *One-Time CMake Arguments* and *Permanent CMake Arguments* for more information on setting CMake arguments.

See *Flash and debug runners* below for more information on the `runner` library used by West. The list of runners which support flashing can be obtained with `west flash -H`; if run from a build directory or with `--build-dir`, this will print additional information on available runners for your board.

Configuration Overrides The CMake cache contains default values West uses while flashing, such as where the board directory is on the file system, the path to the zephyr binaries to flash in several formats, and more. You can override any of this configuration at runtime with additional options.

For example, to override the HEX file containing the Zephyr image to flash (assuming your runner expects a HEX file), but keep other flash configuration at default values:

```
west flash --hex-file path/to/some/other.hex
```

The `west flash -h` output includes a complete list of overrides supported by all runners.

Runner-Specific Overrides Each runner may support additional options related to flashing. For example, some runners support an `--erase` flag, which mass-erases the flash storage on your board before flashing the Zephyr image.

To view all of the available options for the runners your board supports, as well as their usage information, use `--context` (or `-H`):

```
west flash --context
```

Important

Note the capital H in the short option name. This re-runs the build in order to ensure the information displayed is up to date!

When running West outside of a build directory, `west flash -H` just prints a list of runners. You can use `west flash -H -r <runner-name>` to print usage information for options supported by that runner.

For example, to print usage information about the `jlink` runner:

```
west flash -H -r jlink
```

Multi-domain flashing When a *Sysbuild (multi-domain builds)* folder is detected, then `west flash` will flash all domains in the order defined by sysbuild.

It is possible to flash the image from a single domain in a multi-domain project by using `--domain`.

For example, in a multi-domain build with hello_world and MCUboot, you can use the `--domain hello_world` domain to only flash only the image from this domain:

```
west flash --domain hello_world
```

Debugging: `west debug`, `west debugserver`

> **Tip**
>
> Run `west debug -h` or `west debugserver -h` for additional help.

Basics From a Zephyr build directory, to attach a debugger to your board and open up a debug console (e.g. a GDB session):

```
west debug
```

To attach a debugger to your board and open up a local network port you can connect a debugger to (e.g. an IDE debugger):

```
west debugserver
```

Without options, the behavior is the same as `ninja debug` and `ninja debugserver` (or `make debug`, etc.).

To specify the build directory, use `--build-dir` (or `-d`):

```
west debug --build-dir path/to/build/directory
west debugserver --build-dir path/to/build/directory
```

If you don't specify the build directory, these commands search for one in `build`, then the current working directory. If you set the `build.dir-fmt` configuration option (see *Setting Source and Build Directories*), `west debug` searches there instead of `build`.

Choosing a Runner If your board's Zephyr integration supports debugging with multiple programs, you can specify which one to use using the `--runner` (or `-r`) option. For example, if West debugs your board with `pyocd-gdbserver` by default, but it also supports JLink, you can override the default with:

```
west debug --runner jlink
west debugserver --runner jlink
```

See *Flash and debug runners* below for more information on the `runner` library used by West. The list of runners which support debugging can be obtained with `west debug -H`; if run from a

build directory or with `--build-dir`, this will print additional information on available runners for your board.

Configuration Overrides The CMake cache contains default values West uses for debugging, such as where the board directory is on the file system, the path to the zephyr binaries containing symbol tables, and more. You can override any of this configuration at runtime with additional options.

For example, to override the ELF file containing the Zephyr binary and symbol tables (assuming your runner expects an ELF file), but keep other debug configuration at default values:

```
west debug --elf-file path/to/some/other.elf
west debugserver --elf-file path/to/some/other.elf
```

The `west debug -h` output includes a complete list of overrides supported by all runners.

Runner-Specific Overrides Each runner may support additional options related to debugging. For example, some runners support flags which allow you to set the network ports used by debug servers.

To view all of the available options for the runners your board supports, as well as their usage information, use `--context` (or `-H`):

```
west debug --context
```

(The command `west debugserver --context` will print the same output.)

Important

Note the capital H in the short option name. This re-runs the build in order to ensure the information displayed is up to date!

When running West outside of a build directory, `west debug -H` just prints a list of runners. You can use `west debug -H -r <runner-name>` to print usage information for options supported by that runner.

For example, to print usage information about the `jlink` runner:

```
west debug -H -r jlink
```

Multi-domain debugging `west debug` can only debug a single domain at a time. When a *Sysbuild (multi-domain builds)* folder is detected, `west debug` will debug the `default` domain specified by sysbuild.

The default domain will be the application given as the source directory. See the following example:

```
west build --sysbuild path/to/source/directory
```

For example, when building `hello_world` with MCUboot using sysbuild, `hello_world` becomes the default domain:

```
west build --sysbuild samples/hello_world
```

So to debug `hello_world` you can do:

```
west debug
```

or:

```
west debug --domain hello_world
```

If you wish to debug MCUboot, you must explicitly specify MCUboot as the domain to debug:

```
west debug --domain mcuboot
```

Flash and debug runners

The flash and debug commands use Python wrappers around various *Flash & Debug Host Tools*. These wrappers are all defined in a Python library at scripts/west_commands/runners. Each wrapper is called a *runner*. Runners can flash and/or debug Zephyr programs.

The central abstraction within this library is `ZephyrBinaryRunner`, an abstract class which represents runners. The set of available runners is determined by the imported subclasses of `ZephyrBinaryRunner`. `ZephyrBinaryRunner` is available in the `runners.core` module; individual runner implementations are in other submodules, such as `runners.nrfjprog`, `runners.openocd`, etc.

Running Robot Framework tests: `west robot`

> **Tip**
>
> Run `west robot -h` for additional help.

Basics Currently the command supports only one runner which is using `renode-test`, (essentially a wrapper for running Robot tests in Renode), but can be easily extended by adding other runners.

From a Zephyr build directory, to run a Robot test suite:

```
west robot --runner=renode-robot --testsuite path/to/testsuite.robot
```

This will run all tests from testsuite.robot and print output provided by Robot Framework.

To pass additional parameters to Renode use `--renode-robot-args` switch. For example to show Renode logs in addition to Robot Framework's output:

> west robot –runner=renode-robot –testsuite path/to/testsuite.robot –renode-robot-arg="–show-log"

Runner-Specific Overrides To view all of the available options for the Robot runners your board supports, as well as their usage information, use `--context` (or `-H`):

```
west robot --runner=renode-robot --context
```

To view all available options "renode-test" runner supports, use:

```
west robot --runner=renode-robot --renode-robot-help
```

Simulating a board with: `west simulate`

Basics Currently the command supports only one runner which is using Renode, but can be easily extended by adding other runners.

From a Zephyr build directory, to run the built binary:

```
west simulate --runner=renode
```

This will start Renode and configure simulation based on a default `.resc` script for the current platform with the zephyr.elf file loaded by default. The simulation then can be started by typing "start" or "s" in Renode's Monitor. This can also be done by passing a command to Renode, using an argument provided by the runner:

west simulate –runner=renode –renode-command start

To pass an argument to Renode itself, for example to start Renode in console mode instead of a separate window:

west simulate –runner=renode –renode-arg="–console"

From that point on Renode can be used normally in both console and window modes. For details on using Renode see Renode - documentation.

Runner-Specific Overrides To view all of the available options supported by the runners, as well as their usage information, use `--context` (or"-H"):

```
west simulate --runner=renode --context
```

To view all available options Renode supports, use:

```
west simulate --runner=renode --renode-help
```

Hacking

This section documents the `runners.core` module used by the flash and debug commands. This is the core abstraction used to implement support for these features.

Warning

These APIs are provided for reference, but they are more "shared code" used to implement multiple extension commands than a stable API.

Developers can add support for new ways to flash and debug Zephyr programs by implementing additional runners. To get this support into upstream Zephyr, the runner should be added into a new or existing `runners` module, and imported from `runners/__init__.py`.

Note

The test cases in scripts/west_commands/tests add unit test coverage for the runners package and individual runner classes.

Please try to add tests when adding new runners. Note that if your changes break existing test cases, CI testing on upstream pull requests will fail.

Zephyr binary runner core interfaces

This provides the core ZephyrBinaryRunner class meant for public use, as well as some other helpers for concrete runner classes.

class runners.core.**BuildConfiguration**(*build_dir: str*)

This helper class provides access to build-time configuration.

Configuration options can be read as if the object were a dict, either object['CONFIG_FOO'] or object.get('CONFIG_FOO').

Kconfig configuration values are available (parsed from .config).

getboolean(*option*)

If a boolean option is explicitly set to y or n, returns its value. Otherwise, falls back to False.

class runners.core.**DeprecatedAction**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

class runners.core.**FileType**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

exception runners.core.**MissingProgram**(*program*)

FileNotFoundError subclass for missing program dependencies.

No significant changes from the parent FileNotFoundError; this is useful for explicitly signaling that the file in question is a program that some class requires to proceed.

The filename attribute contains the missing program.

class runners.core.**NetworkPortHelper**

Helper class for dealing with local IP network ports.

get_unused_ports(*starting_from*)

Find unused network ports, starting at given values.

starting_from is an iterable of ports the caller would like to use.

The return value is an iterable of ports, in the same order, using the given values if they were unused, or the next sequentially available unused port otherwise.

Ports may be bound between this call's check and actual usage, so callers still need to handle errors involving returned ports.

class runners.core.**RunnerCaps**(*commands: ~typing.Set[str] = <factory>*, *dev_id: bool = False*, *flash_addr: bool = False*, *erase: bool = False*, *reset: bool = False*, *extload: bool = False*, *tool_opt: bool = False*, *file: bool = False*, *hide_load_files: bool = False*)

This class represents a runner class's capabilities.

Each capability is represented as an attribute with the same name. Flag attributes are True or False.

Available capabilities:

- commands: set of supported commands; default is {'flash', 'debug', 'debugserver', 'attach', 'simulate', 'robot'}.
- dev_id: whether the runner supports device identifiers, in the form of an -i, –dev-id option. This is useful when the user has multiple debuggers connected to a single computer, in order to select which one will be used with the command provided.
- flash_addr: whether the runner supports flashing to an arbitrary address. Default is False. If true, the runner must honor the –dt-flash option.
- erase: whether the runner supports an –erase option, which does a mass-erase of the entire addressable flash on the target before flashing. On multi-core SoCs, this may only erase portions of flash specific the actual target core. (This option can be useful for things like clearing out old settings values or other subsystem state that may affect

the behavior of the zephyr image. It is also sometimes needed by SoCs which have flash-like areas that can't be sector erased by the underlying tool before flashing; UICR on nRF SoCs is one example.)

- reset: whether the runner supports a –reset option, which resets the device after a flash operation is complete.
- extload: whether the runner supports a –extload option, which must be given one time and is passed on to the underlying tool that the runner wraps.
- tool_opt: whether the runner supports a –tool-opt (-O) option, which can be given multiple times and is passed on to the underlying tool that the runner wraps.
- file: whether the runner supports a –file option, which specifies exactly the file that should be used to flash, overriding any default discovered in the build directory.
- hide_load_files: whether the elf/hex/bin file arguments should be hidden.

class runners.core.**RunnerConfig**(*build_dir: str*, *board_dir: str*, *elf_file: str | None*, *exe_file: str | None*, *hex_file: str | None*, *bin_file: str | None*, *uf2_file: str | None*, *file: str | None*, *file_type:* FileType *| None = FileType.OTHER*, *gdb: str | None = None*, *openocd: str | None = None*, *openocd_search: List[str] = []*)

Runner execution-time configuration.

This is a common object shared by all runners. Individual runners can register specific configuration options using their do_add_parser() hooks.

bin_file: str | None
Alias for field number 5

board_dir: str
Alias for field number 1

build_dir: str
Alias for field number 0

elf_file: str | None
Alias for field number 2

exe_file: str | None
Alias for field number 3

file: str | None
Alias for field number 7

file_type: *FileType* **| None**
Alias for field number 8

gdb: str | None
Alias for field number 9

hex_file: str | None
Alias for field number 4

openocd: str | None
Alias for field number 10

openocd_search: List[str]
Alias for field number 11

uf2_file: str | None
Alias for field number 6

class runners.core.**SysbuildConfiguration**(*build_dir: str*)

This helper class provides access to sysbuild-time configuration.

Configuration options can be read as if the object were a dict, either object['SB_CONFIG_FOO'] or object.get('SB_CONFIG_FOO').

Kconfig configuration values are available (parsed from .config).

class runners.core.**ZephyrBinaryRunner**(*cfg:* RunnerConfig)

Abstract superclass for binary runners (flashers, debuggers).

Note: this class's API has changed relatively rarely since it as added, but it is not considered a stable Zephyr API, and may change without notice.

With some exceptions, boards supported by Zephyr must provide generic means to be flashed (have a Zephyr firmware binary permanently installed on the device for running) and debugged (have a breakpoint debugger and program loader on a host workstation attached to a running target).

This is supported by four top-level commands managed by the Zephyr build system:

- 'flash': flash a previously configured binary to the board, start execution on the target, then return.
- 'debug': connect to the board via a debugging protocol, program the flash, then drop the user into a debugger interface with symbol tables loaded from the current binary, and block until it exits.
- 'debugserver': connect via a board-specific debugging protocol, then reset and halt the target. Ensure the user is now able to connect to a debug server with symbol tables loaded from the binary.
- 'attach': connect to the board via a debugging protocol, then drop the user into a debugger interface with symbol tables loaded from the current binary, and block until it exits. Unlike 'debug', this command does not program the flash.

This class provides an API for these commands. Every subclass is called a 'runner' for short. Each runner has a name (like 'pyocd'), and declares commands it can handle (like 'flash'). Boards (like 'nrf52dk/nrf52832') declare which runner(s) are compatible with them to the Zephyr build system, along with information on how to configure the runner to work with the board.

The build system will then place enough information in the build directory to create and use runners with this class's create() method, which provides a command line argument parsing API. You can also create runners by instantiating subclasses directly.

In order to define your own runner, you need to:

1. Define a ZephyrBinaryRunner subclass, and implement its abstract methods. You may need to override capabilities().
2. Make sure the Python module defining your runner class is imported, e.g. by editing this package's __init__.py (otherwise, get_runners() won't work).
3. Give your runner's name to the Zephyr build system in your board's board.cmake.

Additional advice:

- If you need to import any non-standard-library modules, make sure to catch ImportError and defer complaints about it to a RuntimeError if one is missing. This avoids affecting users that don't require your runner, while still making it clear what went wrong to users that do require it that don't have the necessary modules installed.
- If you need to ask the user something (e.g. using input()), do it in your create() classmethod, not do_run(). That ensures your __init__() really has everything it needs to call do_run(), and also avoids calling input() when not instantiating within a command line application.

- Use self.logger to log messages using the standard library's logging API; your logger is named "runner.<your-runner-name()>"

For command-line invocation from the Zephyr build system, runners define their own argparse-based interface through the common add_parser() (and runner-specific do_add_parser() it delegates to), and provide a way to create instances of themselves from a RunnerConfig and parsed runner-specific arguments via create().

Runners use a variety of host tools and configuration values, the user interface to which is abstracted by this class. Each runner subclass should take any values it needs to execute one of these commands in its constructor. The actual command execution is handled in the run() method.

`classmethod add_parser`(*parser*)

Adds a sub-command parser for this runner.

The given object, parser, is a sub-command parser from the argparse module. For more details, refer to the documentation for argparse.ArgumentParser.add_subparsers().

The lone common optional argument is:

- –dt-flash (if the runner capabilities includes flash_addr)

Runner-specific options are added through the do_add_parser() hook.

`property build_conf:` *BuildConfiguration*

Get a BuildConfiguration for the build directory.

`call`(*cmd: List[str]*, ***kwargs*) → int

Subclass subprocess.call() wrapper.

Subclasses should use this method to run command in a subprocess and get its return code, rather than using subprocess directly, to keep accurate debug logs.

`classmethod capabilities`() → *RunnerCaps*

Returns a RunnerCaps representing this runner's capabilities.

This implementation returns the default capabilities.

Subclasses should override appropriately if needed.

`cfg`

RunnerConfig for this instance.

`check_call`(*cmd: List[str]*, ***kwargs*)

Subclass subprocess.check_call() wrapper.

Subclasses should use this method to run command in a subprocess and check that it executed correctly, rather than using subprocess directly, to keep accurate debug logs.

`check_output`(*cmd: List[str]*, ***kwargs*) → bytes

Subclass subprocess.check_output() wrapper.

Subclasses should use this method to run command in a subprocess and check that it executed correctly, rather than using subprocess directly, to keep accurate debug logs.

`classmethod create`(*cfg:* RunnerConfig, *args: Namespace*) → *ZephyrBinaryRunner*

Create an instance from command-line arguments.

- `cfg`: runner configuration (pass to superclass __init__)
- `args`: arguments parsed from execution environment, as specified by `add_parser()`.

`classmethod dev_id_help`() → str

Get the ArgParse help text for the –dev-id option.

`abstract classmethod do_add_parser`(*parser*)

Hook for adding runner-specific options.

`abstract classmethod do_create`(*cfg:* RunnerConfig, *args: Namespace*) → *ZephyrBinaryRunner*

Hook for instance creation from command line arguments.

`abstract do_run`(*command: str*, ***kwargs*)

Concrete runner; run() delegates to this. Implement in subclasses.

In case of an unsupported command, raise a ValueError.

`ensure_output`(*output_type: str*) → None

Ensure self.cfg has a particular output artifact.

For example, ensure_output('bin') ensures that self.cfg.bin_file refers to an existing file. Errors out if it's missing or undefined.

Parameters

`output_type` – string naming the output type

`classmethod extload_help`() → str

Get the ArgParse help text for the –extload option.

`static flash_address_from_build_conf`(*build_conf:* BuildConfiguration)

If CONFIG_HAS_FLASH_LOAD_OFFSET is n in build_conf, return the CONFIG_FLASH_BASE_ADDRESS value. Otherwise, return CONFIG_FLASH_BASE_ADDRESS + CONFIG_FLASH_LOAD_OFFSET.

`static get_flash_address`(*args: Namespace*, *build_conf:* BuildConfiguration, *default: int = 0*) → int

Helper method for extracting a flash address.

If args.dt_flash is true, returns the address obtained from ZephyrBinaryRunner.flash_address_from_build_conf(build_conf).

Otherwise (when args.dt_flash is False), the default value is returned.

`static get_runners`() → List[Type[*ZephyrBinaryRunner*]]

Get a list of all currently defined runner classes.

`logger`

logging.Logger for this instance.

`abstract classmethod name`() → str

Return this runner's user-visible name.

When choosing a name, pick something short and lowercase, based on the name of the tool (like openocd, jlink, etc.) or the target architecture/board (like xtensa etc.).

`popen_ignore_int`(*cmd: List[str]*, ***kwargs*) → Popen

Spawn a child command, ensuring it ignores SIGINT.

The returned subprocess.Popen object must be manually terminated.

`static require`(*program: str*, *path: str | None = None*) → str

Require that a program is installed before proceeding.

Parameters

- `program` – name of the program that is required, or path to a program binary.
- `path` – PATH where to search for the program binary. By default check on the system PATH.

If `program` is an absolute path to an existing program binary, this call succeeds. Otherwise, try to find the program by name on the system PATH or in the given PATH, if provided.

If the program can be found, its path is returned. Otherwise, raises MissingProgram.

`run`(*command: str*, ***kwargs*)

Runs command ('flash', 'debug', 'debugserver', 'attach').

This is the main entry point to this runner.

`run_client`(*client*, ***kwargs*)

Run a client that handles SIGINT.

`run_server_and_client`(*server*, *client*, ***kwargs*)

Run a server that ignores SIGINT, and a client that handles it.

This routine portably:

- creates a Popen object for the `server` command which ignores SIGINT
- runs `client` in a subprocess while temporarily ignoring SIGINT
- cleans up the server after the client exits.
- the keyword arguments, if any, will be passed down to both server and client subprocess calls

It's useful to e.g. open a GDB server and client.

`property sysbuild_conf: SysbuildConfiguration`

Get a SysbuildConfiguration for the sysbuild directory.

`property thread_info_enabled: bool`

Returns True if self.build_conf has CONFIG_DEBUG_THREAD_INFO enabled.

`classmethod tool_opt_help`() → str

Get the ArgParse help text for the –tool-opt option.

Doing it By Hand

If you prefer not to use West to flash or debug your board, simply inspect the build directory for the binaries output by the build system. These will be named something like `zephyr/zephyr.elf`, `zephyr/zephyr.hex`, etc., depending on your board's build system integration. These binaries may be flashed to a board using alternative tools of your choice, or used for debugging as needed, e.g. as a source of symbol tables.

By default, these West commands rebuild binaries before flashing and debugging. This can of course also be accomplished using the usual targets provided by Zephyr's build system (in fact, that's how these commands do it).

2.11.11 Signing Binaries

The `west sign` *extension* command can be used to sign a Zephyr application binary for consumption by a bootloader using an external tool. In some configurations, `west sign` is also used to invoke an external, post-processing tool that "stitches" the final components of the image together. Run `west sign -h` for command line help.

MCUboot / imgtool

The Zephyr build system has special support for signing binaries for use with the MCUboot bootloader using the imgtool program provided by its developers. You can both build and sign this type of application binary in one step by setting some Kconfig options. If you do, `west flash` will use the signed binaries.

If you use this feature, you don't need to run `west sign` yourself; the build system will do it for you.

Here is an example workflow, which builds and flashes MCUboot, as well as the hello_world application for chain-loading by MCUboot. Run these commands from the `zephyrproject` workspace you created in the *Getting Started Guide*.

```
west build -b YOUR_BOARD bootloader/mcuboot/boot/zephyr -d build-mcuboot
west build -b YOUR_BOARD zephyr/samples/hello_world -d build-hello-signed -- \
     -DCONFIG_BOOTLOADER_MCUBOOT=y \
     -DCONFIG_MCUBOOT_SIGNATURE_KEY_FILE=\"bootloader/mcuboot/root-rsa-2048.pem\"

west flash -d build-mcuboot
west flash -d build-hello-signed
```

Notes on the above commands:

- `YOUR_BOARD` should be changed to match your board
- The `CONFIG_MCUBOOT_SIGNATURE_KEY_FILE` value is the insecure default provided and used by MCUboot for development and testing
- You can change the `hello_world` application directory to any other application that can be loaded by MCUboot, such as the smp-svr sample.

For more information on these and other related configuration options, see:

- `CONFIG_BOOTLOADER_MCUBOOT`: build the application for loading by MCUboot
- `CONFIG_MCUBOOT_SIGNATURE_KEY_FILE`: the key file to use with `west sign`. If you have your own key, change this appropriately
- `CONFIG_MCUBOOT_EXTRA_IMGTOOL_ARGS`: optional additional command line arguments for `imgtool`
- `CONFIG_MCUBOOT_GENERATE_CONFIRMED_IMAGE`: also generate a confirmed image, which may be more useful for flashing in production environments than the OTA-able default image
- On Windows, if you get "Access denied" issues, the recommended fix is to run `pip3 install imgtool`, then retry with a pristine build directory.

If your `west flash` *runner* uses an image format supported by imgtool, you should see something like this on your device's serial console when you run `west flash -d build-mcuboot`:

```
*** Booting Zephyr OS build zephyr-v2.3.0-2310-gcebac69c8ae1  ***
[00:00:00.004,669] <inf> mcuboot: Starting bootloader
[00:00:00.011,169] <inf> mcuboot: Primary image: magic=unset, swap_type=0x1, copy_done=0x3,␣
↪image_ok=0x3
[00:00:00.021,636] <inf> mcuboot: Boot source: none
[00:00:00.027,313] <wrn> mcuboot: Failed reading image headers; Image=0
[00:00:00.035,064] <err> mcuboot: Unable to find bootable image
```

Then, you should see something like this when you run `west flash -d build-hello-signed`:

```
*** Booting Zephyr OS build zephyr-v2.3.0-2310-gcebac69c8ae1  ***
[00:00:00.004,669] <inf> mcuboot: Starting bootloader
[00:00:00.011,169] <inf> mcuboot: Primary image: magic=unset, swap_type=0x1, copy_done=0x3,␣
↪image_ok=0x3
```

(continues on next page)

(continued from previous page)

```
[00:00:00.021,636] <inf> mcuboot: Boot source: none
[00:00:00.027,374] <inf> mcuboot: Swap type: none
[00:00:00.115,142] <inf> mcuboot: Bootloader chainload address offset: 0xc000
[00:00:00.123,168] <inf> mcuboot: Jumping to the first image slot
*** Booting Zephyr OS build zephyr-v2.3.0-2310-gcebac69c8ae1  ***
Hello World! nrf52840dk_nrf52840
```

Whether `west flash` supports this feature depends on your runner. The `nrfjprog` and `pyocd` runners work with the above flow. If your runner does not support this flow and you would like it to, please send a patch or file an issue for adding support.

Extending signing externally

The signing script used when running `west flash` can be extended or replaced to change features or introduce different signing mechanisms. By default with MCUboot enabled, signing is setup by the `cmake/mcuboot.cmake` file in Zephyr which adds extra post build commands for generating the signed images. The file used for signing can be replaced from a sysbuild scope (if being used) or from a zephyr/zephyr module scope, the priority of which is:

- Sysbuild
- Zephyr property
- Default MCUboot script (if enabled)

From sysbuild, `-D<target>_SIGNING_SCRIPT` can be used to set a signing script for a specific image or `-DSIGNING_SCRIPT` can be used to set a signing script for all images, for example:

```
west build -b <board> <application> -DSIGNING_SCRIPT=<file>
```

The zephyr property method is achieved by adjusting the `SIGNING_SCRIPT` property on the *zephyr_property_target*, ideally from by a module by using:

```
if(CONFIG_BOOTLOADER_MCUBOOT)
  set_target_properties(zephyr_property_target PROPERTIES SIGNING_SCRIPT ${CMAKE_CURRENT_
↪LIST_DIR}/custom_signing.cmake)
endif()
```

This will include the custom signing CMake file instead of the default Zephyr one when projects are built with MCUboot signing support enabled. The base Zephyr MCUboot signing file can be used as a reference for creating a new signing system or extending the default behaviour.

rimage

rimage configuration uses a different approach that does not rely on Kconfig or CMake but on *west config* instead, similar to *Permanent CMake Arguments*.

Signing involves a number of "wrapper" scripts stacked on top of each other: `west flash` invokes `west build` which invokes `cmake` and `ninja` which invokes `west sign` which invokes `imgtool` or `rimage`. As long as the signing parameters desired are the default ones and fairly static, these indirections are not a problem. On the other hand, passing `imgtool` or `rimage` options through all these layers can causes issues typical when the layers don't abstract anything. First, this usually requires boilerplate code in each layer. Quoting whitespace or other special characters through all the wrappers can be difficult. Reproducing a lower `west sign` command to debug some build-time issue can be very time-consuming: it requires at least enabling and searching verbose build logs to find which exact options were used. Copying these options from the build logs can be unreliable: it may produce different results because of subtle environment differences. Last and worst: new signing feature and options are impossible to use until more boilerplate code has been added in each layer.

To avoid these issues, `rimage` parameters can bet set in `west config` instead. Here's a `workspace/.west/config` example:

```
[sign]
# Not needed when invoked from CMake
tool = rimage

[rimage]
# Quoting is optional and works like in Unix shells
# Not needed when rimage can be found in the default PATH
path = "/home/me/zworkspace/build-rimage/rimage"

# Not needed when using the default development key
extra-args = -i 4 -k 'keys/key argument with space.pem'
```

In order to support quoting, values are parsed by Python's `shlex.split()` like in *One-Time CMake Arguments*.

The `extra-args` are passed directly to the `rimage` command. The example above has the same effect as appending them on command line after `--` like this: `west sign --tool rimage -- -i 4 -k 'keys/key argument with space.pem'`. In case both are used, the command-line arguments go last.

2.11.12 Additional Zephyr extension commands

This page documents miscellaneous *Zephyr Extensions*.

Listing boards: `west boards`

The `boards` command can be used to list the boards that are supported by Zephyr without having to resort to additional sources of information.

It can be run by typing:

```
west boards
```

This command lists all supported boards in a default format. If you prefer to specify the display format yourself you can use the `--format` (or `-f`) flag:

```
west boards -f "{arch}:{name}"
```

Additional help about the formatting options can be found by running:

```
west boards -h
```

Shell completion scripts: `west completion`

The `completion` extension command outputs shell completion scripts that can then be used directly to enable shell completion for the supported shells.

It currently supports the following shells:

- bash
- zsh
- fish

Additional instructions are available in the command's help:

```
west help completion
```

Installing CMake packages: `west zephyr-export`

This command registers the current Zephyr installation as a CMake config package in the CMake user package registry.

In Windows, the CMake user package registry is found in `HKEY_CURRENT_USER\Software\Kitware\CMake\Packages`.

In Linux and MacOS, the CMake user package registry is found in. `~/.cmake/packages`.

You may run this command when setting up a Zephyr workspace. If you do, application CMakeLists.txt files that are outside of your workspace will be able to find the Zephyr repository with the following:

```
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
```

See share/zephyr-package/cmake for details.

Software bill of materials: `west spdx`

This command generates SPDX 2.3 tag-value documents, creating relationships from source files to the corresponding generated build files. `SPDX-License-Identifier` comments in source files are scanned and filled into the SPDX documents.

To use this command:

1. Pre-populate a build directory `BUILD_DIR` like this:

   ```
   west spdx --init -d BUILD_DIR
   ```

   This step ensures the build directory contains CMake metadata required for SPDX document generation.

2. Enable `CONFIG_BUILD_OUTPUT_META` in your project.

3. Build your application using this pre-created build directory, like so:

   ```
   west build -d BUILD_DIR [...]
   ```

4. Generate SPDX documents using this build directory:

   ```
   west spdx -d BUILD_DIR
   ```

This generates the following SPDX bill-of-materials (BOM) documents in `BUILD_DIR/spdx/`:

- `app.spdx`: BOM for the application source files used for the build
- `zephyr.spdx`: BOM for the specific Zephyr source code files used for the build
- `build.spdx`: BOM for the built output files
- `modules-deps.spdx`: BOM for modules dependencies. Check *modules* for more details.

Each file in the bill-of-materials is scanned, so that its hashes (SHA256 and SHA1) can be recorded, along with any detected licenses if an `SPDX-License-Identifier` comment appears in the file.

SPDX Relationships are created to indicate dependencies between CMake build targets, build targets that are linked together, and source files that are compiled to generate the built library files.

`west spdx` accepts these additional options:

- `-n PREFIX`: a prefix for the Document Namespaces that will be included in the generated SPDX documents. See SPDX specification clause 6 for details. If `-n` is omitted, a default namespace will be generated according to the default format described in section 2.5 using a random UUID.
- `-s SPDX_DIR`: specifies an alternate directory where the SPDX documents should be written instead of `BUILD_DIR/spdx/`.
- `--analyze-includes`: in addition to recording the compiled source code files (e.g. `.c`, `.S`) in the bills-of-materials, also attempt to determine the specific header files that are included for each `.c` file.

  This takes longer, as it performs a dry run using the C compiler for each `.c` file using the same arguments that were passed to it for the actual build.
- `--include-sdk`: with `--analyze-includes`, also create a fourth SPDX document, `sdk.spdx`, which lists header files included from the SDK.

Working with binary blobs: `west blobs`

The `blobs` command allows users to interact with *binary blobs* declared in one or more *modules* via their *module.yml* file.

The `blobs` command has three sub-commands, used to list, fetch or clean (i.e. delete) the binary blobs themselves.

You can list binary blobs while specifying the format of the output:

```
west blobs list -f '{module}: {type} {path}'
```

For the full set of variables available in `-f/--format` run `west blobs -h`.

Fetching blobs works in a similar manner:

```
west blobs fetch
```

Note that, as described in *the modules section*, fetched blobs are stored in a `zephyr/blobs/` folder relative to the root of the corresponding module repository.

As does deleting them:

```
west blobs clean
```

Additionally the tool allows you to specify the modules you want to list, fetch or clean blobs for by typing the module names as a command-line parameter.

Twister wrapper: `west twister`

This command is a wrapper for *twister*.

Twister can then be invoked via west as follows:

```
west twister -help
west twister -T tests/ztest/base
```

Working with binary descriptors: `west bindesc`

The `bindesc` command allows users to read *binary descriptors* of executable files. It currently supports `.bin`, `.hex`, `.elf` and `.uf2` files as input.

You can search for a specific descriptor in an image, for example:

```
west bindesc search KERNEL_VERSION_STRING build/zephyr/zephyr.bin
```

You can search for a custom descriptor by type and ID, for example:

```
west bindesc custom_search STR 0x200 build/zephyr/zephyr.bin
```

You can dump all of the descriptors in an image using:

```
west bindesc dump build/zephyr/zephyr.bin
```

You can list all known standard descriptor names using:

```
west bindesc list
```

2.11.13 History and Motivation

West was added to the Zephyr project to fulfill two fundamental requirements:

- The ability to work with multiple Git repositories
- The ability to provide an extensible and user-friendly command-line interface for basic Zephyr workflows

During the development of west, a set of *Design Constraints* were identified to avoid the common pitfalls of tools of this kind.

Requirements

Although the motivation behind splitting the Zephyr codebase into multiple repositories is outside of the scope of this page, the fundamental requirements, along with a clear justification of the choice not to use existing tools and instead develop a new one, do belong here.

The basic requirements are:

- **R1**: Keep externally maintained code in separately maintained repositories outside of the main zephyr repository, without requiring users to manually clone each of the external repositories
- **R2**: Provide a tool that both Zephyr users and distributors can make use of to benefit from and extend
- **R3**: Allow users and downstream distributions to override or remove repositories without having to make changes to the zephyr repository
- **R4**: Support both continuous tracking and commit-based (bisectable) project updating

Rationale for a custom tool

Some of west's features are similar to those provided by Git Submodules and Google's repo.

Existing tools were considered during west's initial design and development. None were found suitable for Zephyr's requirements. In particular, these were examined in detail:

- Google repo
  - Does not cleanly support using zephyr as the manifest repository (**R4**)
  - Python 2 only
  - Does not play well with Windows
  - Assumes Gerrit is used for code review

- Git submodules
  - Does not fully support **R1**, since the externally maintained repositories would still need to be inside the main zephyr Git tree
  - Does not support **R3**, since downstream copies would need to either delete or replace submodule definitions
  - Does not support continuous tracking of the latest `HEAD` in external repositories **(R4)**
  - Requires hardcoding of the paths/locations of the external repositories

Multiple Git Repositories

Zephyr intends to provide all required building blocks needed to deploy complex IoT applications. This in turn means that the Zephyr project is much more than an RTOS kernel, and is instead a collection of components that work together. In this context, there are a few reasons to work with multiple Git repositories in a standardized manner within the project:

- Clean separation of Zephyr original code and imported projects and libraries
- Avoidance of license incompatibilities between original and imported code
- Reduction in size and scope of the core Zephyr codebase, with additional repositories containing optional components instead of being imported directly into the tree
- Safety and security certifications
- Enforcement of modularization of the components
- Out-of-tree development based on subsets of the supported boards and SoCs

See *Basics* for information on how west workspaces manage multiple git repositories.

Design Constraints

West is:

- **Optional**: it is always *possible* to drop back to "raw" command-line tools, i.e. use Zephyr without using west (although west itself might need to be installed and accessible to the build system). It may not always be *convenient* to do so, however. (If all of west's features were already conveniently available, there would be no reason to develop it.)
- **Compatible with CMake**: building, flashing and debugging, and emulator support will always remain compatible with direct use of CMake.
- **Cross-platform**: West is written in Python 3, and works on all platforms supported by Zephyr.
- **Usable as a Library**: whenever possible, west features are implemented as libraries that can be used standalone in other programs, along with separate command line interfaces that wrap them. West itself is a Python package named `west`; its libraries are implemented as subpackages.
- **Conservative about features**: no features will be accepted without strong and compelling motivation.
- **Clearly specified**: West's behavior in cases where it wraps other commands is clearly specified and documented. This enables interoperability with third party tools, and means Zephyr developers can always find out what is happening "under the hood" when using west.

See Zephyr issue #6205 and for more details and discussion.

2.11.14 Moving to West

To convert a "pre-west" Zephyr setup on your computer to west, follow these steps. If you are starting from scratch, use the *Getting Started Guide* instead. See *Troubleshooting West* for advice on common issues.

1. Install west.

   On Linux:

   ```
   pip3 install --user -U west
   ```

   On Windows and macOS:

   ```
   pip3 install -U west
   ```

   For details, see *Installing west*.

2. Move your zephyr repository to a new `zephyrproject` parent directory, and change directory there.

   On Linux and macOS:

   ```
   mkdir zephyrproject
   mv zephyr zephyrproject
   cd zephyrproject
   ```

   On Windows `cmd.exe`:

   ```
   mkdir zephyrproject
   move zephyr zephyrproject
   chdir zephyrproject
   ```

   The name `zephyrproject` is recommended, but you can choose any name with no spaces anywhere in the path.

3. Create a *west workspace* using the zephyr repository as a local manifest repository:

   ```
   west init -l zephyr
   ```

   This creates `zephyrproject/.west`, marking the root of your workspace, and does some other setup. It will not change the contents of the zephyr repository in any way.

4. Clone the rest of the repositories used by zephyr:

   ```
   west update
   ```

   Make sure to run this command whenever you pull zephyr. Otherwise, your local repositories will get out of sync. (Run `west list` for current information on these repositories.)

You are done: `zephyrproject` is now set up to use west.

2.11.15 Using Zephyr without west

This page provides information on using Zephyr without west. This is not recommended for beginners due to the extra effort involved. In particular, you will have to do work "by hand" to replace these features:

- cloning the additional source code repositories used by Zephyr in addition to the main zephyr repository, and keeping them up to date
- specifying the locations of these repositories to the Zephyr build system
- flashing and debugging without understanding detailed usage of the relevant host tools

Note

If you have previously installed west and want to stop using it, uninstall it first:

```
pip3 uninstall west
```

Otherwise, Zephyr's build system will find it and may try to use it.

Getting the Source

In addition to downloading the zephyr source code repository itself, you will need to manually clone the additional projects listed in the *west manifest* file inside that repository.

```
mkdir zephyrproject
cd zephyrproject
git clone https://github.com/zephyrproject-rtos/zephyr
# clone additional repositories listed in zephyr/west.yml,
# and check out the specified revisions as well.
```

As you pull changes in the zephyr repository, you will also need to maintain those additional repositories, adding new ones as necessary and keeping existing ones up to date at the latest revisions.

Building applications

You can build a Zephyr application using CMake and Ninja (or make) directly without west installed if you specify any modules manually.

```
cmake -Bbuild -GNinja -DZEPHYR_MODULES=module1;module2;... samples/hello_world
ninja -Cbuild
```

When building with west installed, the Zephyr build system will use it to set *ZEPHYR_MODULES*.

If you don't have west installed and your application does not need any of these repositories, the build will still work.

If you don't have west installed and your application *does* need one of these repositories, you must set **ZEPHYR_MODULES** yourself as shown above.

See *Modules (External projects)* for more details.

Similarly, if your application requires binary blobs and you are not using west, you will need to download and place those blobs in the right places instead of using `west blobs`. See *Binary Blobs* for more details.

Flashing and Debugging

Running build system targets like `ninja flash`, `ninja debug`, etc. is just a call to the corresponding *west command*. For example, `ninja flash` calls `west flash`[1]. If you don't have west installed on your system, running those targets will fail. You can of course still flash and debug using any *Flash & Debug Host Tools* which work for your board (and which those west commands wrap).

[1] Note that `west build` invokes `ninja`, among other tools. There's no recursive invocation of either `west` or `ninja` involved by default, however, as `west build` does not invoke `ninja flash`, `debug`, etc. The one exception is if you specifically run one of these build system targets with a command line like `west build -t flash`. In that case, west is run twice: once for `west build`, and in a subprocess, again for `west flash`. Even in this case, `ninja` is only run once, as `ninja flash`. This is because these build system targets depend on an up to date build of the Zephyr application, so it's compiled before `west flash` is run.

If you want to use these build system targets but do not want to install west on your system using `pip`, it is possible to do so by manually creating a *west workspace*:

```
# cd into zephyrproject if not already there
git clone https://github.com/zephyrproject-rtos/west.git .west/west
```

Then create a file `.west/config` with the following contents:

```
[manifest]
path = zephyr

[zephyr]
base = zephyr
```

After that, and in order for `ninja` to be able to invoke `west` to flash and debug, you must specify the west directory. This can be done by setting the environment variable `WEST_DIR` to point to `zephyrproject/.west/west` before running CMake to set up a build directory.

For details on west's Python APIs, see west-apis.

2.12 Testing

2.12.1 Test Framework

The Zephyr Test Framework (Ztest) provides a simple testing framework intended to be used during development. It provides basic assertion macros and a generic test structure.

The framework can be used in two ways, either as a generic framework for integration testing, or for unit testing specific modules.

Creating a test suite

Using Ztest to create a test suite is as easy as calling the *ZTEST_SUITE*. The macro accepts the following arguments:

- `suite_name` - The name of the suite. This name must be unique within a single binary.
- *ztest_suite_predicate_t* - An optional predicate function to allow choosing when the test will run. The predicate will get a pointer to the global state passed in through *ztest_run_all()* and should return a boolean to decide if the suite should run.
- *ztest_suite_setup_t* - An optional setup function which returns a test fixture. This will be called and run once per test suite run.
- *ztest_suite_before_t* - An optional before function which will run before every single test in this suite.
- *ztest_suite_after_t* - An optional after function which will run after every single test in this suite.
- *ztest_suite_teardown_t* - An optional teardown function which will run at the end of all the tests in the suite.

Below is an example of a test suite using a predicate:

```
#include <zephyr/ztest.h>
#include "test_state.h"

static bool predicate(const void *global_state)
{
```

(continues on next page)

(continued from previous page)

```
     return ((const struct test_state*)global_state)->x == 5;
}

ZTEST_SUITE(alternating_suite, predicate, NULL, NULL, NULL, NULL);
```

Adding tests to a suite

There are 4 macros used to add a test to a suite, they are:

- *ZTEST* (suite_name, test_name) - Which can be used to add a test by test_name to a given suite by suite_name.
- *ZTEST_USER* (suite_name, test_name) - Which behaves the same as *ZTEST*, only that when CONFIG_USERSPACE is enabled, then the test will be run in a userspace thread.
- *ZTEST_F* (suite_name, test_name) - Which behaves the same as *ZTEST*, only that the test function will already include a variable named fixture with the type <suite_name>_fixture.
- *ZTEST_USER_F* (suite_name, test_name) - Which combines the fixture feature of *ZTEST_F* with the userspace threading for the test.

Test fixtures Test fixtures can be used to help simplify repeated test setup operations. In many cases, tests in the same suite will require some initial setup followed by some form of reset between each test. This is achieved via fixtures in the following way:

```
#include <zephyr/ztest.h>

struct my_suite_fixture {
     size_t max_size;
     size_t size;
     uint8_t buff[1];
};

static void *my_suite_setup(void)
{
     /* Allocate the fixture with 256 byte buffer */
   struct my_suite_fixture *fixture = malloc(sizeof(struct my_suite_fixture) + 255);

     zassume_not_null(fixture, NULL);
     fixture->max_size = 256;

     return fixture;
}

static void my_suite_before(void *f)
{
     struct my_suite_fixture *fixture = (struct my_suite_fixture *)f;
     memset(fixture->buff, 0, fixture->max_size);
     fixture->size = 0;
}

static void my_suite_teardown(void *f)
{
   free(f);
}

ZTEST_SUITE(my_suite, NULL, my_suite_setup, my_suite_before, NULL, my_suite_teardown);
```

(continues on next page)

(continued from previous page)

```
ZTEST_F(my_suite, test_feature_x)
{
     zassert_equal(0, fixture->size);
     zassert_equal(256, fixture->max_size);
}
```

Using memory allocated by a test fixture in a userspace thread, such as during execution of *ZTEST_USER* or *ZTEST_USER_F*, requires that memory to be declared userspace accessible. This is because the fixture memory is owned and initialized by kernel space. The Ztest framework provides the *ZTEST_DMEM* and *ZTEST_BMEM* macros for use of such user/kernel space shared memory.

Advanced features

Test result expectations Some tests were made to be broken. In cases where the test is expected to fail or skip due to the nature of the code, it's possible to annotate the test as such. For example:

```
#include <zephyr/ztest.h>

ZTEST_SUITE(my_suite, NULL, NULL, NULL, NULL, NULL);

ZTEST_EXPECT_FAIL(my_suite, test_fail);
ZTEST(my_suite, test_fail)
{
  /** This will fail the test */
  zassert_true(false, NULL);
}

ZTEST_EXPECT_SKIP(my_suite, test_skip);
ZTEST(my_suite, test_skip)
{
  /** This will skip the test */
  zassume_true(false, NULL);
}
```

In this example, the above tests should be marked as failed and skipped respectively. Instead, Ztest will mark both as passed due to the expectation.

Test rules Test rules are a way to run the same logic for every test and every suite. There are a lot of cases where you might want to reset some state for every test in the binary (regardless of which suite is currently running). As an example, this could be to reset mocks, reset emulators, flush the UART, etc.:

```
#include <zephyr/fff.h>
#include <zephyr/ztest.h>

#include "test_mocks.h"

DEFINE_FFF_GLOBALS;

DEFINE_FAKE_VOID_FUN(my_weak_func);

static void fff_reset_rule_before(const struct ztest_unit_test *test, void *fixture)
{
     ARG_UNUSED(test);
     ARG_UNUSED(fixture);

     RESET_FAKE(my_weak_func);
```

(continues on next page)

(continued from previous page)

```
}

ZTEST_RULE(fff_reset_rule, fff_reset_rule_before, NULL);
```

A custom `test_main` While the Ztest framework provides a default `test_main()` function, it's possible that some applications will want to provide custom behavior. This is particularly true if there's some global state that the tests depend on and that state either cannot be replicated or is difficult to replicate without starting the process over. For example, one such state could be a power sequence. Assuming there's a board with several steps in the power-on sequence a test suite can be written using the `predicate` to control when it would run. In that case, the `test_main()` function can be written as follows:

```
#include <zephyr/ztest.h>

#include "my_test.h"

void test_main(void)
{
     struct power_sequence_state state;

     /* Only suites that use a predicate checking for phase == PWR_PHASE_0 will run. */
     state.phase = PWR_PHASE_0;
     ztest_run_all(&state, false, 1, 1);

     /* Only suites that use a predicate checking for phase == PWR_PHASE_1 will run. */
     state.phase = PWR_PHASE_1;
     ztest_run_all(&state, false, 1, 1);

     /* Only suites that use a predicate checking for phase == PWR_PHASE_2 will run. */
     state.phase = PWR_PHASE_2;
     ztest_run_all(&state, false, 1, 1);

     /* Check that all the suites in this binary ran at least once. */
     ztest_verify_all_test_suites_ran();
}
```

Quick start - Integration testing

A simple working base is located at samples/subsys/testsuite/integration. To make a test application for the **bar** component of **foo**, you should copy the sample folder to `tests/foo/bar` and edit files there adjusting for your test application's purposes.

To build and execute all applicable test scenarios defined in your test application use the *Twister* tool, for example:

```
./scripts/twister -T tests/foo/bar/
```

To select just one of the test scenarios, run Twister with `--scenario` command:

```
./scripts/twister --scenario tests/foo/bar/your.test.scenario.name
```

In the command line above `tests/foo/bar` is the path to your test application and `your.test.scenario.name` references a test scenario defined in `testcase.yaml` file, which is like `sample.testing.ztest` in the boilerplate test suite sample.

See *Twister test project diagram* for more details on how Twister deals with Ztest application.

The sample contains the following files:

CMakeLists.txt

```
# SPDX-License-Identifier: Apache-2.0

cmake_minimum_required(VERSION 3.20.0)
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
project(integration)

FILE(GLOB app_sources src/*.c)
target_sources(app PRIVATE ${app_sources})
```

testcase.yaml

```
tests:
  # section.subsection
  sample.testing.ztest:
    build_only: true
    platform_allow:
      - native_posix
      - native_sim
    integration_platforms:
      - native_sim
    tags: test_framework
```

prj.conf

```
CONFIG_ZTEST=y
```

src/main.c (see *best practices*)

```
/*
 * Copyright (c) 2016 Intel Corporation
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <zephyr/ztest.h>


ZTEST_SUITE(framework_tests, NULL, NULL, NULL, NULL, NULL);

/**
 * @brief Test Asserts
 *
 * This test verifies various assert macros provided by ztest.
 *
 */
ZTEST(framework_tests, test_assert)
{
        zassert_true(1, "1 was false");
        zassert_false(0, "0 was true");
        zassert_is_null(NULL, "NULL was not NULL");
        zassert_not_null("foo", "\"foo\" was NULL");
        zassert_equal(1, 1, "1 was not equal to 1");
        zassert_equal_ptr(NULL, NULL, "NULL was not equal to NULL");
}
```

- *Listing Tests*
- *Skipping Tests*

A test application may consist of multiple test suites that either can be testing functionality or APIs. Functions implementing a test case should follow the guidelines below:

- Test cases function names should be prefixed with **test_**
- Test cases should be documented using doxygen
- Test case function names should be unique within the section or component being tested

For example:

```
/**
 * @brief Test Asserts
 *
 * This test case verifies the zassert_true macro.
 */
ZTEST(my_suite, test_assert)
{
        zassert_true(1, "1 was false");
}
```

Listing Tests Tests (test applications) in the Zephyr tree consist of many test scenarios that run as part of a project and test similar functionality, for example an API or a feature. The `twister` script can parse the test scenarios, suites and cases in all test applications or a subset of them, and can generate reports on a granular level, i.e. if test cases have passed or failed or if they were blocked or skipped.

Twister parses the source files looking for test case names, so you can list all kernel test cases, for example, by running:

```
./scripts/twister --list-tests -T tests/kernel
```

Skipping Tests Special- or architecture-specific tests cannot run on all platforms and architectures, however we still want to count those and report them as being skipped. Because the test inventory and the list of tests is extracted from the code, adding conditionals inside the test suite is sub-optimal. Tests that need to be skipped for a certain platform or feature need to explicitly report a skip using *ztest_test_skip()* or `Z_TEST_SKIP_IFDEF`. If the test runs, it needs to report either a pass or fail. For example:

```
#ifdef CONFIG_TEST1
ZTEST(common, test_test1)
{
     zassert_true(1, "true");
}
#else
ZTEST(common, test_test1)
{
     ztest_test_skip();
}
#endif

ZTEST(common, test_test2)
{
     Z_TEST_SKIP_IFDEF(CONFIG_BUGxxxxx);
     zassert_equal(1, 0, NULL);
}

ZTEST_SUITE(common, NULL, NULL, NULL, NULL, NULL);
```

Quick start - Unit testing

Ztest can be used for unit testing. This means that rather than including the entire Zephyr OS for testing a single function, you can focus the testing efforts into the specific module in question. This will speed up testing since only the module will have to be compiled in, and the tested functions will be called directly.

Examples of unit tests can be found in the tests/unit/ folder. In order to declare the unit tests present in a source folder, you need to add the relevant source files to the `testbinary` target from the CMake unittest component. See a minimal example below:

```
cmake_minimum_required(VERSION 3.20.0)

project(app)
find_package(Zephyr COMPONENTS unittest REQUIRED HINTS $ENV{ZEPHYR_BASE})
target_sources(testbinary PRIVATE main.c)
```

Since you won't be including basic kernel data structures that most code depends on, you have to provide function stubs in the test. Ztest provides some helpers for mocking functions, as demonstrated below.

In a unit test, mock objects can simulate the behavior of complex real objects and are used to decide whether a test failed or passed by verifying whether an interaction with an object occurred, and if required, to assert the order of that interaction.

Best practices for declaring the test suite

twister and other validation tools need to obtain the list of test cases that a Zephyr *ztest* test image will expose.

Rationale

This all is for the purpose of traceability. It's not enough to have only a semaphore test application. We also need to show that we have testpoints for all APIs and functionality, and we trace back to documentation of the API, and functional requirements.

The idea is that test reports show results for every test case as passed, failed, blocked, or skipped. Reporting on only the high-level test application, particularly when tests do too many things, is too vague.

Other questions:

- Why not pre-scan with CPP and then parse? or post scan the ELF file?

  If C pre-processing or building fails because of any issue, then we won't be able to tell the subcases.

- Why not declare them in the YAML test configuration?

  A separate test case description file would be harder to maintain than just keeping the information in the test source files themselves – only one file to update when changes are made eliminates duplication.

Stress test framework

Zephyr stress test framework (Ztress) provides an environment for executing user functions in multiple priority contexts. It can be used to validate that code is resilient to preemptions. The framework tracks the number of executions and preemptions for each context. Execution can

have various completion conditions like timeout, number of executions or number of preemptions.

The framework is setting up the environment by creating the requested number of threads (each on different priority), optionally starting a timer. For each context, a user function (different for each context) is called and then the context sleeps for a randomized amount of system ticks. The framework is tracking CPU load and adjusts sleeping periods to achieve higher CPU load. In order to increase the probability of preemptions, the system clock frequency should be relatively high. The default 100 Hz on QEMU x86 is much too low and it is recommended to increase it to 100 kHz.

The stress test environment is setup and executed using *ZTRESS_EXECUTE* which accepts a variable number of arguments. Each argument is a context that is specified by *ZTRESS_TIMER* or *ZTRESS_THREAD* macros. Contexts are specified in priority descending order. Each context specifies completion conditions by providing the minimum number of executions and preemptions. When all conditions are met and the execution has completed, an execution report is printed and the macro returns. Note that while the test is executing, a progress report is periodically printed.

Execution can be prematurely completed by specifying a test timeout (*ztress_set_timeout()*) or an explicit abort (*ztress_abort()*).

User function parameters contains an execution counter and a flag indicating if it is the last execution.

The example below presents how to setup and run 3 contexts (one of which is k_timer interrupt handler context). Completion criteria is set to at least 10000 executions of each context and 1000 preemptions of the lowest priority context. Additionally, the timeout is configured to complete after 10 seconds if those conditions are not met. The last argument of each context is the initial sleep time which will be adjusted throughout the test to achieve the highest CPU load.

```
ztress_set_timeout(K_MSEC(10000));
ZTRESS_EXECUTE(ZTRESS_TIMER(foo_0, user_data_0, 10000, Z_TIMEOUT_TICKS(20)),
               ZTRESS_THREAD(foo_1, user_data_1, 10000, 0, Z_TIMEOUT_TICKS(20)),
               ZTRESS_THREAD(foo_2, user_data_2, 10000, 1000, Z_TIMEOUT_
↪TICKS(20)));
```

Configuration Static configuration of Ztress contains:

- CONFIG_ZTRESS_MAX_THREADS - number of supported threads.
- CONFIG_ZTRESS_STACK_SIZE - Stack size of created threads.
- CONFIG_ZTRESS_REPORT_PROGRESS_MS - Test progress report interval.

API reference

Running tests

group **ztest_test**

This module eases the testing process by providing helpful macros and other testing structures.

Defines

ZTEST_EXPECT_FAIL(_suite_name, _test_name)

Expect a test to fail (mark it passing if it failed)

Adding this macro to your logic will allow the failing test to be considered passing, example:

```
ZTEST_EXPECT_FAIL(my_suite, test_x);
ZTEST(my_suite, text_x) {
  zassert_true(false, NULL);
}
```

Parameters

- **_suite_name** – The name of the suite
- **_test_name** – The name of the test

ZTEST_EXPECT_SKIP(_suite_name, _test_name)

Expect a test to skip (mark it passing if it failed)

Adding this macro to your logic will allow the failing test to be considered passing, example:

```
ZTEST_EXPECT_SKIP(my_suite, test_x);
ZTEST(my_suite, text_x) {
  zassume_true(false, NULL);
}
```

Parameters

- **_suite_name** – The name of the suite
- **_test_name** – The name of the test

ZTEST_TEST_COUNT

Number of registered unit tests.

ZTEST_SUITE_COUNT

Number of registered test suites.

ZTEST_SUITE(SUITE_NAME, PREDICATE, setup_fn, before_fn, after_fn, teardown_fn)

Create and register a ztest suite.

Using this macro creates a new test suite. It then creates a struct *ztest_suite_node* in a specific linker section.

Tests can then be run by calling ztest_run_test_suites(const void *state) by passing in the current state. See the documentation for ztest_run_test_suites for more info.

Parameters

- **SUITE_NAME** – The name of the suite
- **PREDICATE** – A function to test against the state and determine if the test should run.
- **setup_fn** – The setup function to call before running this test suite
- **before_fn** – The function to call before each unit test in this suite
- **after_fn** – The function to call after each unit test in this suite
- **teardown_fn** – The function to call after running all the tests in this suite

ZTEST_DMEM

Make data section used by Ztest userspace accessible.

ZTEST_BMEM
: Make bss section used by Ztest userspace accessible.

ZTEST_SECTION
: Ztest data section for accessing data from userspace.

ZTEST(suite, fn)
: Create and register a new unit test.

  Calling this macro will create a new unit test and attach it to the declared `suite`. The `suite` does not need to be defined in the same compilation unit.

  Parameters

  - **suite** – The name of the test suite to attach this test
  - **fn** – The test function to call.

ZTEST_USER(suite, fn)
: Define a test function that should run as a user thread.

  This macro behaves exactly the same as ZTEST, but calls the test function in user space if `CONFIG_USERSPACE` was enabled.

  Parameters

  - **suite** – The name of the test suite to attach this test
  - **fn** – The test function to call.

ZTEST_F(suite, fn)
: Define a test function.

  This macro behaves exactly the same as *ZTEST()*, but the function takes an argument for the fixture of type `struct suite##_fixture*` named `fixture`.

  Parameters

  - **suite** – The name of the test suite to attach this test
  - **fn** – The test function to call.

ZTEST_USER_F(suite, fn)
: Define a test function that should run as a user thread.

  If CONFIG_USERSPACE is not enabled, this is functionally identical to *ZTEST_F()*. The test function takes a single fixture argument of type `struct suite##_fixture*` named `fixture`.

  Parameters

  - **suite** – The name of the test suite to attach this test
  - **fn** – The test function to call.

ZTEST_RULE(name, before_each_fn, after_each_fn)
: Define a test rule that will run before/after each unit test.

  Functions defined here will run before/after each unit test for every test suite. Along with the callback, the test functions are provided a pointer to the test being run, and the data. This provides a mechanism for tests to perform custom operations depending on the specific test or the data (for example logging may use the test's name).

  Ordering:

  - Test rule's `before` function will run before the suite's `before` function. This is done to allow the test suite's customization to take precedence over the rule which is applied to all suites.

- Test rule's `after` function is not guaranteed to run in any particular order.

Parameters

- **name** – The name for the test rule (must be unique within the compilation unit)
- **before_each_fn** – The callback function (ztest_rule_cb) to call before each test (may be NULL)
- **after_each_fn** – The callback function (ztest_rule_cb) to call after each test (may be NULL)

ztest_run_test_suite(suite, shuffle, suite_iter, case_iter)

Run the specified test suite.

Parameters

- **suite** – Test suite to run.
- **shuffle** – Shuffle tests
- **suite_iter** – Test suite repetitions.
- **case_iter** – Test case repetitions.

Typedefs

typedef void *(***ztest_suite_setup_t**)(void)

Setup function to run before running this suite.

Return
Pointer to the data structure that will be used throughout this test suite

typedef void (***ztest_suite_before_t**)(void *fixture)

Function to run before each test in this suite.

Param fixture
The test suite's fixture returned from setup()

typedef void (***ztest_suite_after_t**)(void *fixture)

Function to run after each test in this suite.

Param fixture
The test suite's fixture returned from setup()

typedef void (***ztest_suite_teardown_t**)(void *fixture)

Teardown function to run after running this suite.

Param fixture
The test suite's data returned from setup()

typedef bool (***ztest_suite_predicate_t**)(const void *global_state)

An optional predicate function to determine if the test should run.

If NULL, then the test will only run once on the first attempt.

Param global_state
The current state of the test application.

Return
True if the suite should be run; false to skip.

typedef void (*`ztest_rule_cb`)(const struct *ztest_unit_test* *test, void *data)

Test rule callback function signature.

The function signature that can be used to register a test rule's before/after callback. This provides access to the test and the fixture data (if provided).

Param test
Pointer to the unit test in context

Param data
Pointer to the test's fixture data (may be NULL)

Enums

enum `ztest_expected_result`

The expected result of a test.

See also

ZTEST_EXPECT_FAIL

See also

ZTEST_EXPECT_SKIP

Values:

enumerator `ZTEST_EXPECTED_RESULT_FAIL` = 0

Expect a test to fail.

enumerator `ZTEST_EXPECTED_RESULT_SKIP`

Expect a test to pass.

enum `ztest_result`

The result of the current running test.

It's possible that the setup function sets the result to ZTEST_RESULT_SUITE_* which will apply the failure/skip to every test in the suite.

Values:

enumerator `ZTEST_RESULT_PENDING`

enumerator `ZTEST_RESULT_PASS`

enumerator `ZTEST_RESULT_FAIL`

enumerator ZTEST_RESULT_SKIP

enumerator ZTEST_RESULT_SUITE_SKIP

enumerator ZTEST_RESULT_SUITE_FAIL

enum ztest_phase

Each enum member represents a distinct phase of execution for the test binary.

TEST_PHASE_FRAMEWORK is active when internal ztest code is executing; the rest refer to corresponding phases of user test code.

Values:

enumerator TEST_PHASE_SETUP

enumerator TEST_PHASE_BEFORE

enumerator TEST_PHASE_TEST

enumerator TEST_PHASE_AFTER

enumerator TEST_PHASE_TEARDOWN

enumerator TEST_PHASE_FRAMEWORK

Functions

void ztest_run_all(const void *state, bool shuffle, int suite_iter, int case_iter)

Default entry point for running or listing registered unit tests.

Parameters

- state – The current state of the machine as it relates to the test executable.
- shuffle – Shuffle tests
- suite_iter – Test suite repetitions.
- case_iter – Test case repetitions.

int ztest_run_test_suites(const void *state, bool shuffle, int suite_iter, int case_iter)

Run the registered unit tests which return true from their predicate function.

Parameters

- state – The current state of the machine as it relates to the test executable.
- shuffle – Shuffle tests
- suite_iter – Test suite repetitions.
- case_iter – Test case repetitions.

Returns

The number of tests that ran.

void **ztest_verify_all_test_suites_ran**(void)

Fails the test if any of the registered tests did not run.

When registering test suites, a pragma function can be provided to determine WHEN the test should run. It is possible that a test suite could be registered but the pragma always prevents it from running. In cases where a test should make sure that ALL suites ran at least once, this function may be called at the end of test_main(). It will cause the test to fail if any suite was registered but never ran.

void **ztest_test_fail**(void)

Fail the currently running test.

This is the function called from failed assertions and the like. You probably don't need to call it yourself.

void **ztest_test_pass**(void)

Pass the currently running test.

Normally a test passes just by returning without an assertion failure. However, if the success case for your test involves a fatal fault, you can call this function from k_sys_fatal_error_handler to indicate that the test passed before aborting the thread.

void **ztest_test_skip**(void)

Skip the current test.

void **ztest_skip_failed_assumption**(void)

void **ztest_simple_1cpu_before**(void *data)

A 'before' function to use in test suites that just need to start 1cpu.

Ignores data, and calls z_test_1cpu_start()

Parameters

- **data** – The test suite's data

void **ztest_simple_1cpu_after**(void *data)

A 'after' function to use in test suites that just need to stop 1cpu.

Ignores data, and calls z_test_1cpu_stop()

Parameters

- **data** – The test suite's data

Variables

struct *k_mem_partition* **ztest_mem_partition**

struct **ztest_expected_result_entry**

#include <ztest_test.h> A single expectation entry allowing tests to fail/skip and be considered passing.

See also

ZTEST_EXPECT_FAIL

See also

ZTEST_EXPECT_SKIP

Public Members

const char *`test_suite_name`
The test suite's name for the expectation.

const char *`test_name`
The test's name for the expectation.

enum *ztest_expected_result* `expected_result`
The expectation.

struct `ztest_unit_test`
#include <ztest_test.h>

Public Members

struct *ztest_unit_test_stats* *const `stats`
Stats.

struct `ztest_suite_stats`
#include <ztest_test.h> Stats about a ztest suite.

Public Members

uint32_t `run_count`
The number of times that the suite ran.

uint32_t `skip_count`
The number of times that the suite was skipped.

uint32_t `fail_count`
The number of times that the suite failed.

struct `ztest_unit_test_stats`
#include <ztest_test.h>

Public Members

uint32_t `run_count`
The number of times that the test ran.

uint32_t **skip_count**

The number of times that the test was skipped.

uint32_t **fail_count**

The number of times that the test failed.

uint32_t **pass_count**

The number of times that the test passed.

uint32_t **duration_worst_ms**

The longest duration of the test across multiple times.

struct **ztest_suite_node**

#include <ztest_test.h> A single node of test suite.

Each node should be added to a single linker section which will allow *ztest_run_test_suites()* to iterate over the various nodes.

Public Members

const char *const **name**

The name of the test suite.

const *ztest_suite_setup_t* **setup**

Setup function.

const *ztest_suite_before_t* **before**

Before function.

const *ztest_suite_after_t* **after**

After function.

const *ztest_suite_teardown_t* **teardown**

Teardown function.

const *ztest_suite_predicate_t* **predicate**

Optional predicate filter.

struct *ztest_suite_stats* *const **stats**

Stats.

struct **ztest_test_rule**

struct **ztest_arch_api**

#include <ztest_test.h> Structure for architecture specific APIs.

Assertions These macros will instantly fail the test if the related assertion fails. When an assertion fails, it will print the current file, line and function, alongside a reason for the failure and an optional message. If the config `CONFIG_ZTEST_ASSERT_VERBOSE` is 0, the assertions will only print the file and line numbers, reducing the binary size of the test.

Example output for a failed macro from `zassert_equal(buf->ref, 2, "Invalid refcount")`:

```
Assertion failed at main.c:62: test_get_single_buffer: Invalid refcount (buf->ref not equal␣
↪to 2)
Aborted at unit test function
```

group **ztest_assert**

This module provides assertions when using Ztest.

Defines

zassert(cond, default_msg, ...)

zassume(cond, default_msg, ...)

zexpect(cond, default_msg, ...)

zassert_unreachable(...)

Assert that this function call won't be reached.

Parameters

- **...** – Optional message and variables to print if the assertion fails

zassert_true(cond, ...)

Assert that *cond* is true.

Parameters

- **cond** – Condition to check
- **...** – Optional message and variables to print if the assertion fails

zassert_false(cond, ...)

Assert that *cond* is false.

Parameters

- **cond** – Condition to check
- **...** – Optional message and variables to print if the assertion fails

zassert_ok(cond, ...)

Assert that *cond* is 0 (success)

Parameters

- **cond** – Condition to check
- **...** – Optional message and variables to print if the assertion fails

zassert_not_ok(cond, ...)

Assert that *cond* is not 0 (failure)

Parameters

- **cond** – Condition to check
- **...** – Optional message and variables to print if the assertion fails

`zassert_is_null`(ptr, ...)
Assert that *ptr* is NULL.

Parameters

- `ptr` – Pointer to compare
- `...` – Optional message and variables to print if the assertion fails

`zassert_not_null`(ptr, ...)
Assert that *ptr* is not NULL.

Parameters

- `ptr` – Pointer to compare
- `...` – Optional message and variables to print if the assertion fails

`zassert_equal`(a, b, ...)
Assert that *a* equals *b*.

a and *b* won't be converted and will be compared directly.

Parameters

- `a` – Value to compare
- `b` – Value to compare
- `...` – Optional message and variables to print if the assertion fails

`zassert_not_equal`(a, b, ...)
Assert that *a* does not equal *b*.

a and *b* won't be converted and will be compared directly.

Parameters

- `a` – Value to compare
- `b` – Value to compare
- `...` – Optional message and variables to print if the assertion fails

`zassert_equal_ptr`(a, b, ...)
Assert that *a* equals *b*.

a and *b* will be converted to `void *` before comparing.

Parameters

- `a` – Value to compare
- `b` – Value to compare
- `...` – Optional message and variables to print if the assertion fails

`zassert_within`(a, b, d, ...)
Assert that *a* is within *b* with delta *d*.

Parameters

- `a` – Value to compare
- `b` – Value to compare
- `d` – Delta
- `...` – Optional message and variables to print if the assertion fails

`zassert_between_inclusive`(a, l, u, ...)

Assert that *a* is greater than or equal to *l* and less than or equal to *u*.

Parameters

- `a` – Value to compare
- `l` – Lower limit
- `u` – Upper limit
- `...` – Optional message and variables to print if the assertion fails

`zassert_mem_equal`(...)

Assert that 2 memory buffers have the same contents.

This macro calls the final memory comparison assertion macro. Using double expansion allows providing some arguments by macros that would expand to more than one values (ANSI-C99 defines that all the macro arguments have to be expanded before macro call).

Parameters

- `...` – Arguments, see *zassert_mem_equal__* for real arguments accepted.

`zassert_mem_equal__`(buf, exp, size, ...)

Internal assert that 2 memory buffers have the same contents.

> **Note**
>
> This is internal macro, to be used as a second expansion. See *zassert_mem_equal*.

Parameters

- `buf` – Buffer to compare
- `exp` – Buffer with expected contents
- `size` – Size of buffers
- `...` – Optional message and variables to print if the assertion fails

`zassert_str_equal`(s1, s2, ...)

Assert that 2 strings have the same contents.

Parameters

- `s1` – The first string
- `s2` – The second string
- `...` – Optional message and variables to print if the expectation fails

Expectations These macros will continue test execution if the related expectation fails and subsequently fail the test at the end of its execution. When an expectation fails, it will print the current file, line, and function, alongside a reason for the failure and an optional message but continue executing the test. If the config `CONFIG_ZTEST_ASSERT_VERBOSE` is 0, the expectations will only print the file and line numbers, reducing the binary size of the test.

For example, if the following expectations fail:

```
zexpect_equal(buf->ref, 2, "Invalid refcount");
zexpect_equal(buf->ref, 1337, "Invalid refcount");
```

The output will look something like:

```
START - test_get_single_buffer
    Expectation failed at main.c:62: test_get_single_buffer: Invalid refcount (buf->ref not␣
↪equal to 2)
    Expectation failed at main.c:63: test_get_single_buffer: Invalid refcount (buf->ref not␣
↪equal to 1337)
 FAIL - test_get_single_buffer in 0.0 seconds
```

group `ztest_expect`

This module provides expectations when using Ztest.

Defines

`zexpect_true`(cond, ...)

Expect that *cond* is true, otherwise mark test as failed but continue its execution.

Parameters

- `cond` – Condition to check
- `...` – Optional message and variables to print if the expectation fails

`zexpect_false`(cond, ...)

Expect that *cond* is false, otherwise mark test as failed but continue its execution.

Parameters

- `cond` – Condition to check
- `...` – Optional message and variables to print if the expectation fails

`zexpect_ok`(cond, ...)

Expect that *cond* is 0 (success), otherwise mark test as failed but continue its execution.

Parameters

- `cond` – Condition to check
- `...` – Optional message and variables to print if the expectation fails

`zexpect_not_ok`(cond, ...)

Expect that *cond* is not 0 (failure), otherwise mark test as failed but continue its execution.

Parameters

- `cond` – Condition to check
- `...` – Optional message and variables to print if the expectation fails

`zexpect_is_null`(ptr, ...)

Expect that *ptr* is NULL, otherwise mark test as failed but continue its execution.

Parameters

- `ptr` – Pointer to compare
- `...` – Optional message and variables to print if the expectation fails

`zexpect_not_null`(ptr, ...)

Expect that *ptr* is not NULL, otherwise mark test as failed but continue its execution.

Parameters

- `ptr` – Pointer to compare
- `...` – Optional message and variables to print if the expectation fails

`zexpect_equal`(a, b, ...)

Expect that *a* equals *b*, otherwise mark test as failed but continue its execution.

Parameters

- `a` – Value to compare
- `b` – Value to compare
- `...` – Optional message and variables to print if the expectation fails

`zexpect_not_equal`(a, b, ...)

Expect that *a* does not equal *b*, otherwise mark test as failed but continue its execution.

a and *b* won't be converted and will be compared directly.

Parameters

- `a` – Value to compare
- `b` – Value to compare
- `...` – Optional message and variables to print if the expectation fails

`zexpect_equal_ptr`(a, b, ...)

Expect that *a* equals *b*, otherwise mark test as failed but continue its execution.

a and *b* will be converted to `void *` before comparing.

Parameters

- `a` – Value to compare
- `b` – Value to compare
- `...` – Optional message and variables to print if the expectation fails

`zexpect_within`(a, b, delta, ...)

Expect that *a* is within *b* with delta *d*, otherwise mark test as failed but continue its execution.

Parameters

- `a` – Value to compare
- `b` – Value to compare
- `delta` – Difference between a and b
- `...` – Optional message and variables to print if the expectation fails

`zexpect_between_inclusive`(a, lower, upper, ...)

Expect that *a* is greater than or equal to *l* and less than or equal to *u*, otherwise mark test as failed but continue its execution.

Parameters

- `a` – Value to compare
- `lower` – Lower limit
- `upper` – Upper limit
- `...` – Optional message and variables to print if the expectation fails

`zexpect_mem_equal`(buf, exp, size, ...)

Expect that 2 memory buffers have the same contents, otherwise mark test as failed but continue its execution.

Parameters

- `buf` – Buffer to compare

- **exp** – Buffer with expected contents
- **size** – Size of buffers
- **...** – Optional message and variables to print if the expectation fails

zexpect_str_equal(s1, s2, ...)

Expect that 2 strings have the same contents, otherwise mark test as failed but continue its execution.

Parameters

- **s1** – The first string
- **s2** – The second string
- **...** – Optional message and variables to print if the expectation fails

Assumptions These macros will instantly skip the test or suite if the related assumption fails. When an assumption fails, it will print the current file, line, and function, alongside a reason for the failure and an optional message. If the config `CONFIG_ZTEST_ASSERT_VERBOSE` is 0, the assumptions will only print the file and line numbers, reducing the binary size of the test.

Example output for a failed macro from `zassume_equal(buf->ref, 2, "Invalid refcount")`:

group **ztest_assume**

This module provides assumptions when using Ztest.

Defines

zassume_true(cond, ...)

Assume that *cond* is true.

If the assumption fails, the test will be marked as "skipped".

Parameters

- **cond** – Condition to check
- **...** – Optional message and variables to print if the assumption fails

zassume_false(cond, ...)

Assume that *cond* is false.

If the assumption fails, the test will be marked as "skipped".

Parameters

- **cond** – Condition to check
- **...** – Optional message and variables to print if the assumption fails

zassume_ok(cond, ...)

Assume that *cond* is 0 (success)

If the assumption fails, the test will be marked as "skipped".

Parameters

- **cond** – Condition to check
- **...** – Optional message and variables to print if the assumption fails

`zassume_not_ok`(cond, ...)

Assume that *cond* is not 0 (failure)

If the assumption fails, the test will be marked as "skipped".

Parameters

- `cond` – Condition to check
- `...` – Optional message and variables to print if the assumption fails

`zassume_is_null`(ptr, ...)

Assume that *ptr* is NULL.

If the assumption fails, the test will be marked as "skipped".

Parameters

- `ptr` – Pointer to compare
- `...` – Optional message and variables to print if the assumption fails

`zassume_not_null`(ptr, ...)

Assume that *ptr* is not NULL.

If the assumption fails, the test will be marked as "skipped".

Parameters

- `ptr` – Pointer to compare
- `...` – Optional message and variables to print if the assumption fails

`zassume_equal`(a, b, ...)

Assume that *a* equals *b*.

a and *b* won't be converted and will be compared directly. If the assumption fails, the test will be marked as "skipped".

Parameters

- `a` – Value to compare
- `b` – Value to compare
- `...` – Optional message and variables to print if the assumption fails

`zassume_not_equal`(a, b, ...)

Assume that *a* does not equal *b*.

a and *b* won't be converted and will be compared directly. If the assumption fails, the test will be marked as "skipped".

Parameters

- `a` – Value to compare
- `b` – Value to compare
- `...` – Optional message and variables to print if the assumption fails

`zassume_equal_ptr`(a, b, ...)

Assume that *a* equals *b*.

a and *b* will be converted to `void *` before comparing. If the assumption fails, the test will be marked as "skipped".

Parameters

- `a` – Value to compare
- `b` – Value to compare

- `...` – Optional message and variables to print if the assumption fails

`zassume_within`(a, b, d, ...)

Assume that *a* is within *b* with delta *d*.

If the assumption fails, the test will be marked as "skipped".

Parameters

- `a` – Value to compare
- `b` – Value to compare
- `d` – Delta
- `...` – Optional message and variables to print if the assumption fails

`zassume_between_inclusive`(a, l, u, ...)

Assume that *a* is greater than or equal to *l* and less than or equal to *u*.

If the assumption fails, the test will be marked as "skipped".

Parameters

- `a` – Value to compare
- `l` – Lower limit
- `u` – Upper limit
- `...` – Optional message and variables to print if the assumption fails

`zassume_mem_equal`(...)

Assume that 2 memory buffers have the same contents.

This macro calls the final memory comparison assumption macro. Using double expansion allows providing some arguments by macros that would expand to more than one values (ANSI-C99 defines that all the macro arguments have to be expanded before macro call).

Parameters

- `...` – Arguments, see *zassume_mem_equal__* for real arguments accepted.

`zassume_mem_equal__`(buf, exp, size, ...)

Internal assume that 2 memory buffers have the same contents.

If the assumption fails, the test will be marked as "skipped".

> **Note**
>
> This is internal macro, to be used as a second expansion. See *zassume_mem_equal*.

Parameters

- `buf` – Buffer to compare
- `exp` – Buffer with expected contents
- `size` – Size of buffers
- `...` – Optional message and variables to print if the assumption fails

`zassume_str_equal`(s1, s2, ...)

Assumes that 2 strings have the same contents.

Parameters

- `s1` – The first string

- **s2** – The second string
- **...** – Optional message and variables to print if the expectation fails

Ztress

group **ztest_ztress**

This module provides test stress when using Ztest.

Defines

ZTRESS_TIMER(handler, user_data, exec_cnt, init_timeout)

Descriptor of a k_timer handler execution context.

The handler is executed in the k_timer handler context which typically means interrupt context. This context will preempt any other used in the set.

> **Note**
>
> There can only be up to one k_timer context in the set and it must be the first argument of *ZTRESS_EXECUTE*.

Parameters

- **handler** – User handler of type *ztress_handler*.
- **user_data** – User data passed to the handler.
- **exec_cnt** – Number of handler executions to complete the test. If 0 then this is not included in completion criteria.
- **init_timeout** – Initial backoff time base (given in *k_timeout_t*). It is adjusted during the test to optimize CPU load. The actual timeout used for the timer is randomized.

ZTRESS_THREAD(handler, user_data, exec_cnt, preempt_cnt, init_timeout)

Descriptor of a thread execution context.

The handler is executed in the thread context. The priority of the thread is determined based on the order in which contexts are listed in *ZTRESS_EXECUTE*.

> **Note**
>
> thread sleeps for random amount of time. Additionally, the thread busy-waits for a random length of time to further increase randomization in the test.

Parameters

- **handler** – User handler of type *ztress_handler*.
- **user_data** – User data passed to the handler.
- **exec_cnt** – Number of handler executions to complete the test. If 0 then this is not included in completion criteria.
- **preempt_cnt** – Number of preemptions of that context to complete the test. If 0 then this is not included in completion criteria.

- **init_timeout** – Initial backoff time base (given in *k_timeout_t*). It is adjusted during the test to optimize CPU load. The actual timeout used for sleeping is randomized.

ZTRESS_CONTEXT_INITIALIZER(_handler, _user_data, _exec_cnt, _preempt_cnt, _t)

Initialize context structure.

For argument types see *ztress_context_data*. For more details see *ZTRESS_THREAD*.

Parameters

- **_handler** – Handler.
- **_user_data** – User data passed to the handler.
- **_exec_cnt** – Execution count limit.
- **_preempt_cnt** – Preemption count limit.
- **_t** – Initial timeout.

ZTRESS_EXECUTE(...)

Setup and run stress test.

It initialises all contexts and calls *ztress_execute*.

Parameters

- ... – List of contexts. Contexts are configured using *ZTRESS_TIMER* and *ZTRESS_THREAD* macros. *ZTRESS_TIMER* must be the first argument if used. Each thread context has an assigned priority. The priority is assigned in a descending order (first listed thread context has the highest priority). The maximum number of supported thread contexts, including the timer context, is configurable in Kconfig (ZTRESS_MAX_THREADS).

Typedefs

typedef bool (*ztress_handler)(void *user_data, uint32_t cnt, bool last, int prio)

User handler called in one of the configured contexts.

Param user_data
User data provided in the context descriptor.

Param cnt
Current execution counter. Counted from 0.

Param last
Flag set to true indicates that it is the last execution because completion criteria are met, test timed out or was aborted.

Param prio
Context priority counting from 0 which indicates the highest priority.

Retval true
continue test.

Retval false
stop executing the current context.

Functions

int **ztress_execute**(struct *ztress_context_data* *timer_data, struct *ztress_context_data* *thread_data, size_t cnt)

Execute contexts.

The test runs until all completion requirements are met or until the test times out (use *ztress_set_timeout* to configure timeout) or until the test is aborted (*ztress_abort*).

on test completion a report is printed (*ztress_report* is called internally).

Parameters

- **timer_data** – Timer context. NULL if timer context is not used.
- **thread_data** – List of thread contexts descriptors in priority descending order.
- **cnt** – Number of thread contexts.

Return values

- **-EINVAL** – If configuration is invalid.
- **0** – if test is successfully performed.

void **ztress_abort**(void)

Abort ongoing stress test.

void **ztress_set_timeout**(*k_timeout_t* t)

Set test timeout.

Test is terminated after timeout disregarding completion criteria. Setting is persistent between executions.

Parameters

- **t** – Timeout.

void **ztress_report**(void)

Print last test report.

Report contains number of executions and preemptions for each context, initial and adjusted timeouts and CPU load during the test.

int **ztress_exec_count**(uint32_t id)

Get number of executions of a given context in the last test.

Parameters

- **id** – Context id. 0 means the highest priority.

Returns

Number of executions.

int **ztress_preempt_count**(uint32_t id)

Get number of preemptions of a given context in the last test.

Parameters

- **id** – Context id. 0 means the highest priority.

Returns

Number of preemptions.

uint32_t **ztress_optimized_ticks**(uint32_t id)

Get optimized timeout base of a given context in the last test.

Optimized value can be used to update initial value. It will improve the test since optimal CPU load will be reach immediately.

Parameters

- **id** – Context id. 0 means the highest priority.

Returns
Optimized timeout base.

struct **ztress_context_data**
#include <ztress.h>

Mocking via FFF Zephyr has integrated with FFF for mocking. See FFF for documentation. To use it, include the relevant header:

```
#include <zephyr/fff.h>
```

Zephyr provides several FFF-based fake drivers which can be used as either stubs or mocks. Fake driver instances are configured via *Devicetree* and *Configuration System (Kconfig)*. See the following devicetree bindings for more information:

- `zephyr,fake-can`
- `zephyr,fake-eeprom`

Zephyr also has defined extensions to FFF for simplified declarations of fake functions. See *FFF Extensions*.

Customizing Test Output

Customization is enabled by setting `CONFIG_ZTEST_TC_UTIL_USER_OVERRIDE` to "y" and adding a file `tc_util_user_override.h` with your overrides.

Add the line `zephyr_include_directories(my_folder)` to your project's `CMakeLists.txt` to let Zephyr find your header file during builds.

See the file subsys/testsuite/include/zephyr/tc_util.h to see which macros and/or defines can be overridden. These will be surrounded by blocks such as:

```
#ifndef SOMETHING
#define SOMETHING <default implementation>
#endif /* SOMETHING */
```

Shuffling Test Sequence

By default the tests are sorted and ran in alphanumerical order. Test cases may be dependent on this sequence. Enable `CONFIG_ZTEST_SHUFFLE` to randomize the order. The output from the test will display the seed for failed tests. For native simulator builds you can provide the seed as an argument to twister with *–seed*

Static configuration of ZTEST_SHUFFLE contains:

- `CONFIG_ZTEST_SHUFFLE_SUITE_REPEAT_COUNT` - Number of iterations the test suite will run.
- `CONFIG_ZTEST_SHUFFLE_TEST_REPEAT_COUNT` - Number of iterations the test will run.

Test Selection

For tests built for native simulator, use command line arguments to list or select tests to run. The test argument expects a comma separated list of `suite::test` . You can substitute the test name with an `*` to run all tests within a suite.

For example

```
$ zephyr.exe -list
$ zephyr.exe -test="fixture_tests::test_fixture_pointer,framework_tests::test_assert_mem_
↪equal"
$ zephyr.exe -test="framework_tests::*"
```

FFF Extensions

group `fff_extensions`

This module provides extensions to FFF for simplifying the configuration and usage of fakes.

Defines

`RETURN_HANDLED_CONTEXT`(FUNCNAME, CONTEXTTYPE, RESULTFIELD, CONTEXTPTRNAME, HANDLERBODY)

Wrap custom fake body to extract defined context struct.

Add extension macro for simplified creation of fake functions needing call-specific context data.

This macro enables a fake to be implemented as follows and requires no familiarity with the inner workings of FFF.

```
struct FUNCNAME##_custom_fake_context
{
    struct instance * const instance;
    int result;
};

int FUNCNAME##_custom_fake(
    const struct instance **instance_out)
{
    RETURN_HANDLED_CONTEXT(
        FUNCNAME,
        struct FUNCNAME##_custom_fake_context,
        result,
        context,
        {
            if (context != NULL)
            {
                if (context->result == 0)
                {
                    if (instance_out != NULL)
                    {
                        *instance_out = context->instance;
                    }
                }
                return context->result;
            }
            return FUNCNAME##_fake.return_val;
        }
    );
}
```

Parameters

- `FUNCNAME` – Name of function being faked

- `CONTEXTTYPE` – type of custom defined fake context struct
- `RESULTFIELD` – name of field holding the return type & value
- `CONTEXTPTRNAME` – expected name of pointer to custom defined fake context struct
- `HANDLERBODY` – in-line custom fake handling logic

2.12.2 Test Runner (Twister)

Twister scans for the set of test applications in the git repository and attempts to execute them. By default, it tries to build each test application on boards marked as default in the board definition file.

The default options will build the majority of the test applications on a defined set of boards and will run in an emulated environment if available for the architecture or configuration being tested.

Because of the limited test execution coverage, twister cannot guarantee local changes will succeed in the full build environment, but it does sufficient testing by building samples and tests for different boards and different configurations to help keep the complete code tree buildable.

When using (at least) one `-v` option, twister's console output shows for every test application how the test is run (qemu, native_sim, etc.) or whether the binary was just built. There are a few reasons why twister only builds a test and doesn't run it:

- The test is marked as `build_only: true` in its `.yaml` configuration file.
- The test configuration has defined a `harness` but you don't have it or haven't set it up.
- The target device is not connected and not available for flashing
- You or some higher level automation invoked twister with `--build-only`.

To run the script in the local tree, follow the steps below:

Linux

```
$ source zephyr-env.sh
$ ./scripts/twister
```

Windows

```
zephyr-env.cmd
python .\scripts\twister
```

If you have a system with a large number of cores and plenty of free storage space, you can build and run all possible tests using the following options:

Linux

```
$ ./scripts/twister --all --enable-slow
```

Windows

```
python .\scripts\twister --all --enable-slow
```

This will build for all available boards and run all applicable tests in a simulated (for example QEMU) environment.

If you want to run tests on one or more specific platforms, you can use the `--platform` option, it is a platform filter for testing, with this option, test suites will only be built/run on the platforms specified. This option also supports different revisions of one same board, you can use `--platform board@revision` to test on a specific revision.

The list of command line options supported by twister can be viewed using:

Linux

```
$ ./scripts/twister --help
```

Windows

```
python .\scripts\twister --help
```

Board Configuration

To build tests for a specific board and to execute some of the tests on real hardware or in an emulation environment such as QEMU a board configuration file is required which is generic enough to be used for other tasks that require a board inventory with details about the board and its configuration that is only available during build time otherwise.

The board metadata file is located in the board directory and is structured using the YAML markup language. The example below shows a board with a data required for best test coverage for this specific board:

```
identifier: frdm_k64f
name: NXP FRDM-K64F
type: mcu
arch: arm
toolchain:
  - zephyr
  - gnuarmemb
  - xtools
supported:
  - arduino_gpio
  - arduino_i2c
  - netif:eth
  - adc
  - i2c
  - nvs
  - spi
  - gpio
  - usb_device
  - watchdog
  - can
  - pwm
testing:
  default: true
```

identifier:
A string that matches how the board is defined in the build system. This same string is used when building, for example when calling `west build` or `cmake`:

```
# with west
west build -b reel_board
# with cmake
cmake -DBOARD=reel_board ..
```

name:
The actual name of the board as it appears in marketing material.

type:
Type of the board or configuration, currently we support 2 types: mcu, qemu

simulation:
Simulator used to simulate the platform, e.g. qemu.

arch:
Architecture of the board

toolchain:
The list of supported toolchains that can build this board. This should match one of the values used for *ZEPHYR_TOOLCHAIN_VARIANT* when building on the command line

ram:
Available RAM on the board (specified in KB). This is used to match test scenario requirements. If not specified we default to 128KB.

flash:
Available FLASH on the board (specified in KB). This is used to match test scenario requirements. If not specified we default to 512KB.

supported:
A list of features this board supports. This can be specified as a single word feature or as a variant of a feature class. For example:

```
supported:
  - pci
```

This indicates the board does support PCI. You can make a test scenario build or run only on such boards, or:

```
supported:
  - netif:eth
  - sensor:bmi16
```

A test scenario can depend on 'eth' to only test ethernet or on 'netif' to run on any board with a networking interface.

testing:
testing relating keywords to provide best coverage for the features of this board.

default: [True|False]:
This is a default board, it will tested with the highest priority and is covered when invoking the simplified twister without any additional arguments.

ignore_tags:
Do not attempt to build (and therefore run) tests marked with this list of tags.

only_tags:
Only execute tests with this list of tags on a specific platform.

timeout_multiplier: <float> (default 1)
Multiply each test scenario timeout by specified ratio. This option allows to tune timeouts only for required platform. It can be useful in case naturally slow platform I.e.: HW board with power-efficient but slow CPU or simulation platform which can perform instruction accurate simulation but does it slowly.

env:
A list of environment variables. Twister will check if all these environment variables are set, and otherwise skip this platform. This allows the user to define a platform which should be used, for example, only if some required software or hardware is present, and to signal that presence to twister using these environment variables.

Tests

Tests are detected by the presence of a `testcase.yaml` or a `sample.yaml` files in the application's project directory. This test application configuration file may contain one or more entries in the tests section each identifying a test scenario.

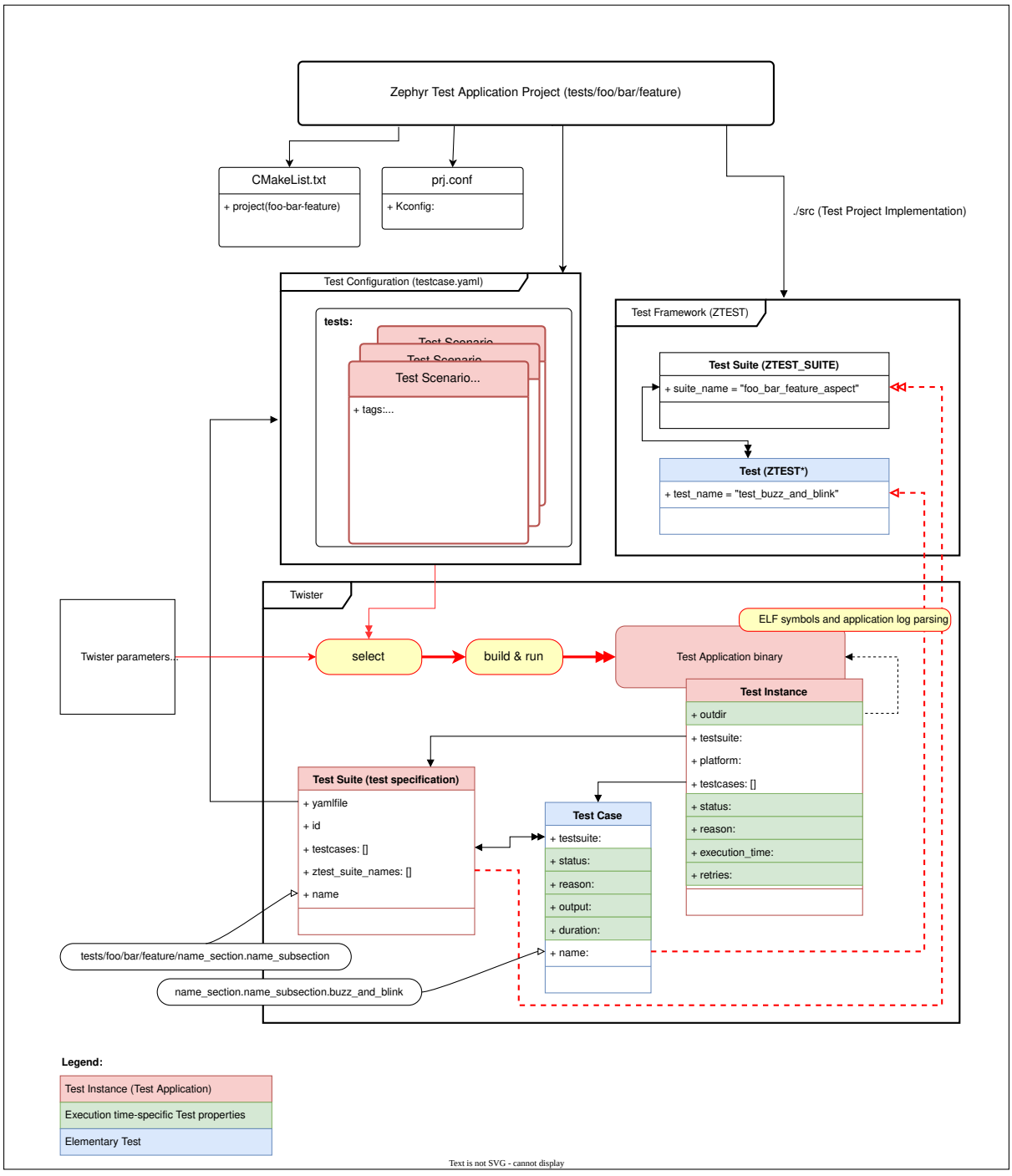


Fig. 4: Twister and a Test applications' project.

Test application configurations are written using the YAML syntax and share the same structure as samples.

A test scenario is a set of conditions or variables, defined in test scenario entry, under which a set of test suites will be executed. Can be used interchangeably with test scenario entry.

A test suite is a collection of test cases that are intended to be used to test a software program to ensure it meets certain requirements. The test cases in a test suite are often related or meant to be executed together.

The name of each test scenario needs to be unique in the context of the overall test application and has to follow basic rules:

1. The format of the test scenario identifier shall be a string without any spaces or special characters (allowed characters: alphanumeric and [_=]) consisting of multiple sections delimited with a dot (.).
2. Each test scenario identifier shall start with a section followed by a subsection separated by a dot. For example, a test scenario that covers semaphores in the kernel shall start with `kernel.semaphore`.
3. All test scenario identifiers within a `testcase.yaml` file need to be unique. For example a `testcase.yaml` file covering semaphores in the kernel can have:
   - `kernel.semaphore`: For general semaphore tests
   - `kernel.semaphore.stress`: Stress testing semaphores in the kernel.
4. Depending on the nature of the test, an identifier can consist of at least two sections:
   - Ztest tests: The individual test cases in the ztest testsuite will be concatenated by dot (.) to the identifier in the `testcase.yaml` file generating unique identifiers for every test case in the suite.
   - Standalone tests and samples: This type of test should at least have 3 sections concatnated by dot (.) in the test scenario identifier in the `testcase.yaml` (or `sample.yaml`) file. The last section of the name shall signify the test case itself.

The following is an example test configuration with a few options that are explained in this document.

```
tests:
  bluetooth.gatt:
    build_only: true
    platform_allow: qemu_cortex_m3 qemu_x86
    tags: bluetooth
  bluetooth.gatt.br:
    build_only: true
    extra_args: CONF_FILE="prj_br.conf"
    filter: not CONFIG_DEBUG
    platform_exclude: up_squared
    platform_allow: qemu_cortex_m3 qemu_x86
    tags: bluetooth
```

A sample with tests will have the same structure with additional information related to the sample and what is being demonstrated:

```
sample:
  name: hello world
  description: Hello World sample, the simplest Zephyr application
tests:
  sample.basic.hello_world:
    build_only: true
    tags: tests
    min_ram: 16
```

(continues on next page)

(continued from previous page)

```
  sample.basic.hello_world.singlethread:
    build_only: true
    extra_args: CONF_FILE=prj_single.conf
    filter: not CONFIG_BT
    tags: tests
    min_ram: 16
```

The full canonical name for each test scenario is:`<path to test application>/<test scenario identifier>`

A test scenario entry is a a block or entry starting with test scenario identifier in the YAML files.

Each test scenario entry in the test application configuration can define the following key/value pairs:

tags: <list of tags> (required)
A set of string tags for the test scenario. Usually pertains to functional domains but can be anything. Command line invocations of this script can filter the set of tests to run based on tag.

skip: <True|False> (default False)
skip test scenario unconditionally. This can be used for broken tests for example.

slow: <True|False> (default False)
Don't run this test scenario unless `--enable-slow` or `--enable-slow-only` was passed in on the command line. Intended for time-consuming test scenarios that are only run under certain circumstances, like daily builds. These test scenarios are still compiled.

extra_args: <list of extra arguments>
Extra arguments to pass to build tool when building or running the test scenario.

extra_configs: <list of extra configurations>
Extra configuration options to be merged with a main prj.conf when building or running the test scenario. For example:

```
common:
  tags: drivers adc
tests:
  test:
    depends_on: adc
  test_async:
    extra_configs:
      - CONFIG_ADC_ASYNC=y
```

Using namespacing, it is possible to apply a configuration only to some hardware. Currently both architectures and platforms are supported:

```
common:
  tags: drivers adc
tests:
  test:
    depends_on: adc
  test_async:
    extra_configs:
      - arch:x86:CONFIG_ADC_ASYNC=y
      - platform:qemu_x86:CONFIG_DEBUG=y
```

build_only: <True|False> (default False)
If true, twister will not try to run the test even if the test is runnable on the platform.

This keyword is reserved for tests that are used to test if some code actually builds. A `build_only` test is not designed to be run in any environment and should not be testing any functionality, it only verifies that the code builds.

This option is often used to test drivers and the fact that they are correctly enabled in Zephyr and that the code builds, for example sensor drivers. Such test shall not be used to verify the functionality of the driver.

build_on_all: <True|False> (default False)
If true, attempt to build test scenario on all available platforms. This is mostly used in CI for increased coverage. Do not use this flag in new tests.

depends_on: <list of features>
A board or platform can announce what features it supports, this option will enable the test only those platforms that provide this feature.

levels: <list of levels>
Test levels this test should be part of. If a level is present, this test will be selectable using the command line option `--level <level name>`

min_ram: <integer>
minimum amount of RAM in KB needed for this test to build and run. This is compared with information provided by the board metadata.

min_flash: <integer>
minimum amount of ROM in KB needed for this test to build and run. This is compared with information provided by the board metadata.

timeout: <number of seconds>
Length of time to run test before automatically killing it. Default to 60 seconds.

arch_allow: <list of arches, such as x86, arm, arc>
Set of architectures that this test scenario should only be run for.

arch_exclude: <list of arches, such as x86, arm, arc>
Set of architectures that this test scenario should not run on.

platform_allow: <list of platforms>
Set of platforms that this test scenario should only be run for. Do not use this option to limit testing or building in CI due to time or resource constraints, this option should only be used if the test or sample can only be run on the allowed platform and nothing else.

integration_platforms: <YML list of platforms/boards>
This option limits the scope to the listed platforms when twister is invoked with the `--integration` option. Use this instead of platform_allow if the goal is to limit scope due to timing or resource constraints.

platform_exclude: <list of platforms>
Set of platforms that this test scenario should not run on.

extra_sections: <list of extra binary sections>
When computing sizes, twister will report errors if it finds extra, unexpected sections in the Zephyr binary unless they are named here. They will not be included in the size calculation.

sysbuild: <True|False> (default False)
Build the project using sysbuild infrastructure. Only the main project's generated device-tree and Kconfig will be used for filtering tests. on device testing must use the hardware map, or west flash to load the images onto the target. The `--erase` option of west flash is not supported with this option. Usage of unsupported options will result in tests requiring sysbuild support being skipped.

harness: <string>
A harness keyword in the `testcase.yaml` file identifies a Twister harness needed to run a test successfully. A harness is a feature of Twister and implemented by Twister, some harnesses are defined as placeholders and have no implementation yet.

A harness can be seen as the handler that needs to be implemented in Twister to be able to evaluate if a test passes criteria. For example, a keyboard harness is set on tests that require keyboard interaction to reach verdict on whether a test has passed or failed, however, Twister lack this harness implementation at the moment.

Supported harnesses:

- ztest
- test
- console
- pytest
- gtest
- robot

Harnesses `ztest`, `gtest` and `console` are based on parsing of the output and matching certain phrases. `ztest` and `gtest` harnesses look for pass/fail/etc. frames defined in those frameworks. Use `gtest` harness if you've already got tests written in the gTest framework and do not wish to update them to zTest. The `console` harness tells Twister to parse a test's text output for a regex defined in the test's YAML file. The `robot` harness is used to execute Robot Framework test suites in the Renode simulation framework.

Some widely used harnesses that are not supported yet:

- keyboard
- net
- bluetooth

Harness `bsim` is implemented in limited way - it helps only to copy the final executable (`zephyr.exe`) from build directory to BabbleSim's `bin` directory (`${BSIM_OUT_PATH}/bin`). This action is useful to allow BabbleSim's tests to directly run after. By default, the executable file name is (with dots and slashes replaced by underscores): `bs_<platform_name>_<test_path>_<test_scenario_name>`. This name can be overridden with the `bsim_exe_name` option in `harness_config` section.

platform_key: <list of platform attributes>
Often a test needs to only be built and run once to qualify as passing. Imagine a library of code that depends on the platform architecture where passing the test on a single platform for each arch is enough to qualify the tests and code as passing. The platform_key attribute enables doing just that.

For example to key on (arch, simulation) to ensure a test is run once per arch and simulation (as would be most common):

```
platform_key:
  - arch
  - simulation
```

Adding platform (board) attributes to include things such as soc name, soc family, and perhaps sets of IP blocks implementing each peripheral interface would enable other interesting uses. For example, this could enable building and running SPI tests once for each unique IP block.

harness_config: <harness configuration options>
Extra harness configuration options to be used to select a board and/or for handling generic Console with regex matching. Config can announce what features it supports. This option will enable the test to run on only those platforms that fulfill this external dependency.

The following options are currently supported:

type: <one_line|multi_line> (required)
Depends on the regex string to be matched

regex: <list of regular expressions> (required)
Strings with regular expressions to match with the test's output to confirm the test runs as expected.

ordered: <True|False> (default False)
Check the regular expression strings in orderly or randomly fashion

record: <recording options> (optional)

regex: <regular expression> (required)
The regular expression with named subgroups to match data fields at the test's output lines where the test provides some custom data for further analysis. These records will be written into the build directory `recording.csv` file as well as `recording` property of the test suite object in `twister.json`.

For example, to extract three data fields `metric`, `cycles`, `nanoseconds`:

```
record:
  regex: "(?P<metric>.*):(?P<cycles>.*) cycles, (?P<nanoseconds>.*) ns"
```

as_json: <list of regex subgroup names> (optional)
Data fields, extracted by the regular expression into named subgroups, which will be additionally parsed as JSON encoded strings and written into `twister.json` as nested `recording` object properties. The corresponding `recording.csv` columns will contain strings as-is.

Using this option, a test log can convey layered data structures passed from the test image for further analysis with summary results, traces, statistics, etc.

For example, this configuration:

```
record:
  regex: "RECORD:(?P<type>.*):DATA:(?P<metrics>.*)"
  as_json: [metrics]
```

when matched to a test log string:

```
RECORD:jitter_drift:DATA:{"rollovers":0, "mean_us":1000.0}
```

will be reported in `twister.json` as:

```
"recording":[
    {
        "type":"jitter_drift",
        "metrics":{
            "rollovers":0,
            "mean_us":1000.0
        }
    }
]
```

fixture: <expression>
Specify a test scenario dependency on an external device(e.g., sensor), and identify setups that fulfill this dependency. It depends on specific test setup and board selection logic to pick the particular board(s) out of multiple boards that fulfill the dependency in an automation setup based on `fixture` keyword. Some sample fixture names are i2c_hts221, i2c_bme280, i2c_FRAM, ble_fw and gpio_loop.

Only one fixture can be defined per test scenario and the fixture name has to be unique across all tests in the test suite.

pytest_root: <list of pytest testpaths> (default pytest)
Specify a list of pytest directories, files or subtests that need to be executed when a test scenario begins to run. The default pytest directory is `pytest`. After the pytest run is finished, Twister will check if the test scenario passed or failed according to the pytest report. As an example, a list of valid pytest roots is presented below:

```
harness_config:
  pytest_root:
    - "pytest/test_shell_help.py"
    - "../shell/pytest/test_shell.py"
    - "/tmp/test_shell.py"
    - "~/tmp/test_shell.py"
    - "$ZEPHYR_BASE/samples/subsys/testsuite/pytest/shell/pytest/test_shell.
↪py"
    - "pytest/test_shell_help.py::test_shell2_sample"  # select pytest␣
↪subtest
    - "pytest/test_shell_help.py::test_shell2_sample[param_a]"  # select␣
↪pytest parametrized subtest
```

pytest_args: <list of arguments> (default empty)
Specify a list of additional arguments to pass to `pytest` e.g.: `pytest_args: ['-k=test_method', '--log-level=DEBUG']`. Note that `--pytest-args` can be passed multiple times to pass several arguments to the pytest.

pytest_dut_scope: <function|class|module|package|session> (default function)
The scope for which `dut` and `shell` pytest fixtures are shared. If the scope is set to `function`, DUT is launched for every test case in python script. For `session` scope, DUT is launched only once.

robot_testsuite: <robot file path> (default empty)
Specify one or more paths to a file containing a Robot Framework test suite to be run.

robot_option: <robot option> (default empty)
One or more options to be send to robotframework.

bsim_exe_name: <string>
If provided, the executable filename when copying to BabbleSim's bin directory, will be `bs_<platform_name>_<bsim_exe_name>` instead of the default based on the test path and scenario name.

The following is an example yaml file with a few harness_config options.

```
sample:
  name: HTS221 Temperature and Humidity Monitor
common:
  tags: sensor
  harness: console
  harness_config:
    type: multi_line
    ordered: false
    regex:
      - "Temperature:(.*)C"
      - "Relative Humidity:(.*)%"
    fixture: i2c_hts221
tests:
  test:
    tags: sensors
    depends_on: i2c
```

The following is an example yaml file with pytest harness_config options, default pytest_root name "pytest" will be used if pytest_root not specified. please refer the examples in samples/subsys/testsuite/pytest/.

```
common:
  harness: pytest
tests:
  pytest.example.directories:
```

(continues on next page)

(continued from previous page)

```
    harness_config:
      pytest_root:
        - pytest_dir1
        - $ENV_VAR/samples/test/pytest_dir2
  pytest.example.files_and_subtests:
    harness_config:
      pytest_root:
        - pytest/test_file_1.py
        - test_file_2.py::test_A
        - test_file_2.py::test_B[param_a]
```

The following is an example yaml file with robot harness_config options.

```
tests:
  robot.example:
    harness: robot
    harness_config:
      robot_testsuite: [robot file path]
```

It can be more than one test suite using a list.

```
tests:
  robot.example:
    harness: robot
    harness_config:
      robot_testsuite:
        - [robot file path 1]
        - [robot file path 2]
        - [robot file path n]
```

One or more options can be passed to robotframework.

```
tests:
  robot.example:
    harness: robot
    harness_config:
      robot_testsuite: [robot file path]
      robot_option:
        - --exclude tag
        - --stop-on-error
```

filter: <expression>
Filter whether the test scenario should be run by evaluating an expression against an environment containing the following values:

```
{ ARCH : <architecture>,
  PLATFORM : <platform>,
  <all CONFIG_* key/value pairs in the test's generated defconfig>,
  *<env>: any environment variable available
}
```

Twister will first evaluate the expression to find if a “limited” cmake call, i.e. using package_helper cmake script, can be done. Existence of “dt_*” entries indicates devicetree is needed. Existence of “CONFIG*” entries indicates kconfig is needed. If there are no other types of entries in the expression a filtration can be done without creating a complete build system. If there are entries of other types a full cmake is required.

The grammar for the expression language is as follows:

```
expression : expression 'and' expression
           | expression 'or' expression
```

(continues on next page)

(continued from previous page)

```
            | 'not' expression
            | '(' expression ')'
            | symbol '==' constant
            | symbol '!=' constant
            | symbol '<' NUMBER
            | symbol '>' NUMBER
            | symbol '>=' NUMBER
            | symbol '<=' NUMBER
            | symbol 'in' list
            | symbol ':' STRING
            | symbol
            ;

list : '[' list_contents ']';

list_contents : constant (',' constant)*;

constant : NUMBER | STRING;
```

For the case where `expression ::= symbol`, it evaluates to `true` if the symbol is defined to a non-empty string.

Operator precedence, starting from lowest to highest:

- or (left associative)
- and (left associative)
- not (right associative)
- all comparison operators (non-associative)

`arch_allow`, `arch_exclude`, `platform_allow`, `platform_exclude` are all syntactic sugar for these expressions. For instance:

```
arch_exclude = x86 arc
```

Is the same as:

```
filter = not ARCH in ["x86", "arc"]
```

The `:` operator compiles the string argument as a regular expression, and then returns a true value only if the symbol's value in the environment matches. For example, if `CONFIG_SOC="stm32f107xc"` then

```
filter = CONFIG_SOC : "stm.*"
```

Would match it.

required_snippets: <list of needed snippets>

Snippets are supported in twister for test scenarios that require them. As with normal applications, twister supports using the base zephyr snippet directory and test application directory for finding snippets. Listed snippets will filter supported tests for boards (snippets must be compatible with a board for the test to run on them, they are not optional).

The following is an example yaml file with 2 required snippets.

```
tests:
  snippet.example:
    required_snippets:
      - cdc-acm-console
      - user-snippet-example
```

The set of test scenarios that actually run depends on directives in the test scenario filed and options passed in on the command line. If there is any confusion, running with `-v` or examining the discard report (`twister_discard.csv`) can help show why particular test scenarios were skipped.

Metrics (such as pass/fail state and binary size) for the last code release are stored in `scripts/release/twister_last_release.csv`. To update this, pass the `--all --release` options.

To load arguments from a file, add + before the file name, e.g., `+file_name`. File content must be one or more valid arguments separated by line break instead of white spaces.

Most everyday users will run with no arguments.

Managing tests timeouts

There are several parameters which control tests timeouts on various levels:

- `timeout` option in each test scenario. See *here* for more details.
- `timeout_multiplier` option in board configuration. See *here* for more details.
- `--timeout-multiplier` twister option which can be used to adjust timeouts in exact twister run. It can be useful in case of simulation platform as simulation time may depend on the host speed & load or we may select different simulation method (i.e. cycle accurate but slower one), etc...

Overall test scenario timeout is a multiplication of these three parameters.

Running in Integration Mode

This mode is used in continuous integration (CI) and other automated environments used to give developers fast feedback on changes. The mode can be activated using the `--integration` option of twister and narrows down the scope of builds and tests if applicable to platforms defined under the integration keyword in the test configuration file (`testcase.yaml` and `sample.yaml`).

Running tests on custom emulator

Apart from the already supported QEMU and other simulated environments, Twister supports running any out-of-tree custom emulator defined in the board's `board.cmake`. To use this type of simulation, add the following properties to `custom_board/custom_board.yaml`:

```
simulation: custom
simulation_exec: <name_of_emu_binary>
```

This tells Twister that the board is using a custom emulator called `<name_of_emu_binary>`, make sure this binary exists in the PATH.

Then, in `custom_board/board.cmake`, set the supported emulation platforms to `custom`:

```
set(SUPPORTED_EMU_PLATFORMS custom)
```

Finally, implement the `run_custom` target in `custom_board/board.cmake`. It should look something like this:

```
add_custom_target(run_custom
  COMMAND
  <name_of_emu_binary to invoke during 'run'>
  <any args to be passed to the command, i.e. ${BOARD}, ${APPLICATION_BINARY_DIR}/zephyr/
↪zephyr.elf>
```

(continues on next page)

(continued from previous page)

```
  WORKING_DIRECTORY ${APPLICATION_BINARY_DIR}
  DEPENDS ${logical_target_for_zephyr_elf}
  USES_TERMINAL
  )
```

Running Tests on Hardware

Beside being able to run tests in QEMU and other simulated environments, twister supports running most of the tests on real devices and produces reports for each run with detailed FAIL/PASS results.

Executing tests on a single device To use this feature on a single connected device, run twister with the following new options:

Linux

```
scripts/twister --device-testing --device-serial /dev/ttyACM0 \
--device-serial-baud 115200 -p frdm_k64f  -T tests/kernel
```

Windows

```
python .\scripts\twister --device-testing --device-serial COM1 \
--device-serial-baud 115200 -p frdm_k64f  -T tests/kernel
```

The `--device-serial` option denotes the serial device the board is connected to. This needs to be accessible by the user running twister. You can run this on only one board at a time, specified using the `--platform` option.

The `--device-serial-baud` option is only needed if your device does not run at 115200 baud.

To support devices without a physical serial port, use the `--device-serial-pty` option. In this cases, log messages are captured for example using a script. In this case you can run twister with the following options:

Linux

```
scripts/twister --device-testing --device-serial-pty "script.py" \
-p intel_adsp/cavs25 -T tests/kernel
```

Windows

> **Note**
>
> Not supported on Windows OS

The script is user-defined and handles delivering the messages which can be used by twister to determine the test execution status.

The `--device-flash-timeout` option allows to set explicit timeout on the device flash operation, for example when device flashing takes significantly large time.

The `--device-flash-with-test` option indicates that on the platform the flash operation also executes a test scenario, so the flash timeout is increased by a test scenario timeout.

Executing tests on multiple devices To build and execute tests on multiple devices connected to the host PC, a hardware map needs to be created with all connected devices and their details

such as the serial device, baud and their IDs if available. Run the following command to produce the hardware map:

Linux

```
./scripts/twister --generate-hardware-map map.yml
```

Windows

```
python .\scripts\twister --generate-hardware-map map.yml
```

The generated hardware map file (map.yml) will have the list of connected devices, for example:

Linux

```
- connected: true
  id: OSHW000032254e4500128002ab98002784d1000097969900
  platform: unknown
  product: DAPLink CMSIS-DAP
  runner: pyocd
  serial: /dev/cu.usbmodem146114202
- connected: true
  id: 000683759358
  platform: unknown
  product: J-Link
  runner: unknown
  serial: /dev/cu.usbmodem0006837593581
```

Windows

```
- connected: true
  id: OSHW000032254e4500128002ab98002784d1000097969900
  platform: unknown
  product: unknown
  runner: unknown
  serial: COM1
- connected: true
  id: 000683759358
  platform: unknown
  product: unknown
  runner: unknown
  serial: COM2
```

Any options marked as unknown need to be changed and set with the correct values, in the above example the platform names, the products and the runners need to be replaced with the correct values corresponding to the connected hardware. In this example we are using a reel_board and an nrf52840dk/nrf52840:

Linux

```
- connected: true
  id: OSHW000032254e4500128002ab98002784d1000097969900
  platform: reel_board
  product: DAPLink CMSIS-DAP
  runner: pyocd
  serial: /dev/cu.usbmodem146114202
  baud: 9600
- connected: true
  id: 000683759358
  platform: nrf52840dk/nrf52840
  product: J-Link
  runner: nrfjprog
  serial: /dev/cu.usbmodem0006837593581
  baud: 9600
```

Windows

```
- connected: true
  id: OSHW000032254e4500128002ab98002784d1000097969900
  platform: reel_board
  product: DAPLink CMSIS-DAP
  runner: pyocd
  serial: COM1
  baud: 9600
- connected: true
  id: 000683759358
  platform: nrf52840dk/nrf52840
  product: J-Link
  runner: nrfjprog
  serial: COM2
  baud: 9600
```

The baud entry is only needed if not running at 115200.

If the map file already exists, then new entries are added and existing entries will be updated. This way you can use one single master hardware map and update it for every run to get the correct serial devices and status of the devices.

With the hardware map ready, you can run any tests by pointing to the map

Linux

```
./scripts/twister --device-testing --hardware-map map.yml -T samples/hello_world/
```

Windows

```
python .\scripts\twister --device-testing --hardware-map map.yml -T samples\hello_world
```

The above command will result in twister building tests for the platforms defined in the hardware map and subsequently flashing and running the tests on those platforms.

Note

Currently only boards with support for pyocd, nrfjprog, jlink, openocd, or dediprog are supported with the hardware map features. Boards that require other runners to flash the Zephyr binary are still work in progress.

Hardware map allows to set `--device-flash-timeout` and `--device-flash-with-test` command line options as `flash-timeout` and `flash-with-test` fields respectively. These hardware map values override command line options for the particular platform.

Serial PTY support using `--device-serial-pty` can also be used in the hardware map:

```
- connected: true
  id: None
  platform: intel_adsp/cavs25
  product: None
  runner: intel_adsp
  serial_pty: path/to/script.py
  runner_params:
    - --remote-host=remote_host_ip_addr
    - --key=/path/to/key.pem
```

The runner_params field indicates the parameters you want to pass to the west runner. For some boards the west runner needs some extra parameters to work. It is equivalent to following west and twister commands.

Linux

```
west flash --remote-host remote_host_ip_addr --key /path/to/key.pem

twister -p intel_adsp/cavs25 --device-testing --device-serial-pty script.py
--west-flash="--remote-host=remote_host_ip_addr,--key=/path/to/key.pem"
```

Windows

Note

Not supported on Windows OS

Note

For serial PTY, the "–generate-hardware-map" option cannot scan it out and generate a correct hardware map automatically. You have to edit it manually according to above example. This is because the serial port of the PTY is not fixed and being allocated in the system at runtime.

Fixtures Some tests require additional setup or special wiring specific to the test. Running the tests without this setup or test fixture may fail. A test scenario can specify the fixture it needs which can then be matched with hardware capability of a board and the fixtures it supports via the command line or using the hardware map file.

Fixtures are defined in the hardware map file as a list:

```
- connected: true
  fixtures:
    - gpio_loopback
  id: 0240000026334e450015400f5e0e000b4eb1000097969900
  platform: frdm_k64f
  product: DAPLink CMSIS-DAP
  runner: pyocd
  serial: /dev/ttyACM9
```

When running `twister` with `--device-testing`, the configured fixture in the hardware map file will be matched to test scenarios requesting the same fixtures and these tests will be executed on the boards that provide this fixture.

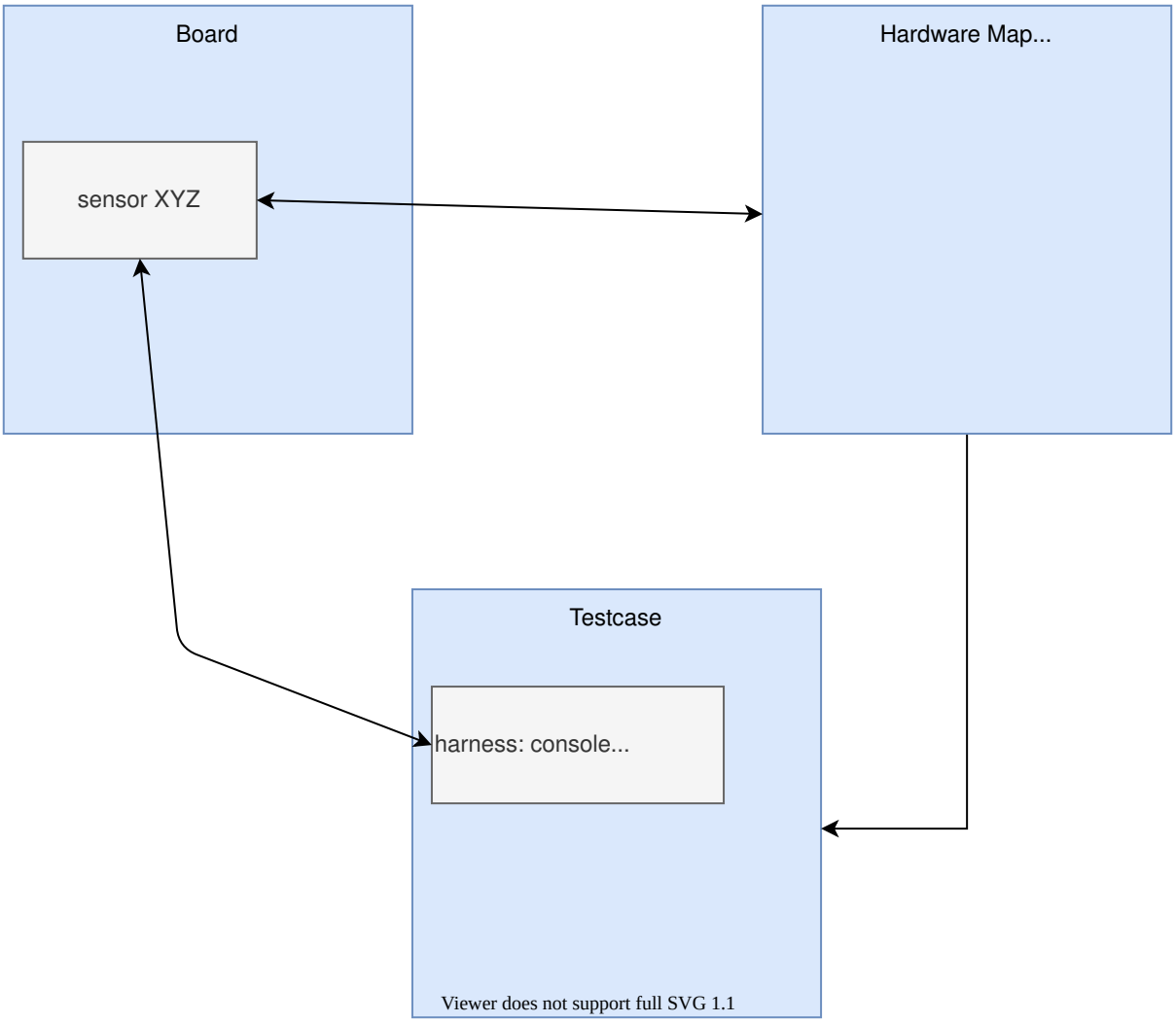
Fixtures can also be provided via twister command option `--fixture`, this option can be used multiple times and all given fixtures will be appended as a list. And the given fixtures will be assigned to all boards, this means that all boards set by current twister command can run those test scenarios which request the same fixtures.

Some fixtures allow for configuration strings to be appended, separated from the fixture name by a `:`. Only the fixture name is matched against the fixtures requested by test scenarios.

Notes It may be useful to annotate board descriptions in the hardware map file with additional information. Use the `notes` keyword to do this. For example:

```
- connected: false
  fixtures:
    - gpio_loopback
  id: 000683290670
  notes: An nrf5340dk/nrf5340 is detected as an nrf52840dk/nrf52840 with no serial
    port, and three serial ports with an unknown platform.  The board id of the serial
    ports is not the same as the board id of the development kit.  If you regenerate
    this file you will need to update serial to reference the third port, and platform
```

(continues on next page)


Board
sensor XYZ
Hardware Map...
Testcase
harness: console...
Viewer does not support full SVG 1.1

(continued from previous page)

```
    to nrf5340dk/nrf5340/cpuapp or another supported board target.
  platform: nrf52840dk/nrf52840
  product: J-Link
  runner: jlink
  serial: null
```

Overriding Board Identifier When (re-)generated the hardware map file will contain an `id` keyword that serves as the argument to `--board-id` when flashing. In some cases the detected ID is not the correct one to use, for example when using an external J-Link probe. The `probe_id` keyword overrides the `id` keyword for this purpose. For example:

```
- connected: false
  id: 0229000005d9ebc600000000000000000000000097969905
  platform: mimxrt1060_evk
  probe_id: 000609301751
  product: DAPLink CMSIS-DAP
  runner: jlink
  serial: null
```

Quarantine Twister allows user to provide configuration files defining a list of tests or platforms to be put under quarantine. Such tests will be skipped and marked accordingly in the output reports. This feature is especially useful when running larger test suits, where a failure of one test can affect the execution of other tests (e.g. putting the physical board in a corrupted state).

To use the quarantine feature one has to add the argument `--quarantine-list <PATH_TO_QUARANTINE_YAML>` to a twister call. Multiple quarantine files can be used. The current status of tests on the quarantine list can also be verified by adding `--quarantine-verify` to the above argument. This will make twister skip all tests which are not on the given list.

A quarantine yaml has to be a sequence of dictionaries. Each dictionary has to have `scenarios` and `platforms` entries listing combinations of scenarios and platforms to put under quarantine. In addition, an optional entry `comment` can be used, where some more details can be given (e.g. link to a reported issue). These comments will also be added to the output reports.

When quarantining a class of tests or many scenarios in a single testsuite or when dealing with multiple issues within a subsystem, it is possible to use regular expressions, for example, **kernel.*** would quarantine all kernel tests.

An example of entries in a quarantine yaml:

```
- scenarios:
    - sample.basic.helloworld
  comment: "Link to the issue: https://github.com/zephyrproject-rtos/zephyr/pull/33287"

- scenarios:
    - kernel.common
    - kernel.common.(misra|tls)
    - kernel.common.nano64
  platforms:
    - .*_cortex_.*
    - native_sim
```

To exclude a platform, use the following syntax:

```
- platforms:
  - qemu_x86
  comment: "broken qemu"
```

Additionally you can quarantine entire architectures or a specific simulator for executing tests.

Test Configuration

A test configuration can be used to customize various aspects of twister and the default enabled options and features. This allows tweaking the filtering capabilities depending on the environment and makes it possible to adapt and improve coverage when targeting different sets of platforms.

The test configuration also adds support for test levels and the ability to assign a specific test to one or more levels. Using command line options of twister it is then possible to select a level and just execute the tests included in this level.

Additionally, the test configuration allows defining level dependencies and additional inclusion of tests into a specific level if the test itself does not have this information already.

In the configuration file you can include complete components using regular expressions and you can specify which test level to import from the same file, making management of levels easier.

To help with testing outside of upstream CI infrastructure, additional options are available in the configuration file, which can be hosted locally. As of now, those options are available:

- Ability to ignore default platforms as defined in board definitions (Those are mostly emulation platforms used to run tests in upstream CI)
- Option to specify your own list of default platforms overriding what upstream defines.
- Ability to override *build_on_all* options used in some test scenarios. This will treat tests or sample as any other just build for default platforms you specify in the configuration file or on the command line.
- Ignore some logic in twister to expand platform coverage in cases where default platforms are not in scope.

Platform Configuration The following options control platform filtering in twister:

- *override_default_platforms*: override default key a platform sets in board configuration and instead use the list of platforms provided in the configuration file as the list of default platforms. This option is set to False by default.
- *increased_platform_scope*: This option is set to True by default, when disabled, twister will not increase platform coverage automatically and will only build and run tests on the specified platforms.
- *default_platforms*: A list of additional default platforms to add. This list can either be used to replace the existing default platforms or can extend it depending on the value of *override_default_platforms*.

And example platforms configuration:

```
platforms:
  override_default_platforms: true
  increased_platform_scope: false
  default_platforms:
    - qemu_x86
```

Test Level Configuration The test configuration allows defining test levels, level dependencies and additional inclusion of tests into a specific test level if the test itself does not have this information already.

In the configuration file you can include complete components using regular expressions and you can specify which test level to import from the same file, making management of levels simple.

And example test level configuration:

```
levels:
  - name: my-test-level
    description: >
      my custom test level
    adds:
      - kernel.threads.*
      - kernel.timer.behavior
      - arch.interrupt
      - boards.*
```

Combined configuration To mix the Platform and level configuration, you can take an example as below:

An example platforms plus level configuration:

```
platforms:
  override_default_platforms: true
  default_platforms:
    - frdm_k64f
levels:
  - name: smoke
    description: >
        A plan to be used verifying basic zephyr features.
  - name: unit
    description: >
        A plan to be used verifying unit test.
  - name: integration
    description: >
        A plan to be used verifying integration.
  - name: acceptance
    description: >
        A plan to be used verifying acceptance.
  - name: system
    description: >
        A plan to be used verifying system.
  - name: regression
    description: >
        A plan to be used verifying regression.
```

To run with above test_config.yaml file, only default_platforms with given test level test scenarios will run.

Linux

```
scripts/twister --test-config=<path to>/test_config.yaml
 -T tests --level="smoke"
```

Running in Tests in Random Order

Enable ZTEST framework's `CONFIG_ZTEST_SHUFFLE` config option to run your tests in random order. This can be beneficial for identifying dependencies between test cases. For native_sim platforms, you can provide the seed to the random number generator by providing `-seed=value` as an argument to twister. See *Shuffling Test Sequence* for more details.

Robot Framework Tests

Zephyr supports Robot Framework as one of solutions for automated testing.

Robot files allow you to express interactive test scenarios in human-readable text format and execute them in simulation or against hardware. At this moment Zephyr integration supports running Robot tests in the Renode simulation framework.

To execute a Robot test suite with twister, run the following command:

Linux

```
$ ./scripts/twister --platform hifive1 --test samples/subsys/shell/shell_module/sample.
↪shell.shell_module.robot
```

Windows

```
python .\scripts\twister --platform hifive1 --test samples/subsys/shell/shell_module/sample.
↪shell.shell_module.robot
```

Writing Robot tests For the list of keywords provided by the Robot Framework itself, refer to the official Robot documentation.

Information on writing and running Robot Framework tests in Renode can be found in the testing section of Renode documentation. It provides a list of the most commonly used keywords together with links to the source code where those are defined.

It's possible to extend the framework by adding new keywords expressed directly in Robot test suite files, as an external Python library or, like Renode does it, dynamically via XML-RPC. For details see the extending Robot Framework section in the official Robot documentation.

Running a single testsuite To run a single testsuite instead of a whole group of test you can run:

```
$ twister -p qemu_riscv32 -s tests/kernel/interrupt/arch.shared_interrupt
```

2.12.3 Twister blackbox tests

This guide aims to explain the structure of a test file so the reader will be able to understand existing files and create their own. All developers should fix any tests they break and create new ones when introducing new features, so this knowledge is important for any Twister developer.

Basics

Twister blackbox tests are written in python, using the `pytest` library. Read up on it *here* . Auxiliary test data follows whichever format it was in originally. Tests and data are wholly contained in the scripts/tests/twister_blackbox directory and prepended with `test_`.

Blackbox tests should not be aware of the internal twister code. Instead, they should call twister as user would and check the results.

Sample test file

```python
#!/usr/bin/env python3
# Copyright (c) 2024 Intel Corporation
#
# SPDX-License-Identifier: Apache-2.0

import importlib
import mock
import os
import pytest
import sys
import json

from conftest import ZEPHYR_BASE, TEST_DATA, testsuite_filename_mock
from twisterlib.testplan import TestPlan


class TestDummy:
    TESTDATA_X = [
        ("smoke", 5),
        ("acceptance", 6),
    ]

    @classmethod
    def setup_class(cls):
        apath = os.path.join(ZEPHYR_BASE, "scripts", "twister")
        cls.loader = importlib.machinery.SourceFileLoader("__main__", apath)
        cls.spec = importlib.util.spec_from_loader(cls.loader.name, cls.loader)
        cls.twister_module = importlib.util.module_from_spec(cls.spec)

    @classmethod
    def teardown_class(cls):
        pass

    @pytest.mark.parametrize(
        "level, expected_tests", TESTDATA_X, ids=["smoke", "acceptance"]
    )
    @mock.patch.object(TestPlan, "TESTSUITE_FILENAME", testsuite_filename_mock)
    def test_level(self, capfd, out_path, level, expected_tests):
        # Select platforms used for the tests
        test_platforms = ["qemu_x86", "frdm_k64f"]
        # Select test root
        path = os.path.join(TEST_DATA, "tests")
        config_path = os.path.join(TEST_DATA, "test_config.yaml")

        # Set flags for our Twister command as a list of strs
        args = (
            # Flags related to the generic test setup:
            # * Control the level of detail in stdout/err
            # * Establish the output directory
            # * Select Zephyr tests to use
            # * Control whether to only build or build and run aforementioned tests
            ["-i", "--outdir", out_path, "-T", path, "-y"]
            # Flags under test
            + ["--level", level]
            # Flags required for the test
            + ["--test-config", config_path]
            # Flags related to platform selection
            + [
                val
                for pair in zip(["-p"] * len(test_platforms), test_platforms)
                for val in pair
            ]
```

(continues on next page)

(continued from previous page)

```
        )

        # First, provide the args variable as our Twister command line arguments.
        # Then, catch the exit code in the sys_exit variable.
        with mock.patch.object(sys, "argv", [sys.argv[0]] + args), pytest.raises(
            SystemExit
        ) as sys_exit:
            # Execute the Twister call itself.
            self.loader.exec_module(self.twister_module)

        # Check whether the Twister call succeeded
        assert str(sys_exit.value) == "0"

        # Access to the test file output
        with open(os.path.join(out_path, "testplan.json")) as f:
            j = json.load(f)
        filtered_j = [
            (ts["platform"], ts["name"], tc["identifier"])
            for ts in j["testsuites"]
            for tc in ts["testcases"]
            if "reason" not in tc
        ]

        # Read stdout and stderr to out and err variables respectively
        out, err = capfd.readouterr()
        # Rewrite the captured buffers to stdout and stderr so the user can still read them
        sys.stdout.write(out)
        sys.stderr.write(err)

        # Test-relevant checks
        assert expected_tests == len(filtered_j)
```

Comparison with CLI

Test above runs the command

```
twister -i --outdir $OUTDIR -T $TEST_DATA/tests -y --level $LEVEL
--test-config $TEST_DATA/test_config.yaml -p qemu_x86 -p frdm_k64f
```

It presumes a CLI with the `zephyr-env.sh` or `zephyr-env.cmd` already run.

Such a test provides us with all the outputs we typically expect of a Twister run thanks to `importlib`'s `exec_module()`[1] . We can easily set up all flags that we expect from a Twister call via `args` variable[2] . We can check the standard output or stderr in `out` and `err` variables.

Beside the standard outputs, we can also investigate the file outputs, normally placed in `twister-out` directories. Most of the time, we will use the `out_path` fixture in conjunction with `--outdir` flag (L52) to keep test-generated files in temporary directories. Typical files read in blackbox tests are `testplan.json` , `twister.xml` and `twister.log` .

Other functionalities

Decorators

- `@pytest.mark.usefixtures('clear_log')`

[1] Take note of the `setup_class()` class function, which allows us to run `twister` python file as if it were called directly (bypassing the `__name__ == '__main__'` check).

[2] We advise you to keep the first section of `args` definition intact in almost all of your tests, as it is used for the common test setup.

  - allows us to use `clear_log` fixture from `conftest.py` . The fixture is to become `autouse` in the future. After that, this decorator can be removed.

- `@pytest.mark.parametrize('level, expected_tests', TESTDATA_X, ids=['smoke', 'acceptance'])`
  - this is an example of `pytest` 's test parametrization. Read up on it here. TESTDATAs are most often declared as class fields.
- `@mock.patch.object(TestPlan, 'TESTSUITE_FILENAME', testsuite_filename_mock)`
  - this decorator allows us to use only tests defined in the `test_data` and ignore the Zephyr testcases in the `tests` directory. **Note that all "test_data" tests use `test_data.yaml` as a filename, not `testcase.yaml` !** Read up on the `mock` library here.

Fixtures Blackbox tests use `pytest` 's fixtures, further reading on which is available here.

If you would like to add your own fixtures, consider whether they will be used in just one test file, or in many.

- If in many, create such a fixture in the scripts/tests/twister_blackbox/conftest.py file.
  - scripts/tests/twister_blackbox/conftest.py already contains some fixtures - take a look there for an example.
- If in just one, declare it in that file.
  - Consider using class fields instead - look at TESTDATAs for an example.

How do I...

Call Twister multiple times in one test? Sometimes we want to test something that requires prior Twister use. `--test-only` flag would be a typical example, as it is to be coupled with previous `--build-only` Twister call. How should we approach that?

If we just call the `importlib` 's `exec_module` two times, we will experience log duplication. `twister.log` will duplicate every line (triplicate if we call it three times, etc.) instead of overwriting the log or appending to the end of it.

It is caused by the use of logger module variables in the Twister files. Thus us executing the module again causes the loggers to have multiple handles.

To overcome this, between the calls you ought to use

```
capfd.readouterr()   # To remove output from the buffer
                     # Note that if you want output from all runs after each other,
                     # skip this line.
clear_log_in_test()  # To remove log duplication
```

2.12.4 Integration with pytest test framework

Please mind that integration of twister with pytest is still work in progress. Not every platform type is supported in pytest (yet). If you find any issue with the integration or have an idea for an improvement, please, let us know about it and open a GitHub issue/enhancement.

Introduction

Pytest is a python framework that *"makes it easy to write small, readable tests, and can scale to support complex functional testing for applications and libraries"* (https://docs.pytest.org/en/7.3.x/). Python is known for its free libraries and ease of using it for scripting. In addition, pytest utilizes the concept of plugins and fixtures, increasing its expendability and reusability. A pytest plugin *pytest-twister-harness* was introduced to provide an integration between pytest and twister, allowing Zephyr's community to utilize pytest functionality with keeping twister as the main framework.

Integration with twister

By default, there is nothing to be done to enable pytest support in twister. The plugin is developed as a part of Zephyr's tree. To enable install-less operation, twister first extends `PYTHONPATH` with path to this plugin, and then during pytest call, it appends the command with `-p twister_harness.plugin` argument. If one prefers to use the installed version of the plugin, they must add `--allow-installed-plugin` flag to twister's call.

Pytest-based test suites are discovered the same way as other twister tests, i.e., by a presence of test/sample.yaml. Inside, a keyword `harness` tells twister how to handle a given test. In the case of `harness: pytest`, most of twister workflow (test suites discovery, parallelization, building and reporting) remains the same as for other harnesses. The change happens during the execution step. The below picture presents a simplified overview of the integration.

If `harness: pytest` is used, twister delegates the test execution to pytest, by calling it as a subprocess. Required parameters (such as build directory, device to be used, etc.) are passed through a CLI command. When pytest is done, twister looks for a pytest report (results.xml) and sets the test result accordingly.

How to create a pytest test

An example folder containing a pytest test, application source code and Twister configuration .yaml file can look like the following:

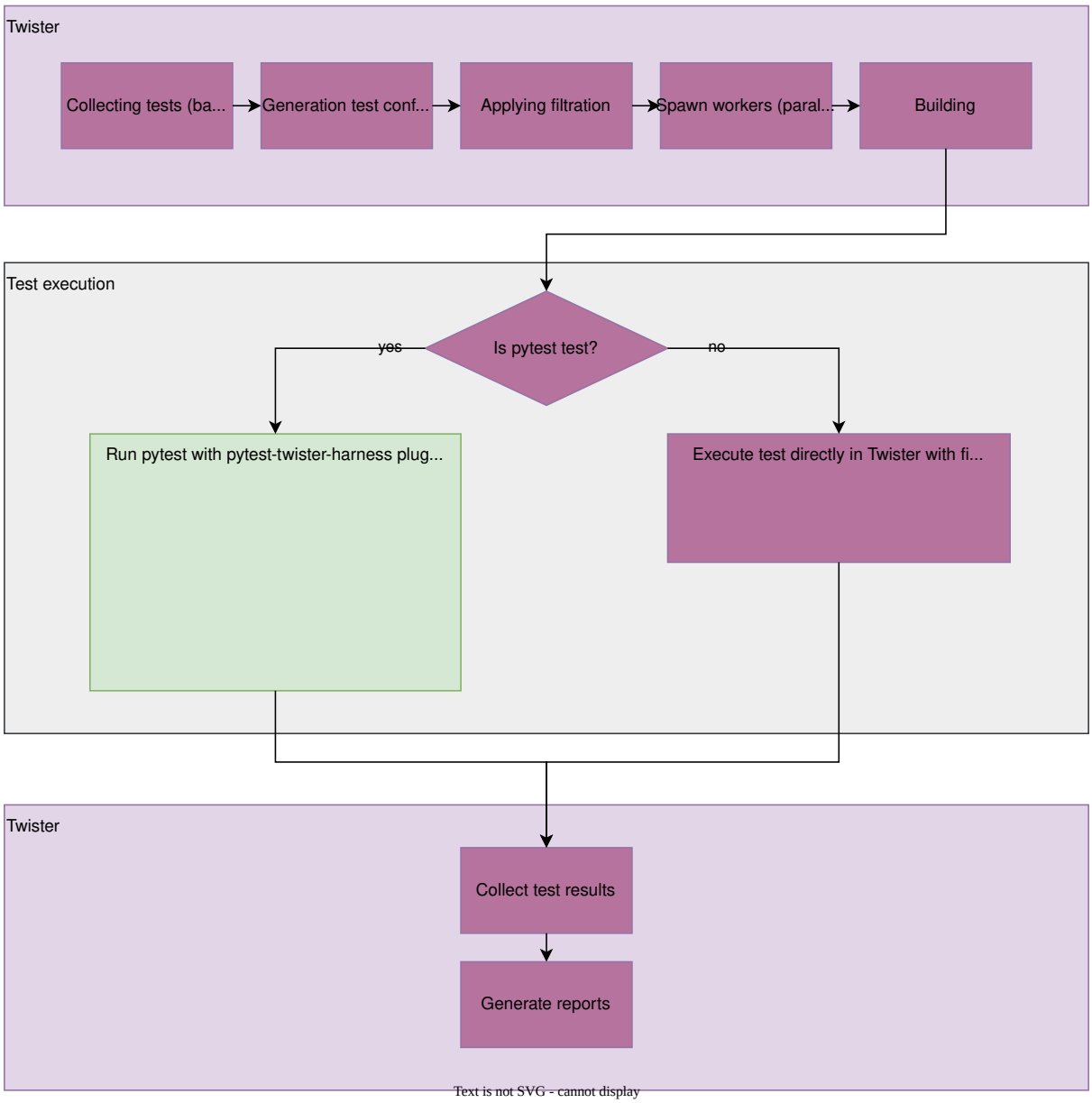
```
test_foo/
├── pytest/
│   └── test_foo.py
├── src/
│   └── main.c
├── CMakeList.txt
├── prj.conf
└── testcase.yaml
```

An example of a pytest test is given at samples/subsys/testsuite/pytest/shell/pytest/test_shell.py. Using the configuration provided in the `testcase.yaml` file, Twister builds the application from `src` and then, if the .yaml file contains a `harness: pytest` entry, it calls pytest in a separate subprocess. A sample configuration file may look like this:

```
tests:
  some.foo.test:
    harness: pytest
    tags: foo
```

By default, pytest tries to look for tests in a `pytest` directory located next to a directory with binary sources. A keyword `pytest_root` placed under `harness_config` section in .yaml file can be used to point to other files, directories or subtests (more info *here*).

Pytest scans the given locations looking for tests, following its default discovery rules.


Twister
Collecting tests (ba...
Generation test conf...
Applying filtration
Spawn workers (paral...
Building
Test execution
Is pytest test?
yes
no
Run pytest with pytest-twister-harness plug...
Execute test directly in Twister with fi...
Twister
Collect test results
Generate reports
Text is not SVG - cannot display

Passing extra arguments There are two ways for passing extra arguments to the called pytest subprocess:

1. From .yaml file, using `pytest_args` placed under `harness_config` section - more info *here*.
2. Through Twister command line interface as `--pytest-args` argument. This can be particularly useful when one wants to select a specific testcase from a test suite. For instance, one can use a command:

```
$ ./scripts/twister --platform native_sim -T samples/subsys/testsuite/pytest/shell \
-s samples/subsys/testsuite/pytest/shell/sample.pytest.shell \
--pytest-args='-k test_shell_print_version'
```

Fixtures

dut Give access to a *DeviceAdapter* type object, that represents Device Under Test. This fixture is the core of pytest harness plugin. It is required to launch DUT (initialize logging, flash device, connect serial etc). This fixture yields a device prepared according to the requested type (`native`, `qemu`, `hardware`, etc.). All types of devices share the same API. This allows for writing tests which are device-type-agnostic. Scope of this fixture is determined by the `pytest_dut_scope` keyword placed under `harness_config` section (more info *here*).

```
from twister_harness import DeviceAdapter

def test_sample(dut: DeviceAdapter):
   dut.readlines_until('Hello world')
```

shell Provide a *Shell* class object with methods used to interact with shell application. It calls `wait_for_promt` method, to not start scenario until DUT is ready. The shell fixture calls `dut` fixture, hence has access to all its methods. The `shell` fixture adds methods optimized for interactions with a shell. It can be used instead of `dut` for tests. Scope of this fixture is determined by the `pytest_dut_scope` keyword placed under `harness_config` section (more info *here*).

```
from twister_harness import Shell

def test_shell(shell: Shell):
   shell.exec_command('help')
```

mcumgr Sample fixture to wrap `mcumgr` command-line tool used to manage remote devices. More information about MCUmgr can be found here *MCUmgr*.

> **Note**
>
> This fixture requires the `mcumgr` available in the system PATH

Only selected functionality of MCUmgr is wrapped by this fixture. For example, here is a test with a fixture `mcumgr`

```
from twister_harness import DeviceAdapter, Shell, McuMgr

def test_upgrade(dut: DeviceAdapter, shell: Shell, mcumgr: McuMgr):
   # free the serial port for mcumgr
   dut.disconnect()
   # upload the signed image
   mcumgr.image_upload('path/to/zephyr.signed.bin')
```

(continues on next page)

(continued from previous page)

```
    # obtain the hash of uploaded image from the device
    second_hash = mcumgr.get_hash_to_test()
    # test a new upgrade image
    mcumgr.image_test(second_hash)
    # reset the device remotely
    mcumgr.reset_device()
    # continue test scenario, check version etc.
```

Classes

DeviceAdapter

class twister_harness.**DeviceAdapter**(*device_config: DeviceConfig*)

This class defines a common interface for all device types (hardware, simulator, QEMU) used in tests to gathering device output and send data to it.

launch() → None

Start by closing previously running application (no effect if not needed). Then, flash and run test application. Finally, start an internal reader thread capturing an output from a device.

connect(*retry_s: int = 0*) → None

Connect to device - allow for output gathering.

readline(*timeout: float | None = None*, *print_output: bool = True*) → str

Read line from device output. If timeout is not provided, then use base_timeout.

readlines(*print_output: bool = True*) → list[str]

Read all available output lines produced by device from internal buffer.

readlines_until(*regex: str | None = None*, *num_of_lines: int | None = None*, *timeout: float | None = None*, *print_output: bool = True*) → list[str]

Read available output lines produced by device from internal buffer until following conditions:

1. If regex is provided - read until regex regex is found in read line (or until timeout).
2. If num_of_lines is provided - read until number of read lines is equal to num_of_lines (or until timeout).
3. If none of above is provided - return immediately lines collected so far in internal buffer.

If timeout is not provided, then use base_timeout.

write(*data: bytes*) → None

Write data bytes to device.

disconnect() → None

Disconnect device - block output gathering.

close() → None

Disconnect, close device and close reader thread.

Shell

class twister_harness.**Shell**(*device:* DeviceAdapter, *prompt: str = 'uart:~$'*, *timeout: float | None = None*)

Helper class that provides methods used to interact with shell application.

`exec_command`(*command: str*, *timeout: float | None = None*, *print_output: bool = True*) → list[str]

Send shell command to a device and return response. Passed command is extended by double enter sings - first one to execute this command on a device, second one to receive next prompt what is a signal that execution was finished. Method returns printout of the executed command.

`wait_for_prompt`(*timeout: float | None = None*) → bool

Send every 0.5 second "enter" command to the device until shell prompt statement will occur (return True) or timeout will be exceeded (return False).

Examples of pytest tests in the Zephyr project

- pytest_shell
- MCUmgr tests - tests/boot/with_mcumgr
- LwM2M tests - tests/net/lib/lwm2m/interop
- GDB stub tests - tests/subsys/debug/gdbstub

FAQ

How to flash/run application only once per pytest session?

`dut` is a fixture responsible for flashing/running application. By default, its scope is set as `function`. This can be changed by adding to .yaml file `pytest_dut_scope` keyword placed under `harness_config` section:

```
harness: pytest
harness_config:
   pytest_dut_scope: session
```

More info can be found *here*.

How to run only one particular test from a python file?

This can be achieved in several ways. In .yaml file it can be added using a `pytest_root` entry placed under `harness_config` with list of tests which should be run:

```
harness: pytest
harness_config:
   pytest_root:
      - "pytest/test_shell.py::test_shell_print_help"
```

Particular tests can be also chosen by pytest `-k` option (more info about pytest keyword filter can be found here). It can be applied by adding `-k` filter in `pytest_args` in .yaml file:

```
harness: pytest
harness_config:
   pytest_args:
      - "-k test_shell_print_help"
```

or by adding it to Twister command overriding parameters from the .yaml file:

```
$ ./scripts/twister ... --pytest-args='-k test_shell_print_help'
```

How to get information about used device type in test?

This can be taken from `dut` fixture (which represents *DeviceAdapter* object):

```
device_type: str = dut.device_config.type
if device_type == 'hardware':
    ...
elif device_type == 'native':
    ...
```

How to rerun locally pytest tests without rebuilding application by Twister?

This can be achieved by running Twister once again with `--test-only` argument added to Twister command. Another way is running Twister with highest verbosity level (`-vv`) and then copy-pasting from logs command dedicated for spawning pytest (log started by `Running pytest command: ...`).

Is this possible to run pytest tests in parallel?

Basically `pytest-harness-plugin` wasn't written with intention of running pytest tests in parallel. Especially those one dedicated for hardware. There was assumption that parallelization of tests is made by Twister, and it is responsible for managing available sources (jobs and hardwares). If anyone is interested in doing this for some reasons (for example via pytest-xdist plugin) they do so at their own risk.

Limitations

- Not every platform type is supported in the plugin (yet).

2.12.5 Generating coverage reports

With Zephyr, you can generate code coverage reports to analyze which parts of the code are covered by a given test or application.

You can do this in two ways:

- In a real embedded target or QEMU, using Zephyr's gcov integration
- Directly in your host computer, by compiling your application targeting the POSIX architecture

Test coverage reports in embedded devices or QEMU

Overview GCC GCOV is a test coverage program used together with the GCC compiler to analyze and create test coverage reports for your programs, helping you create more efficient, faster running code and discovering untested code paths

In Zephyr, gcov collects coverage profiling data in RAM (and not to a file system) while your application is running. Support for gcov collection and reporting is limited by available RAM size and so is currently enabled only for QEMU emulation of embedded targets.

Details There are 2 parts to enable this feature. The first is to enable the coverage for the device and the second to enable in the test application. As explained earlier the code coverage with gcov is a function of RAM available. Therefore ensure that the device has enough RAM when enabling the coverage for it. For example a small device like frdm_k64f can run a simple test application but the more complex test cases which consume more RAM will crash when coverage is enabled.

To enable the device for coverage, select `CONFIG_HAS_COVERAGE_SUPPORT` in the Kconfig.board file.

To report the coverage for the particular test application set `CONFIG_COVERAGE`.

Steps to generate code coverage reports These steps will produce an HTML coverage report for a single application.

1. Build the code with CONFIG_COVERAGE=y.

   ```
   west build -b mps2/an385 -- -DCONFIG_COVERAGE=y -DCONFIG_COVERAGE_DUMP=y
   ```

2. Capture the emulator output into a log file. You may need to terminate the emulator with `Ctrl-A X` for this to complete after the coverage dump has been printed:

   ```
   ninja -Cbuild run | tee log.log
   ```

   or

   ```
   ninja -Cbuild run | tee log.log
   ```

3. Generate the gcov `.gcda` and `.gcno` files from the log file that was saved:

   ```
   $ python3 scripts/gen_gcov_files.py -i log.log
   ```

4. Find the gcov binary placed in the SDK. You will need to pass the path to the gcov binary for the appropriate architecture when you later invoke `gcovr`:

   ```
   $ find $ZEPHYR_SDK_INSTALL_DIR -iregex ".*gcov"
   ```

5. Create an output directory for the reports:

   ```
   $ mkdir -p gcov_report
   ```

6. Run `gcovr` to get the reports:

   ```
   $ gcovr -r $ZEPHYR_BASE . --html -o gcov_report/coverage.html --html-details --gcov-
   ↪executable <gcov_path_in_SDK>
   ```

Coverage reports using the POSIX architecture

When compiling for the POSIX architecture, you utilize your host native tooling to build a native executable which contains your application, the Zephyr OS, and some basic HW emulation.

That means you can use the same tools you would while developing any other desktop application.

To build your application with gcc's gcov, simply set `CONFIG_COVERAGE` before compiling it. When you run your application, `gcov` coverage data will be dumped into the respective `gcda` and `gcno` files. You may postprocess these with your preferred tools. For example:

```
west build -b native_sim samples/hello_world -- -DCONFIG_COVERAGE=y
```

```
$ ./build/zephyr/zephyr.exe
# Press Ctrl+C to exit
lcov --capture --directory ./ --output-file lcov.info -q --rc lcov_branch_coverage=1
genhtml lcov.info --output-directory lcov_html -q --ignore-errors source --branch-coverage -
↪-highlight --legend
```

Note

You need a recent version of lcov (at least 1.14) with support for intermediate text format. Such packages exist in recent Linux distributions.

Alternatively, you can use gcovr (at least version 4.2).

Coverage reports using Twister

Zephyr's *twister script* can automatically generate a coverage report from the tests which were executed. You just need to invoke it with the `--coverage` command line option.

For example, you may invoke:

```
$ twister --coverage -p qemu_x86 -T tests/kernel
```

or:

```
$ twister --coverage -p native_sim -T tests/bluetooth
```

which will produce `twister-out/coverage/index.html` report as well as the coverage data collected by `gcovr` tool in `twister-out/coverage.json`.

Other reports might be chosen with `--coverage-tool` and `--coverage-formats` command line options.

The process differs for unit tests, which are built with the host toolchain and require a different board:

```
$ twister --coverage -p unit_testing -T tests/unit
```

which produces a report in the same location as non-unit testing.

Using different toolchains Twister looks at the environment variable `ZEPHYR_TOOLCHAIN_VARIANT` to check which gcov tool to use by default. The following are used as the default for the Twister `--gcov-tool` argument default:

Toolchain	`--gcov-tool` value
host	`gcov`
llvm	`llvm-cov gcov`
zephyr	`gcov`

2.12.6 BabbleSim

BabbleSim and Zephyr

In the Zephyr project we use the Babblesim simulator to test some of the Zephyr radio protocols, including the BLE stack, 802.15.4, and some of the networking stack.

BabbleSim is a physical layer simulator, which in combination with the Zephyr bsim boards can be used to simulate a network of BLE and 15.4 devices. When we build Zephyr targeting a bsim board we produce a Linux executable, which includes the application, Zephyr OS, and models of the HW.

When there is radio activity, this Linux executable will connect to the BabbleSim Phy simulation to simulate the radio channel.

In the BabbleSim documentation you can find more information on how to get and build the simulator. In the nrf52_bsim, nrf5340bsim, and nrf54l15bsim boards documentation you can find more information about how to build Zephyr targeting these particular boards, and a few examples.

Types of tests

Tests without radio activity: bsim tests with twister The bsim boards can be used without radio activity, and in that case, it is not necessary to connect them to a physical layer simulation. Thanks to this, these target boards can be used just like native_sim with *twister*, to run all standard Zephyr twister tests, but with models of a real SOC HW, and their drivers.

Tests with radio activity When there is radio activity, BabbleSim tests require at the very least a physical layer simulation running, and most, more than 1 simulated device. Due to this, these tests are not build and run with twister, but with a dedicated set of tests scripts.

These tests are kept in the `tests/bsim/` folder. The `compile.sh` and `run_parallel.sh` scripts contained in that folder are used by the CI system to build the needed images and execute these tests in batch.

See sections below for more information about how to build and run them, as well as the conventions they follow.

There are two main sets of tests:

- Self checking embedded application/tests: In which some of the simulated devices applications are built with some checks which decide if the test is passing or failing. These embedded applications tests use the bs_tests system to report the pass or failure, and in many cases to build several tests into the same binary.
- Test using the EDTT tool, in which a EDTT (python) test controls the embedded applications over an RPC mechanism, and decides if the test passes or not. Today these tests include a very significant subset of the BT qualification test suite.

More information about how different tests types relate to BabbleSim and the bsim boards can be found in the bsim boards tests section.

Test coverage and BabbleSim

As the nrf52_bsim and nrf5340bsim, and nrf54l15bsim boards are based on the POSIX architecture, you can easily collect test coverage information.

You can use the script tests/bsim/generate_coverage_report.sh to generate an html coverage report from tests.

Check *the page on coverage generation* for more info.

Building and running the tests

See the nrf52_bsim page for setting up the simulator.

The scripts also expect a few environment variables to be set. For example, from Zephyr's root folder, you can run:

```
# Build all the tests
${ZEPHYR_BASE}/tests/bsim/compile.sh

# Run them (in parallel)
RESULTS_FILE=${ZEPHYR_BASE}/myresults.xml \
   SEARCH_PATH=${ZEPHYR_BASE}/tests/bsim \
      ${ZEPHYR_BASE}/tests/bsim/run_parallel.sh
```

Or to build and run only a specific subset, e.g. host advertising tests:

```
# Build the Bluetooth host advertising tests
${ZEPHYR_BASE}/tests/bsim/bluetooth/host/adv/compile.sh

# Run them (in parallel)
RESULTS_FILE=${ZEPHYR_BASE}/myresults.xml \
   SEARCH_PATH=${ZEPHYR_BASE}/tests/bsim/bluetooth/host/adv \
      ${ZEPHYR_BASE}/tests/bsim/run_parallel.sh
```

Check the `run_parallel.sh` help for more options and examples on how to use this script to run the tests in batch.

After building the tests' required binaries you can run a test directly using its individual test script.

For example you can build the required binaries for the networking tests with

```
WORK_DIR=${ZEPHYR_BASE}/bsim_out ${ZEPHYR_BASE}/tests/bsim/net/compile.sh
```

and then directly run one of the tests:

```
${ZEPHYR_BASE}/tests/bsim/net/sockets/echo_test/tests_scripts/echo_test_802154.sh
```

Conventions

Test code See the Bluetooth sample test for conventions that apply to test code.

Build scripts The build scripts `compile.sh` simply build all the required test and sample applications for the tests' scripts placed in the subfolders below.

This build scripts use the common compile.source which provide a function (compile) which calls cmake and ninja with the provided application, configuration and overlay files.

To speed up compilation for users interested only in a subset of tests, several compile scripts exist in several subfolders, where the upper ones call into the lower ones.

Note that cmake and ninja are used directly instead of the `west build` wrapper as west is not required, and some Zephyr users do not use or have west, but still use the build and tests scripts.

Test scripts Please follow the existing conventions and do not design one-off bespoke runners (e.g. a python script, or another shell abstraction).

The rationale is that it is easier and faster for the maintainers to perform tree-wide updates for build system or compatibility changes if the tests are run in the same manner, with the same variables, etc..

If you have a good idea for improving your test script, please make a PR changing *all* the test scripts in order to benefit everyone and conserve homogeneity. You can of course discuss it first in an RFC issue or on the babblesim discord channel.

Scripts starting with an underscore (_) are not automatically discovered and run. They can serve as either helper functions for the main script, or can be used for local development utilities, e.g. building and running tests locally, debugging, etc..

Here are the conventions:

- Each test is defined by a shell script with the extension `.sh`, in a subfolder called `test_scripts/`.
- It is recommended to run a single test per script file. It allows for better parallelization of the runs in CI.
- Scripts expect that the binaries they require are already built. They should not compile binaries.
- Scripts will spawn the processes for every simulated device and the physical layer simulation.
- Scripts must return 0 to the invoking shell if the test passes, and not 0 if the test fails.
- Each test must have a unique simulation id, to enable running different tests in parallel.
- Neither the scripts nor the images should modify the workstation filesystem content beyond the `${BSIM_OUT_PATH}/results/<simulation_id>/` or `/tmp/` folders. That is, they should not leave stray files behind.
- Tests that require several consecutive simulations (e.g, if simulating a device pairing, powering off, and powering up after as a new simulation) should use separate simulation ids for each simulation segment, ensuring that the radio activity of each segment can be inspected a posteriori.
- Avoid overly long tests. If the test takes over 20 seconds of runtime, consider if it is possible to split it in several separate tests.
- If the test takes over 5 seconds, set `EXECUTE_TIMEOUT` to a value that is at least 5 times bigger than the measured run-time.
- Do not set `EXECUTE_TIMEOUT` to a value lower than the default.
- Tests should not be overly verbose: less than a hundred lines are expected on the outputs. Do make use of `LOG_DBG()` extensively, but don't enable the `DBG` log level by default.

2.12.7 ZTest Deprecated APIs

Ztest is currently being migrated to a new API, this documentation provides information about the deprecated APIs which will eventually be removed. See *Test Framework* for the new API. Similarly, ZTest's mocking framework is also deprecated (see *Mocking via FFF*).

Quick start - Unit testing

Ztest can be used for unit testing. This means that rather than including the entire Zephyr OS for testing a single function, you can focus the testing efforts into the specific module in question. This will speed up testing since only the module will have to be compiled in, and the tested functions will be called directly.

Since you won't be including basic kernel data structures that most code depends on, you have to provide function stubs in the test. Ztest provides some helpers for mocking functions, as demonstrated below.

In a unit test, mock objects can simulate the behavior of complex real objects and are used to decide whether a test failed or passed by verifying whether an interaction with an object occurred, and if required, to assert the order of that interaction.

Best practices for declaring the test suite *twister* and other validation tools need to obtain the list of subcases that a Zephyr *ztest* test image will expose.

Rationale

This all is for the purpose of traceability. It's not enough to have only a semaphore test project. We also need to show that we have testpoints for all APIs and functionality, and we trace back to documentation of the API, and functional requirements.

The idea is that test reports show results for every sub-testcase as passed, failed, blocked, or skipped. Reporting on only the high-level test project level, particularly when tests do too many things, is too vague.

There exist two alternatives to writing tests. The first, and more verbose, approach is to directly declare and run the test suites. Here is a generic template for a test showing the expected use of `ztest_test_suite()`:

```
#include <zephyr/ztest.h>

extern void test_sometest1(void);
extern void test_sometest2(void);
#ifndef CONFIG_WHATEVER              /* Conditionally skip test_sometest3 */
void test_sometest3(void)
{
     ztest_test_skip();
}
#else
extern void test_sometest3(void);
#endif
extern void test_sometest4(void);
...

void test_main(void)
{
     ztest_test_suite(common,
                      ztest_unit_test(test_sometest1),
                      ztest_unit_test(test_sometest2),
                      ztest_unit_test(test_sometest3),
                      ztest_unit_test(test_sometest4)
                );
     ztest_run_test_suite(common);
}
```

Alternatively, it is possible to split tests across multiple files using `ztest_register_test_suite()` which bypasses the need for `extern`:

```
#include <zephyr/ztest.h>

void test_sometest1(void) {
      zassert_true(1, "true");
}

ztest_register_test_suite(common, NULL,
                          ztest_unit_test(test_sometest1)
                          );
```

The above sample simple registers the test suite and uses a NULL pragma function (more on that later). It is important to note that the test suite isn't directly run in this file. Instead two alternatives exist for running the suite. First, if to do nothing. A default test_main function is provided by ztest. This is the preferred approach if the test doesn't involve a state and doesn't require use of the pragma.

In cases of an integration test it is possible that some general state needs to be set between test suites. This can be thought of as a state diagram in which test_main simply goes through various actions that modify the board's state and different test suites need to run. This is achieved in the following:

```
#include <zephyr/ztest.h>

struct state {
	bool is_hibernating;
	bool is_usb_connected;
}

static bool pragma_always(const void *state)
{
	return true;
}

static bool pragma_not_hibernating_not_connected(const void *s)
{
	struct state *state = s;
	return !state->is_hibernating && !state->is_usb_connected;
}

static bool pragma_usb_connected(const void *s)
{
	return ((struct state *)s)->is_usb_connected;
}

ztest_register_test_suite(baseline, pragma_always,
                          ztest_unit_test(test_case0));
ztest_register_test_suite(before_usb, pragma_not_hibernating_not_connected,
                          ztest_unit_test(test_case1),
                          ztest_unit_test(test_case2));
ztest_register_test_suite(with_usb, pragma_usb_connected,,
                          ztest_unit_test(test_case3),
                          ztest_unit_test(test_case4));

void test_main(void)
{
	struct state state;

	/* Should run `baseline` test suite only. */
	ztest_run_registered_test_suites(&state);

	/* Simulate power on and update state. */
	emulate_power_on();
	/* Should run `baseline` and `before_usb` test suites. */
	ztest_run_registered_test_suites(&state);

	/* Simulate plugging in a USB device. */
	emulate_plugging_in_usb();
	/* Should run `baseline` and `with_usb` test suites. */
	ztest_run_registered_test_suites(&state);

	/* Verify that all the registered test suites actually ran. */
	ztest_verify_all_registered_test_suites_ran();
}
```

For *twister* to parse source files and create a list of subcases, the declarations of `ztest_test_suite()` and `ztest_register_test_suite()` must follow a few rules:

- one declaration per line
- conditional execution by using *ztest_test_skip()*

What to avoid:

- packing multiple testcases in one source file

```
void test_main(void)
{
#ifdef TEST_feature1
        ztest_test_suite(feature1,
                         ztest_unit_test(test_1a),
                         ztest_unit_test(test_1b),
                         ztest_unit_test(test_1c)
                         );
        ztest_run_test_suite(feature1);
#endif

#ifdef TEST_feature2
        ztest_test_suite(feature2,
                         ztest_unit_test(test_2a),
                         ztest_unit_test(test_2b)
                         );
        ztest_run_test_suite(feature2);
#endif
}
```

- Do not use `#if`

```
        ztest_test_suite(common,
                         ztest_unit_test(test_sometest1),
                         ztest_unit_test(test_sometest2),
#ifdef CONFIG_WHATEVER
                         ztest_unit_test(test_sometest3),
#endif
                         ztest_unit_test(test_sometest4),
        ...
```

- Do not add comments on lines with a call to *ztest_unit_test()*:

```
ztest_test_suite(common,
                 ztest_unit_test(test_sometest1),
                 ztest_unit_test(test_sometest2) /* will fail */,
/* will fail! */ ztest_unit_test(test_sometest3),
                 ztest_unit_test(test_sometest4),
...
```

- Do not define multiple definitions of unit / user unit test case per line

```
ztest_test_suite(common,
                 ztest_unit_test(test_sometest1), ztest_unit_test(test_sometest2),
                 ztest_unit_test(test_sometest3),
                 ztest_unit_test(test_sometest4),
...
```

Other questions:

- Why not pre-scan with CPP and then parse? or post scan the ELF file?

  If C pre-processing or building fails because of any issue, then we won't be able to tell the subcases.

- Why not declare them in the YAML testcase description?

  A separate testcase description file would be harder to maintain than just keeping the information in the test source files themselves – only one file to update when changes are made eliminates duplication.

Mocking

These functions allow abstracting callbacks and related functions and controlling them from specific tests. You can enable the mocking framework by setting `CONFIG_ZTEST_MOCKING` to "y" in the configuration file of the test. The amount of concurrent return values and expected parameters is limited by `CONFIG_ZTEST_PARAMETER_COUNT`.

Here is an example for configuring the function `expect_two_parameters` to expect the values `a=2` and `b=3`, and telling `returns_int` to return 5:

```
#include <zephyr/ztest.h>

static void expect_two_parameters(int a, int b)
{
        ztest_check_expected_value(a);
        ztest_check_expected_value(b);
}

static void parameter_tests(void)
{
        ztest_expect_value(expect_two_parameters, a, 2);
        ztest_expect_value(expect_two_parameters, b, 3);
        expect_two_parameters(2, 3);
}

static int returns_int(void)
{
        return ztest_get_return_value();
}

static void return_value_tests(void)
{
        ztest_returns_value(returns_int, 5);
        zassert_equal(returns_int(), 5, NULL);
}

void test_main(void)
{
        ztest_test_suite(mock_framework_tests,
                ztest_unit_test(parameter_test),
                ztest_unit_test(return_value_test)
        );

        ztest_run_test_suite(mock_framework_tests);
}
```

group **ztest_mock**

This module provides simple mocking functions for unit testing.

These need CONFIG_ZTEST_MOCKING=y.

Defines

`ztest_expect_value`(func, param, value)

Tell function *func* to expect the value *value* for *param*.

When using *ztest_check_expected_value()*, tell that the value of *param* should be *value*. The value will internally be stored as an `uintptr_t`.

Parameters

- `func` – Function in question
- `param` – Parameter for which the value should be set
- `value` – Value for *param*

`ztest_check_expected_value`(param)

If *param* doesn't match the value set by *ztest_expect_value()*, fail the test.

This will first check that does *param* have a value to be expected, and then checks whether the value of the parameter is equal to the expected value. If either of these checks fail, the current test will fail. This must be called from the called function.

Parameters

- `param` – Parameter to check

`ztest_expect_data`(func, param, data)

Tell function *func* to expect the data *data* for *param*.

When using *ztest_check_expected_data()*, the data pointed to by *param* should be same *data* in this function. Only data pointer is stored by this function, so it must still be valid when ztest_check_expected_data is called.

Parameters

- `func` – Function in question
- `param` – Parameter for which the data should be set
- `data` – pointer for the data for parameter *param*

`ztest_check_expected_data`(param, length)

If data pointed by *param* don't match the data set by *ztest_expect_data()*, fail the test.

This will first check that *param* is expected to be null or non-null and then check whether the data pointed by parameter is equal to expected data. If either of these checks fail, the current test will fail. This must be called from the called function.

Parameters

- `param` – Parameter to check
- `length` – Length of the data to compare

`ztest_return_data`(func, param, data)

Tell function *func* to return the data *data* for *param*.

When using *ztest_return_data()*, the data pointed to by *param* should be same *data* in this function. Only data pointer is stored by this function, so it must still be valid when ztest_copy_return_data is called.

Parameters

- `func` – Function in question
- `param` – Parameter for which the data should be set
- `data` – pointer for the data for parameter *param*

`ztest_copy_return_data`(param, length)

Copy the data set by ztest_return_data to the memory pointed by *param*.

This will first check that *param* is not null and then copy the data. This must be called from the called function.

Parameters

- `param` – Parameter to return data for
- `length` – Length of the data to return

`ztest_returns_value`(func, value)

Tell *func* that it should return *value*.

Parameters

- `func` – Function that should return *value*
- `value` – Value to return from *func*

`ztest_get_return_value`()

Get the return value for current function.

The return value must have been set previously with *ztest_returns_value()*. If no return value exists, the current test will fail.

Returns

The value the current function should return

`ztest_get_return_value_ptr`()

Get the return value as a pointer for current function.

The return value must have been set previously with *ztest_returns_value()*. If no return value exists, the current test will fail.

Returns

The value the current function should return as a `void *`

2.13 Static Code Analysis (SCA)

Support for static code analysis tools in Zephyr is possible through CMake.

The build setting `ZEPHYR_SCA_VARIANT` can be used to specify the SCA tool to use. `ZEPHYR_SCA_VARIANT` is also supported as *environment variable*.

Use `-DZEPHYR_SCA_VARIANT=<tool>`, for example `-DZEPHYR_SCA_VARIANT=sparse` to enable the static analysis tool `sparse`.

2.13.1 SCA Tool infrastructure

Support for an SCA tool is implemented in a file:*sca.cmake* file. The file:*sca.cmake* must be placed under file:*<SCA_ROOT>/cmake/sca/<tool>/sca.cmake*. Zephyr itself is always added as an `SCA_ROOT` but the build system offers the possibility to add additional folders to the `SCA_ROOT` setting.

You can provide support for out of tree SCA tools by creating the following structure:

```
<sca_root>/                 # Custom SCA root
└── cmake/
    └── sca/
        └── <tool>/         # Name of SCA tool, this is the value given to ZEPHYR_SCA_
```

(continues on next page)

(continued from previous page)

```
↪VARIANT
            └── sca.cmake   # CMake code that configures the tool to be used with Zephyr
```

To add `foo` under `/path/to/my_tools/cmake/sca` create the following structure:

```
/path/to/my_tools
         └── cmake/
             └── sca/
                 └── foo/
                     └── sca.cmake
```

To use `foo` as SCA tool you must then specify `-DZEPHYR_SCA_VARIANT=foo`.

Remember to add `/path/to/my_tools` to **SCA_ROOT**.

SCA_TOOL can be set as a regular CMake setting using `-DSCA_ROOT=<sca_root>`, or added by a Zephyr module in its `module.yml` file, see *Zephyr Modules - Build settings*

2.13.2 Native SCA Tool support

The following is a list of SCA tools natively supported by Zephyr build system.

CodeChecker support

CodeChecker is a static analysis infrastructure. It executes analysis tools available on the build system, such as Clang-Tidy, Clang Static Analyzer and Cppcheck. Refer to the analyzer's websites for installation instructions.

Installing CodeChecker CodeChecker itself is a python package available on pypi.

```
pip install codechecker
```

Running with CodeChecker To run CodeChecker, *west build* should be called with a `-DZEPHYR_SCA_VARIANT=codechecker` parameter, e.g.

```
west build -b mimxrt1064_evk samples/basic/blinky -- -DZEPHYR_SCA_VARIANT=codechecker
```

Configuring CodeChecker To configure CodeChecker or analyzers used, arguments can be passed using the `CODECHECKER_ANALYZE_OPTS` parameter, e.g.

```
west build -b mimxrt1064_evk samples/basic/blinky -- -DZEPHYR_SCA_VARIANT=codechecker \
-DCODECHECKER_ANALYZE_OPTS="--config;$CODECHECKER_CONFIG_FILE;--timeout;60"
```

Storing CodeChecker results If a CodeChecker server is active the results can be uploaded and stored for tracking purposes. Storing is done using the optional `CODECHECKER_STORE=y` or `CODECHECKER_STORE_OPTS="arg;list"` parameters, e.g.

```
west build -b mimxrt1064_evk samples/basic/blinky -- -DZEPHYR_SCA_VARIANT=codechecker \
-DCODECHECKER_STORE_OPTS="--name;build;--url;localhost:8001/Default"
```

> **Note**
>
> If `--name` isn't passed to either `CODECHECKER_ANALYZE_OPTS` or `CODECHECKER_STORE_OPTS`, the default `zephyr` is used.

Exporting CodeChecker reports Optional reports can be generated using the CodeChecker results, when passing a `-DCODECHECKER_EXPORT=<type>` parameter. Allowed types are: `html`, `json`, `codeclimate`, `gerrit`, `baseline`. Multiple types can be passed as comma-separated arguments.

Optional parser configuration arguments can be passed using the `CODECHECKER_PARSE_OPTS` parameter, e.g.

```
west build -b mimxrt1064_evk samples/basic/blinky -- -DZEPHYR_SCA_VARIANT=codechecker \
-DCODECHECKER_EXPORT=html,json -DCODECHECKER_PARSE_OPTS="--trim-path-prefix;$PWD"
```

Failing the build on CodeChecker issues By default, CodeChecker identified issues will not fail the build, only generate a report. To fail the build if any issues are found (for example, for use in CI), pass the `CODECHECKER_PARSE_EXIT_STATUS=y` parameter, e.g.

```
west build -b mimxrt1064_evk samples/basic/blinky -- -DZEPHYR_SCA_VARIANT=codechecker \
-DCODECHECKER_PARSE_EXIT_STATUS=y
```

Sparse support

Sparse is a static code analysis tool. Apart from performing common code analysis tasks it also supports an `address_space` attribute, which allows introduction of distinct address spaces in C code with subsequent verification that pointers to different address spaces do not get confused. Additionally it supports a `force` attribute which should be used to cast pointers between different address spaces. At the moment Zephyr introduces a single custom address space `__cache` used to identify pointers from the cached address range on the Xtensa architecture. This helps identify cases where cached and uncached addresses are confused.

Running with sparse To run a sparse verification build *west build* should be called with a `-DZEPHYR_SCA_VARIANT=sparse` parameter, e.g.

```
west build -d hello -b intel_adsp/cavs25 zephyr/samples/hello_world -- -DZEPHYR_SCA_
↪VARIANT=sparse
```

GCC static analysis support

Static analysis was introduced in GCC 10 and it is enabled with the option `-fanalyzer`. This option performs a much more expensive and thorough analysis of the code than traditional warnings.

Run GCC static analysis To run GCC static analysis, *west build* should be called with a `-DZEPHYR_SCA_VARIANT=gcc` parameter, e.g.

```
west build -b qemu_x86 samples/userspace/hello_world_user -- -DZEPHYR_SCA_VARIANT=gcc
```

Parasoft C/C++test support

Parasoft C/C++test is a software testing and static analysis tool for C and C++. It is a commercial software and you must acquire a commercial license to use it.

Documentation of C/C++test can be found at https://docs.parasoft.com/. Please refer to the documentation for how to use it.

Generating Build Data Files To use C/C++test, `cpptestscan` must be found in your *PATH* environment variable. And *west build* should be called with a `-DZEPHYR_SCA_VARIANT=cpptest` parameter, e.g.

```
west build -b qemu_cortex_m3 zephyr/samples/hello_world -- -DZEPHYR_SCA_VARIANT=cpptest
```

A `.bdf` file will be generated as `build/sca/cpptest/cpptestscan.bdf`.

Generating a report file Please refer to Parasoft C/C++test documentation for more details.

To import and generate a report file, something like the following should work.

```
cpptestcli -data out -localsettings local.conf -bdf build/sca/cpptest/cpptestscan.bdf -
↪config "builtin://Recommended Rules" -report out/report
```

You might need to set `bdf.import.c.compiler.exec`, `bdf.import.cpp.compiler.exec`, and `bdf.import.linker.exec` to the toolchain *west build* used.

2.14 Toolchains

Guides on how to set up toolchains for Zephyr development.

2.14.1 Zephyr SDK

The Zephyr Software Development Kit (SDK) contains toolchains for each of Zephyr's supported architectures. It also includes additional host tools, such as custom QEMU and OpenOCD.

Use of the Zephyr SDK is highly recommended and may even be required under certain conditions (for example, running tests in QEMU for some architectures).

Supported architectures

The Zephyr SDK supports the following target architectures:

- ARC (32-bit and 64-bit; ARCv1, ARCv2, ARCv3)
- ARM (32-bit and 64-bit; ARMv6, ARMv7, ARMv8; A/R/M Profiles)
- MIPS (32-bit and 64-bit)
- Nios II
- RISC-V (32-bit and 64-bit; RV32I, RV32E, RV64I)
- x86 (32-bit and 64-bit)
- Xtensa

Installation bundle and variables

The Zephyr SDK bundle supports all major operating systems (Linux, macOS and Windows) and is delivered as a compressed file. The installation consists of extracting the file and running the included setup script. Additional OS-specific instructions are described in the sections below.

If no toolchain is selected, the build system looks for Zephyr SDK and uses the toolchain from there. You can enforce this by setting the environment variable *ZEPHYR_TOOLCHAIN_VARIANT* to `zephyr`.

If you install the Zephyr SDK outside any of the default locations (listed in the operating system specific instructions below) and you want automatic discovery of the Zephyr SDK, then you must register the Zephyr SDK in the CMake package registry by running the setup script. If you decide not to register the Zephyr SDK in the CMake registry, then the *ZEPHYR_SDK_INSTALL_DIR* can be used to point to the Zephyr SDK installation directory.

You can also set *ZEPHYR_SDK_INSTALL_DIR* to point to a directory containing multiple Zephyr SDKs, allowing for automatic toolchain selection. For example, you can set `ZEPHYR_SDK_INSTALL_DIR` to `/company/tools`, where the `company/tools` folder contains the following subfolders:

- `/company/tools/zephyr-sdk-0.13.2`
- `/company/tools/zephyr-sdk-a.b.c`
- `/company/tools/zephyr-sdk-x.y.z`

This allows the Zephyr build system to choose the correct version of the SDK, while allowing multiple Zephyr SDKs to be grouped together at a specific path.

Zephyr SDK version compatibility

In general, the Zephyr SDK version referenced in this page should be considered the recommended version for the corresponding Zephyr version.

For the full list of compatible Zephyr and Zephyr SDK versions, refer to the Zephyr SDK Version Compatibility Matrix.

Zephyr SDK installation

Note

You can change `0.16.8` to another version in the instructions below if needed; the Zephyr SDK Releases page contains all available SDK releases.

Note

If you want to uninstall the SDK, you may simply remove the directory where you installed it.

Ubuntu

1. Download and verify the Zephyr SDK bundle:

```
cd ~
wget https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.16.8/
↪zephyr-sdk-0.16.8_linux-x86_64.tar.xz
```

```
wget -O - https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.
↪16.8/sha256.sum | shasum --check --ignore-missing
```

If your host architecture is 64-bit ARM (for example, Raspberry Pi), replace `x86_64` with `aarch64` in order to download the 64-bit ARM Linux SDK.

2. Extract the Zephyr SDK bundle archive:

```
tar xvf zephyr-sdk-0.16.8_linux-x86_64.tar.xz
```

> **Note**
>
> It is recommended to extract the Zephyr SDK bundle at one of the following locations:
>
> - `$HOME`
> - `$HOME/.local`
> - `$HOME/.local/opt`
> - `$HOME/bin`
> - `/opt`
> - `/usr/local`
>
> The Zephyr SDK bundle archive contains the `zephyr-sdk-<version>` directory and, when extracted under `$HOME`, the resulting installation path will be `$HOME/zephyr-sdk-<version>`.

3. Run the Zephyr SDK bundle setup script:

```
cd zephyr-sdk-0.16.8
./setup.sh
```

> **Note**
>
> You only need to run the setup script once after extracting the Zephyr SDK bundle.
>
> You must rerun the setup script if you relocate the Zephyr SDK bundle directory after the initial setup.

4. Install udev rules, which allow you to flash most Zephyr boards as a regular user:

```
sudo cp ~/zephyr-sdk-0.16.8/sysroots/x86_64-pokysdk-linux/usr/share/openocd/
↪contrib/60-openocd.rules /etc/udev/rules.d
sudo udevadm control --reload
```

macOS

1. Download and verify the Zephyr SDK bundle:

```
cd ~
curl -L -O https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.
↪16.8/zephyr-sdk-0.16.8_macos-x86_64.tar.xz
curl -L https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.16.
↪8/sha256.sum | shasum --check --ignore-missing
```

If your host architecture is 64-bit ARM (Apple Silicon), replace `x86_64` with `aarch64` in order to download the 64-bit ARM macOS SDK.

2. Extract the Zephyr SDK bundle archive:

```
tar xvf zephyr-sdk-0.16.8_macos-x86_64.tar.xz
```

> **Note**
>
> It is recommended to extract the Zephyr SDK bundle at one of the following locations:
>
> - `$HOME`
> - `$HOME/.local`
> - `$HOME/.local/opt`
> - `$HOME/bin`
> - `/opt`
> - `/usr/local`
>
> The Zephyr SDK bundle archive contains the `zephyr-sdk-<version>` directory and, when extracted under `$HOME`, the resulting installation path will be `$HOME/zephyr-sdk-<version>`.

3. Run the Zephyr SDK bundle setup script:

```
cd zephyr-sdk-0.16.8
./setup.sh
```

> **Note**
>
> You only need to run the setup script once after extracting the Zephyr SDK bundle.
>
> You must rerun the setup script if you relocate the Zephyr SDK bundle directory after the initial setup.

Windows

1. Open a `cmd.exe` terminal window **as a regular user**
2. Download the Zephyr SDK bundle:

```
cd %HOMEPATH%
wget https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.16.8/
↪zephyr-sdk-0.16.8_windows-x86_64.7z
```

3. Extract the Zephyr SDK bundle archive:

```
7z x zephyr-sdk-0.16.8_windows-x86_64.7z
```

> **Note**
>
> It is recommended to extract the Zephyr SDK bundle at one of the following locations:
>
> - `%HOMEPATH%`
> - `%PROGRAMFILES%`
>
> The Zephyr SDK bundle archive contains the `zephyr-sdk-<version>` directory and, when extracted under `%HOMEPATH%`, the resulting installation path will be `%HOMEPATH%\zephyr-sdk-<version>`.

4. Run the Zephyr SDK bundle setup script:

```
cd zephyr-sdk-0.16.8
setup.cmd
```

> **Note**
>
> You only need to run the setup script once after extracting the Zephyr SDK bundle.
>
> You must rerun the setup script if you relocate the Zephyr SDK bundle directory after the initial setup.

2.14.2 Arm Compiler 6

1. Download and install a development suite containing the Arm Compiler 6 for your operating system.
2. *Set these environment variables*:
   - Set *ZEPHYR_TOOLCHAIN_VARIANT* to `armclang`.
   - Set `ARMCLANG_TOOLCHAIN_PATH` to the toolchain installation directory.
3. The Arm Compiler 6 needs the `ARMLMD_LICENSE_FILE` environment variable to point to your license file or server.

For example:

```
# Linux, macOS, license file:
export ARMLMD_LICENSE_FILE=/<path>/license_armds.dat
# Linux, macOS, license server:
export ARMLMD_LICENSE_FILE=8224@myserver
```

```
# Windows, license file:
set ARMLMD_LICENSE_FILE=c:\<path>\license_armds.dat
# Windows, license server:
set ARMLMD_LICENSE_FILE=8224@myserver
```

1. If the Arm Compiler 6 was installed as part of an Arm Development Studio, then you must set the `ARM_PRODUCT_DEF` to point to the product definition file: See also: Product and toolkit configuration. For example if the Arm Development Studio is installed in: `/opt/armds-2020-1` with a Gold license, then set `ARM_PRODUCT_DEF` to point to `/opt/armds-2020-1/gold.elmap`.

> **Note**
>
> The Arm Compiler 6 uses `armlink` for linking. This is incompatible with Zephyr's linker script template, which works with GNU ld. Zephyr's Arm Compiler 6 support Zephyr's CMake linker script generator, which supports generating scatter files. Basic scatter file support is in place, but there are still areas covered in ld templates which are not fully supported by the CMake linker script generator.
>
> Some Zephyr subsystems or modules may also contain C or assembly code that relies on GNU intrinsics and have not yet been updated to work fully with `armclang`.

2.14.3 Cadence Tensilica Xtensa C/C++ Compiler (XCC)

1. Obtain Tensilica Software Development Toolkit targeting the specific SoC on hand. This usually contains two parts:
   - The Xtensa Xplorer which contains the necessary executables and libraries.
   - A SoC-specific add-on to be installed on top of Xtensa Xplorer.

- This add-on allows the compiler to generate code for the SoC on hand.

2. Install Xtensa Xplorer and then the SoC add-on.
   - Follow the instruction from Cadence on how to install the SDK.
   - Depending on the SDK, there are two set of compilers:
     - GCC-based compiler: `xt-xcc` and its friends.
     - Clang-based compiler: `xt-clang` and its friends.
3. Make sure you have obtained a license to use the SDK, or has access to a remote licensing server.
4. *Set these environment variables*:
   - Set *ZEPHYR_TOOLCHAIN_VARIANT* to `xcc` or `xt-clang`.
   - Set `XTENSA_TOOLCHAIN_PATH` to the toolchain installation directory.
   - Set `XTENSA_CORE` to the SoC ID where application is being targeting.
   - Set `TOOLCHAIN_VER` to the Xtensa SDK version.
5. For example, assuming the SDK is installed in `/opt/xtensa`, and using the SDK for application development on `intel_adsp_cavs15`, setup the environment using:

```
# Linux
export ZEPHYR_TOOLCHAIN_VARIANT=xcc
export XTENSA_TOOLCHAIN_PATH=/opt/xtensa/XtDevTools/install/tools/
export XTENSA_CORE=X6H3SUE_RI_2018_0
export TOOLCHAIN_VER=RI-2018.0-linux
```

6. To use Clang-based compiler:
   - Set *ZEPHYR_TOOLCHAIN_VARIANT* to `xt-clang`.
   - Note that the Clang-based compiler may contain an old LLVM bug which results in the following error:

```
/tmp/file.s: Assembler messages:
/tmp/file.s:20: Error: file number 1 already allocated
clang-3.9: error: Xtensa-as command failed with exit code 1
```

     If this happens, set `XCC_NO_G_FLAG` to `1`.
     - For example:

```
# Linux
export XCC_NO_G_FLAG=1
```

2.14.4 DesignWare ARC MetaWare Development Toolkit (MWDT)

1. You need to have ARC MWDT installed on your host.
2. You need to have *Zephyr SDK* installed on your host.

> **Note**
>
> A Zephyr SDK is used as a source of tools like device tree compiler (DTC), QEMU, etc... Even though ARC MWDT toolchain is used for Zephyr RTOS build, still the GNU preprocessor & GNU objcopy might be used for some steps like device tree preprocessing and `.bin` file generation. We used Zephyr SDK as a source of these ARC GNU tools as well. To setup ARC GNU toolchain please use SDK Bundle (Full or Minimal) instead of manual

installation of separate tarballs. It installs and registers toolchain and host tools in the system, that allows you to avoid toolchain related issues while building Zephyr.

3. *Set these environment variables*:
   - Set *ZEPHYR_TOOLCHAIN_VARIANT* to `arcmwdt`.
   - Set `ARCMWDT_TOOLCHAIN_PATH` to the toolchain installation directory. MWDT installation provides `METAWARE_ROOT` so simply set `ARCMWDT_TOOLCHAIN_PATH` to `$METAWARE_ROOT/../` (Linux) or `%METAWARE_ROOT%\..\` (Windows).

Tip

If you have only one ARC MWDT toolchain version installed on your machine you may skip setting `ARCMWDT_TOOLCHAIN_PATH` - it would be detected automatically.

4. To check that you have set these variables correctly in your current environment, follow these example shell sessions (the `ARCMWDT_TOOLCHAIN_PATH` values may be different on your system):

```
# Linux:
$ echo $ZEPHYR_TOOLCHAIN_VARIANT
arcmwdt
$ echo $ARCMWDT_TOOLCHAIN_PATH
/home/you/ARC/MWDT_2023.03/

# Windows:
> echo %ZEPHYR_TOOLCHAIN_VARIANT%
arcmwdt
> echo %ARCMWDT_TOOLCHAIN_PATH%
C:\ARC\MWDT_2023.03\
```

2.14.5 GNU Arm Embedded

1. Download and install a GNU Arm Embedded build for your operating system and extract it on your file system.

Note

On Windows, we'll assume for this guide that you install into the directory `C:\gnu_arm_embedded`. You can also choose the default installation path used by the ARM GCC installer, in which case you will need to adjust the path accordingly in the guide below.

Warning

On macOS Catalina or later you might need to *change a security policy* for the toolchain to be able to run from the terminal.

2. *Set these environment variables*:
   - Set *ZEPHYR_TOOLCHAIN_VARIANT* to `gnuarmemb`.
   - Set `GNUARMEMB_TOOLCHAIN_PATH` to the toolchain installation directory.

3. To check that you have set these variables correctly in your current environment, follow these example shell sessions (the GNUARMEMB_TOOLCHAIN_PATH values may be different on your system):

```
# Linux, macOS:
$ echo $ZEPHYR_TOOLCHAIN_VARIANT
gnuarmemb
$ echo $GNUARMEMB_TOOLCHAIN_PATH
/home/you/Downloads/gnu_arm_embedded

# Windows:
> echo %ZEPHYR_TOOLCHAIN_VARIANT%
gnuarmemb
> echo %GNUARMEMB_TOOLCHAIN_PATH%
C:\gnu_arm_embedded
```

Warning

On macOS, if you are having trouble with the suggested procedure, there is an unofficial package on brew that might help you. Run `brew install gcc-arm-embedded` and configure the variables

- Set *ZEPHYR_TOOLCHAIN_VARIANT* to gnuarmemb.
- Set GNUARMEMB_TOOLCHAIN_PATH to the brew installation directory (something like /usr/local)

2.14.6 Intel oneAPI Toolkit

1. Download Intel oneAPI Base Toolkit
2. Assuming the toolkit is installed in /opt/intel/oneApi, set environment using:

```
# Linux, macOS:
export ONEAPI_TOOLCHAIN_PATH=/opt/intel/oneapi
source $ONEAPI_TOOLCHAIN_PATH/compiler/latest/env/vars.sh

# Windows:
> set ONEAPI_TOOLCHAIN_PATH=C:\Users\Intel\oneapi
```

   To setup the complete oneApi environment, use:

```
source  /opt/intel/oneapi/setvars.sh
```

   The above will also change the python environment to the one used by the toolchain and might conflict with what Zephyr uses.

3. Set *ZEPHYR_TOOLCHAIN_VARIANT* to oneApi.

2.14.7 Crosstool-NG (Deprecated)

Warning

xtools toolchain variant is deprecated. The *cross-compile toolchain variant* should be used when using a custom toolchain built with Crosstool-NG.

You can build toolchains from source code using crosstool-NG.

1. Follow the steps on the crosstool-NG website to prepare your host.
2. Follow the Zephyr SDK with Crosstool NG instructions to build your toolchain. Repeat as necessary to build toolchains for multiple target architectures.

   You will need to clone the `sdk-ng` repo and run the following command:

   ```
   ./go.sh <arch>
   ```

   Note

   Currently, only i586 and Arm toolchain builds are verified.

3. *Set these environment variables*:
   - Set *ZEPHYR_TOOLCHAIN_VARIANT* to `xtools`.
   - Set `XTOOLS_TOOLCHAIN_PATH` to the toolchain build directory.
4. To check that you have set these variables correctly in your current environment, follow these example shell sessions (the `XTOOLS_TOOLCHAIN_PATH` values may be different on your system):

   ```
   # Linux, macOS:
   $ echo $ZEPHYR_TOOLCHAIN_VARIANT
   xtools
   $ echo $XTOOLS_TOOLCHAIN_PATH
   /Volumes/CrossToolNGNew/build/output/
   ```

2.14.8 Host Toolchains

In some specific configurations, like when building for non-MCU x86 targets on a Linux host, you may be able to reuse the native development tools provided by your operating system.

To use your host gcc, set the *ZEPHYR_TOOLCHAIN_VARIANT environment variable* to `host`. To use clang, set *ZEPHYR_TOOLCHAIN_VARIANT* to `llvm`.

2.14.9 Other Cross Compilers

This toolchain variant is borrowed from the Linux kernel build system's mechanism of using a `CROSS_COMPILE` environment variable to set up a GNU-based cross toolchain.

Examples of such "other cross compilers" are cross toolchains that your Linux distribution packaged, that you compiled on your own, or that you downloaded from the net. Unlike toolchains specifically listed in *Toolchains*, the Zephyr build system may not have been tested with them, and doesn't officially support them. (Nonetheless, the toolchain set-up mechanism itself is supported.)

Follow these steps to use one of these toolchains.

1. Install a cross compiler suitable for your host and target systems.

   For example, you might install the `gcc-arm-none-eabi` package on Debian-based Linux systems, or `arm-none-eabi-newlib` on Fedora or Red Hat:

   ```
   # On Debian or Ubuntu
   sudo apt-get install gcc-arm-none-eabi
   # On Fedora or Red Hat
   sudo dnf install arm-none-eabi-newlib
   ```

2. *Set these environment variables*:
   - Set *ZEPHYR_TOOLCHAIN_VARIANT* to `cross-compile`.
   - Set `CROSS_COMPILE` to the common path prefix which your toolchain's binaries have, e.g. the path to the directory containing the compiler binaries plus the target triplet and trailing dash.
3. To check that you have set these variables correctly in your current environment, follow these example shell sessions (the `CROSS_COMPILE` value may be different on your system):

```
# Linux, macOS:
$ echo $ZEPHYR_TOOLCHAIN_VARIANT
cross-compile
$ echo $CROSS_COMPILE
/usr/bin/arm-none-eabi-
```

   You can also set `CROSS_COMPILE` as a CMake variable.

When using this option, all of your toolchain binaries must reside in the same directory and have a common file name prefix. The `CROSS_COMPILE` variable is set to the directory concatenated with the file name prefix. In the Debian example above, the `gcc-arm-none-eabi` package installs binaries such as `arm-none-eabi-gcc` and `arm-none-eabi-ld` in directory `/usr/bin/`, so the common prefix is `/usr/bin/arm-none-eabi-` (including the trailing dash, `-`). If your toolchain is installed in `/opt/mytoolchain/bin` with binary names based on target triplet `myarch-none-elf`, `CROSS_COMPILE` would be set to `/opt/mytoolchain/bin/myarch-none-elf-`.

2.14.10 Custom CMake Toolchains

To use a custom toolchain defined in an external CMake file, *set these environment variables*:

- Set *ZEPHYR_TOOLCHAIN_VARIANT* to your toolchain's name
- Set `TOOLCHAIN_ROOT` to the path to the directory containing your toolchain's CMake configuration files.

Zephyr will then include the toolchain cmake files located in the `TOOLCHAIN_ROOT` directory:

- `cmake/toolchain/<toolchain name>/generic.cmake`: configures the toolchain for "generic" use, which mostly means running the C preprocessor on the generated *Devicetree* file.
- `cmake/toolchain/<toolchain name>/target.cmake`: configures the toolchain for "target" use, i.e. building Zephyr and your application's source code.

Here <toolchain name> is the same as the name provided in *ZEPHYR_TOOLCHAIN_VARIANT* See the zephyr files cmake/modules/FindHostTools.cmake and cmake/modules/FindTargetTools.cmake for more details on what your `generic.cmake` and `target.cmake` files should contain.

You can also set `ZEPHYR_TOOLCHAIN_VARIANT` and `TOOLCHAIN_ROOT` as CMake variables when generating a build system for a Zephyr application, like so:

```
west build ... -- -DZEPHYR_TOOLCHAIN_VARIANT=... -DTOOLCHAIN_ROOT=...
```

```
cmake -DZEPHYR_TOOLCHAIN_VARIANT=... -DTOOLCHAIN_ROOT=...
```

If you do this, `-C <initial-cache>` cmake option may useful. If you save your **ZEPHYR_TOOLCHAIN_VARIANT**, **TOOLCHAIN_ROOT**, and other settings in a file named `my-toolchain.cmake`, you can then invoke cmake as `cmake -C my-toolchain.cmake ...` to save typing.

Zephyr includes `include/toolchain.h` which again includes a toolchain specific header based on the compiler identifier, such as `__llvm__` or `__GNUC__`. Some custom compilers identify themselves as the compiler on which they are based, for example `llvm` which then gets the `toolchain/llvm.h` included. This included file may though not be right for the custom

toolchain. In order to solve this, and thus to get the `include/other.h` included instead, add the set(TOOLCHAIN_USE_CUSTOM 1) cmake line to the generic.cmake and/or target.cmake files located under `<TOOLCHAIN_ROOT>/cmake/toolchain/<toolchain name>/`.

When **TOOLCHAIN_USE_CUSTOM** is set, the `other.h` must be available out-of-tree and it must include the correct header for the custom toolchain. A good location for the `other.h` header file, would be a directory under the directory specified in `TOOLCHAIN_ROOT` as `include/toolchain`. To get the toolchain header included in zephyr's build, the **USERINCLUDE** can be set to point to the include directory, as shown here:

```
west build -- -DZEPHYR_TOOLCHAIN_VARIANT=... -DTOOLCHAIN_ROOT=... -DUSERINCLUDE=...
```

2.15 Tools and IDEs

2.15.1 CLion

CLion is a cross-platform C/C++ IDE that supports multi-threaded RTOS debugging.

This guide describes the process of setting up, building, and debugging Zephyr's multi-thread-blinky sample in CLion.

The instructions have been tested on Windows. In terms of the CLion workflow, the steps would be the same for macOS and Linux, but make sure to select the correct environment file and to adjust the paths.

Get CLion

Download CLion and install it.

Initialize a new workspace

This guide gives details on how to build and debug the multi-thread-blinky sample application, but the instructions would be similar for any Zephyr project and *workspace layout*.

Before you start, make sure you have a working Zephyr development environment, as per the instructions in the *Getting Started Guide*.

Open the project in CLion

1. In CLion, click *Open* on the Welcome screen or select *File* ▸ *Open* from the main menu.
2. Navigate to your Zephyr workspace (i.e.the `zephyrproject` folder in your HOME directory if you have followed the Getting Started instructions), then select `zephyr/samples/basic/threads` or another sample project folder.

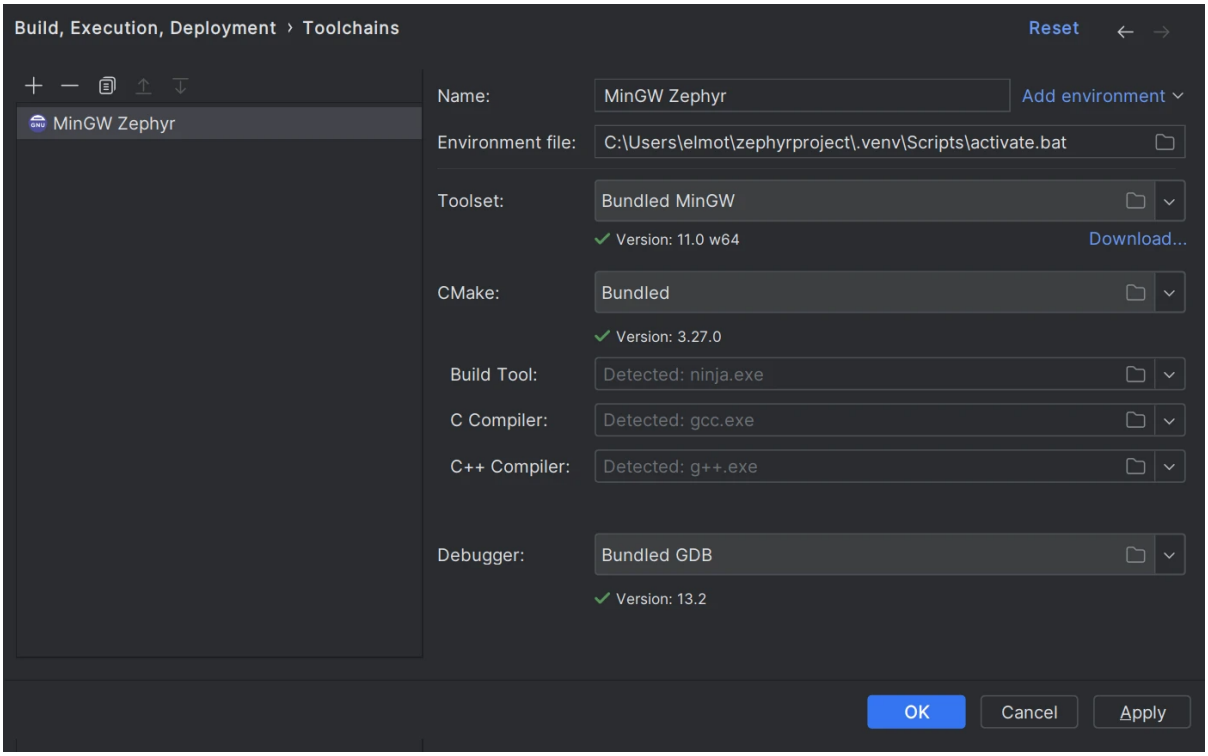
   Click *OK*.
3. If prompted, click *Trust Project*.

   See the Project security section in CLion web help for more information on project security.

Configure the toolchain and CMake profile

CLion will open the *Open Project Wizard* with the CMake profile settings. If that does not happen, go to *Settings* ▸ *Build, Execution, Deployment* ▸ *CMake*.

1. Click *Manage Toolchains* next to the *Toolchain* field. This will open the *Toolchain* settings dialog.
2. We recommend that you use the *Bundled MinGW* toolchain with default settings on Windows, or the *System* (default) toolchain on Unix machines.
3. Click *Add environment* ▸ *From file* and select `..\.venv\Scripts\activate.bat`.



   Click *Apply* to save the changes.

4. Back in the CMake profile settings dialog, specify your board in the *CMake options* field. For example:

   ```
   -DBOARD=nrf52840dk/nrf52840
   ```

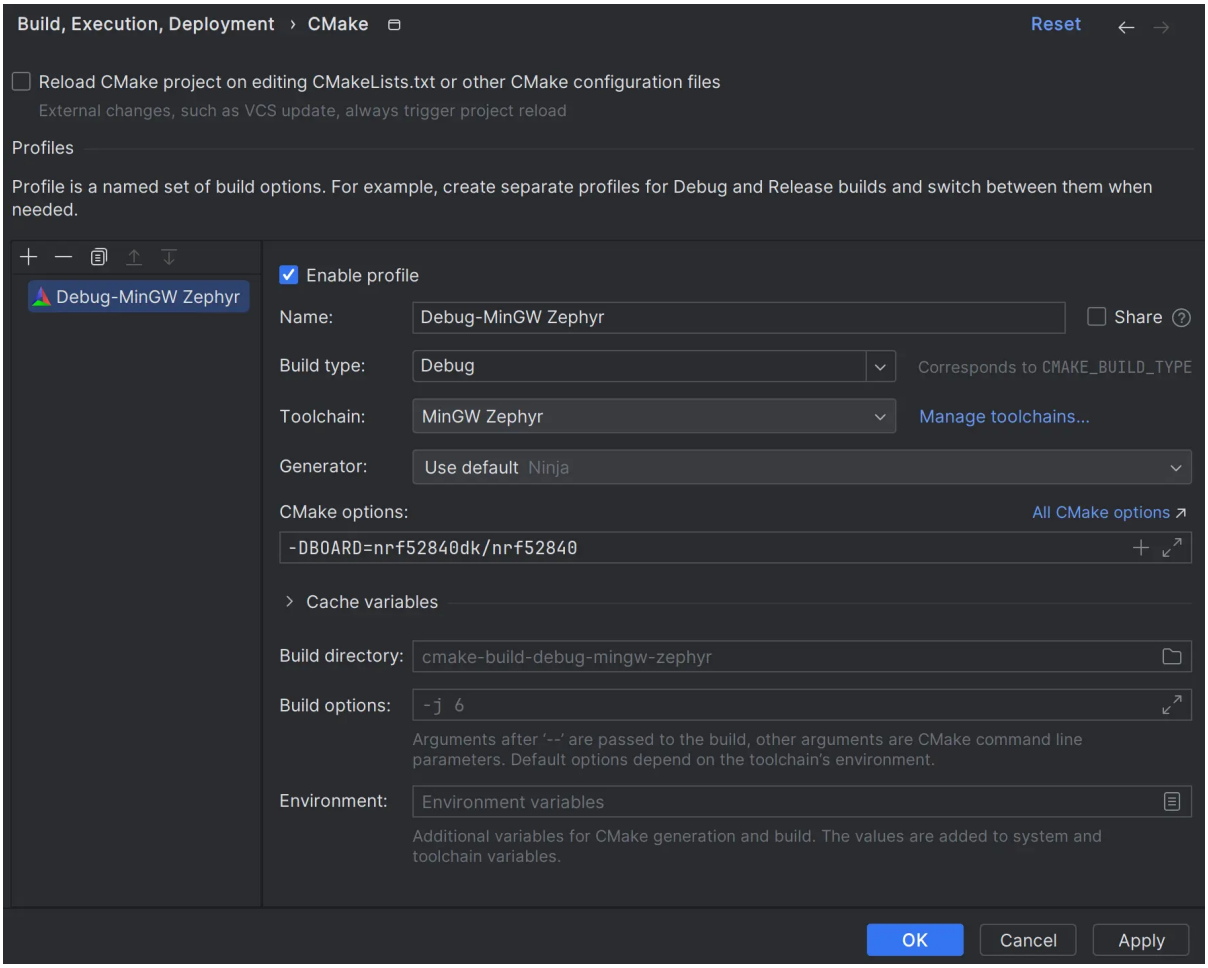
5. Click *Apply* to save the changes.

   CMake load should finish successfully.

Configure Zephyr parameters for debug

1. In the configuration switcher on the top right, select *guiconfig* and click the hammer icon.
2. Use the GUI application to set the following flags:

   ```
   DEBUG_THREAD_INFO
   THREAD_RUNTIME_STATS
   DEBUG_OPTIMIZATIONS
   ```


Build, Execution, Deployment › CMake
Reset
Reload CMake project on editing CMakeLists.txt or other CMake configuration files
External changes, such as VCS update, always trigger project reload
Profiles
Profile is a named set of build options. For example, create separate profiles for Debug and Release builds and switch between them when needed.
Debug-MinGW Zephyr
Enable profile
Name:
Debug-MinGW Zephyr
Share
Build type:
Debug
Corresponds to CMAKE_BUILD_TYPE
Toolchain:
MinGW Zephyr
Manage toolchains...
Generator:
Use default Ninja
CMake options:
All CMake options
-DBOARD=nrf52840dk/nrf52840
Cache variables
Build directory:
cmake-build-debug-mingw-zephyr
Build options:
-j 6
Arguments after '--' are passed to the build, other arguments are CMake command line parameters. Default options depend on the toolchain's environment.
Environment:
Environment variables
Additional variables for CMake generation and build. The values are added to system and toolchain variables.
OK
Cancel
Apply

Build the project

In the configuration switcher, select **zephyr_final** and click the hammer icon.

Note that other CMake targets like `puncover` or `hardenconfig` can also be called at this point.

Enable RTOS integration

1. Go to *Settings* ▸ *Build, Execution, Deployment* ▸ *Embedded Development* ▸ *RTOS Integration*.
2. Set the *Enable RTOS Integration* checkbox.

   This option enables Zephyr tasks view during debugging. See Multi-threaded RTOS debug in CLion web help for more information.

   You can leave the option set to *Auto*. CLion will detect Zephyr automatically.

Create an Embedded GDB Server configuration

In order to debug a Zephyr application in CLion, you need to create a run/debug configuration out of the Embedded GDB Server template.

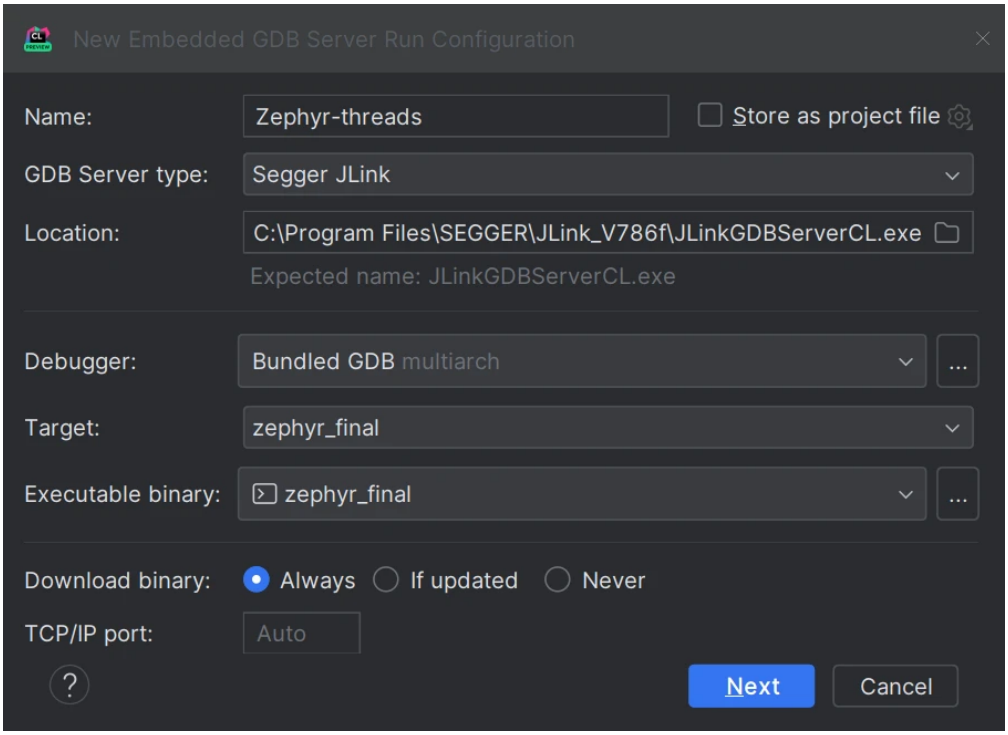
Instructions below show the case of a Nordic Semiconductor board and a Segger J-Link debug probe. If your setup is different, make sure to adjust the configuration settings accordingly.

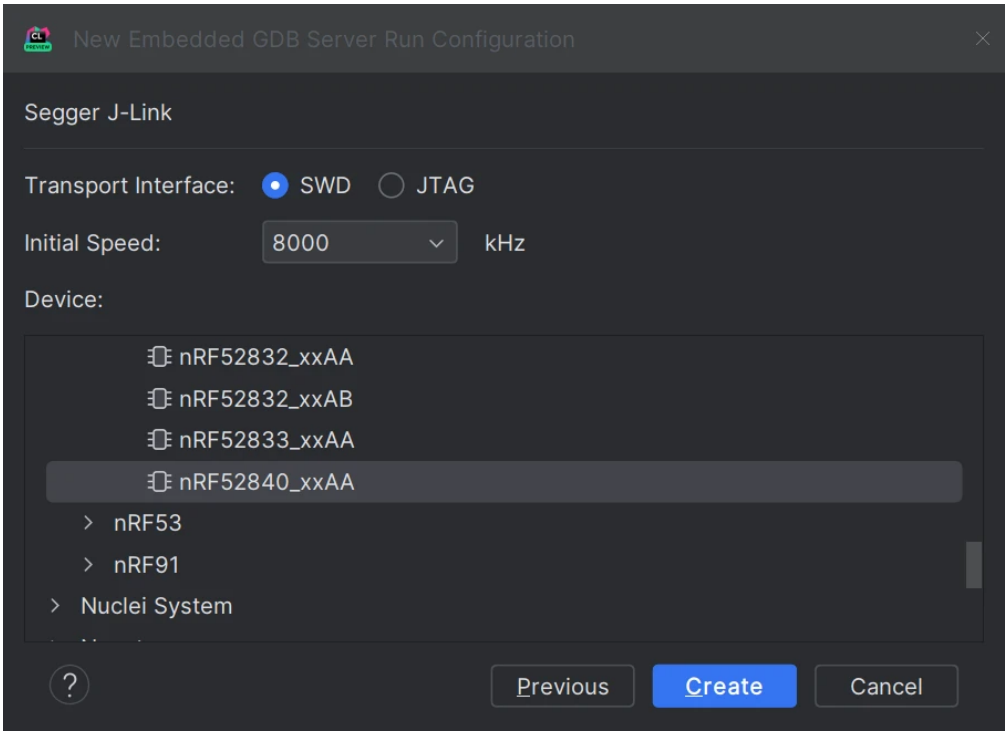
1. Select *Run* ▸ *New Embedded Configuration* from the main menu.
2. Configure the settings:

Option	Value
Name (optional)	Zephyr-threads
GDB Server Type	Segger JLink
Location	The path to `JLinkGDBServerCL.exe` on Windows or the `JLinkGDBServer` binary on macOS/Linux.
Debugger	Bundled GDB **Note** For non-ARM and non-x86 architectures, use a GDB executable from Zephyr SDK. Make sure to pick a version with Python support (for example, **riscv64-zephyr-elf-gdb-py**) and check that Python is present in the system `PATH`.
Target	zephyr-final
Executable binary	zephyr-final
Download binary	Always
TCP/IP port	Auto

3. Click *Next* to set the Segger J-Link parameters.

4. Click *Create* when ready.

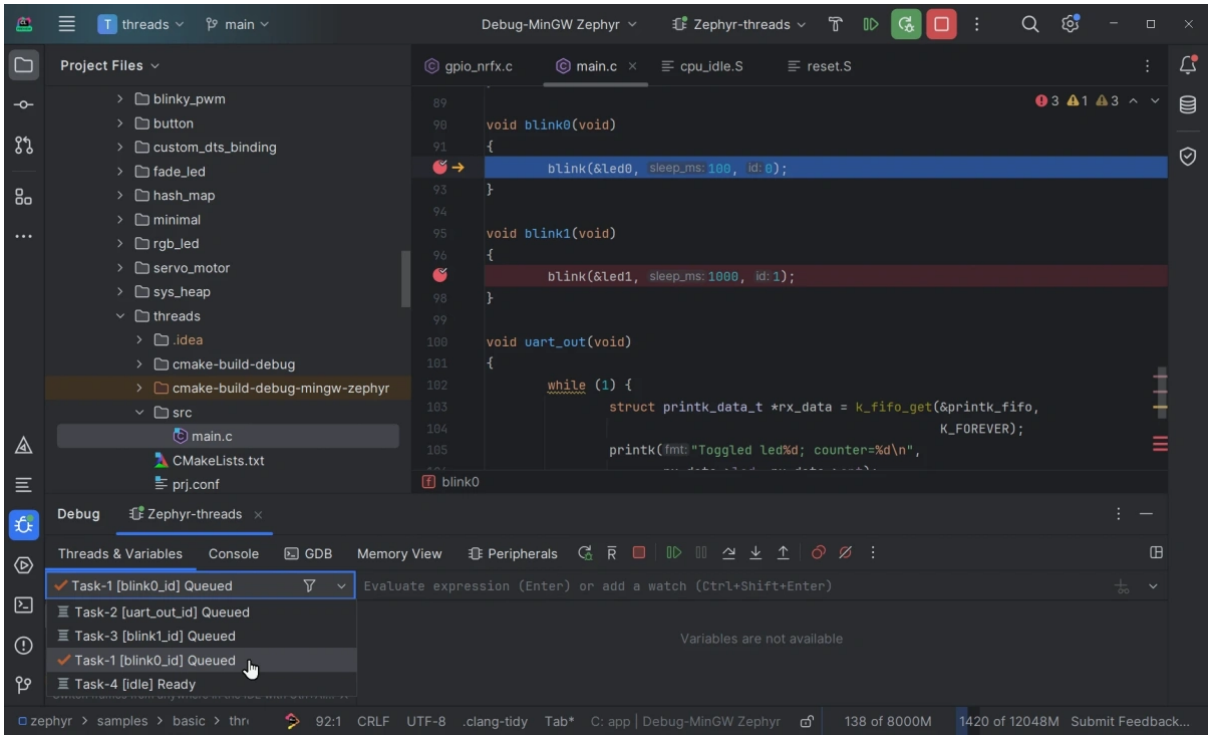

New Embedded GDB Server Run Configuration
Name:
Zephyr-threads
Store as project file
GDB Server type:
Segger JLink
Location:
C:\Program Files\SEGGER\JLink_V786f\JLinkGDBServerCL.exe
Expected name: JLinkGDBServerCL.exe
Debugger:
Bundled GDB multiarch
Target:
zephyr_final
Executable binary:
zephyr_final
Download binary:
Always
If updated
Never
TCP/IP port:
Auto
Next
Cancel


New Embedded GDB Server Run Configuration
Segger J-Link
Transport Interface:
SWD
JTAG
Initial Speed:
8000
kHz
Device:
nRF52832_xxAA
nRF52832_xxAB
nRF52833_xxAA
nRF52840_xxAA
nRF53
nRF91
Nuclei System
Previous
Create
Cancel

Start debugging

1. Place breakpoints by clicking in the left gutter next to the code lines.
2. Make sure that **Zephyr-threads** is selected in the configuration switcher and click the bug icon or press `Ctrl+D`.
3. When a breakpoint is hit, CLion opens the Debug tool window.

   Zephyr tasks are listed in the *Threads & Variables* pane. You can switch between them and inspect the variables for each task.



Refer to CLion web help for detailed description of the IDE debug capabilities.

2.15.2 Coccinelle

Coccinelle is a tool for pattern matching and text transformation that has many uses in kernel development, including the application of complex, tree-wide patches and detection of problematic programming patterns.

> **Note**
>
> Linux and macOS development environments are supported, but not Windows.

Getting Coccinelle

The semantic patches included in the kernel use features and options which are provided by Coccinelle version 1.0.0-rc11 and above. Using earlier versions will fail as the option names used by the Coccinelle files and `coccicheck` have been updated.

Coccinelle is available through the package manager of many distributions, e.g. :

- Debian
- Fedora
- Ubuntu
- OpenSUSE
- Arch Linux
- NetBSD
- FreeBSD

Some distribution packages are obsolete and it is recommended to use the latest version released from the Coccinelle homepage at http://coccinelle.lip6.fr/

Or from Github at:

https://github.com/coccinelle/coccinelle

Once you have it, run the following commands:

```
./autogen
./configure
make
```

as a regular user, and install it with:

```
sudo make install
```

More detailed installation instructions to build from source can be found at:

https://github.com/coccinelle/coccinelle/blob/master/install.txt

Supplemental documentation

For Semantic Patch Language(SmPL) grammar documentation refer to:

https://coccinelle.gitlabpages.inria.fr/website/documentation.html

Using Coccinelle on Zephyr

`coccicheck` checker is the front-end to the Coccinelle infrastructure and has various modes:

Four basic modes are defined: `patch`, `report`, `context`, and `org`. The mode to use is specified by setting `--mode=<mode>` or `-m=<mode>`.

- `patch` proposes a fix, when possible.
- `report` generates a list in the following format: file:line:column-column: message
- `context` highlights lines of interest and their context in a diff-like style.Lines of interest are indicated with `-`.
- `org` generates a report in the Org mode format of Emacs.

Note that not all semantic patches implement all modes. For easy use of Coccinelle, the default mode is `report`.

Two other modes provide some common combinations of these modes.

- `chain` tries the previous modes in the order above until one succeeds.
- `rep+ctxt` runs successively the report mode and the context mode. It should be used with the C option (described later) which checks the code on a file basis.

Examples

To make a report for every semantic patch, run the following command:

```
./scripts/coccicheck --mode=report
```

To produce patches, run:

```
./scripts/coccicheck --mode=patch
```

The `coccicheck` target applies every semantic patch available in the sub-directories of `scripts/coccinelle` to the entire source code tree.

For each semantic patch, a commit message is proposed. It gives a description of the problem being checked by the semantic patch, and includes a reference to Coccinelle.

As any static code analyzer, Coccinelle produces false positives. Thus, reports must be carefully checked, and patches reviewed.

To enable verbose messages set `--verbose=1` option, for example:

```
./scripts/coccicheck --mode=report --verbose=1
```

Coccinelle parallelization

By default, `coccicheck` tries to run as parallel as possible. To change the parallelism, set the `--jobs=<number>` option. For example, to run across 4 CPUs:

```
./scripts/coccicheck --mode=report --jobs=4
```

As of Coccinelle 1.0.2 Coccinelle uses Ocaml parmap for parallelization, if support for this is detected you will benefit from parmap parallelization.

When parmap is enabled `coccicheck` will enable dynamic load balancing by using `--chunksize 1` argument, this ensures we keep feeding threads with work one by one, so that we avoid the situation where most work gets done by only a few threads. With dynamic load balancing, if a thread finishes early we keep feeding it more work.

When parmap is enabled, if an error occurs in Coccinelle, this error value is propagated back, the return value of the `coccicheck` command captures this return value.

Using Coccinelle with a single semantic patch

The option `--cocci` can be used to check a single semantic patch. In that case, the variable must be initialized with the name of the semantic patch to apply.

For instance:

```
./scripts/coccicheck --mode=report --cocci=<example.cocci>
```

or:

```
./scripts/coccicheck --mode=report --cocci=./path/to/<example.cocci>
```

Controlling which files are processed by Coccinelle

By default the entire source tree is checked.

To apply Coccinelle to a specific directory, pass the path of specific directory as an argument.

For example, to check drivers/usb/ one may write:

```
./scripts/coccicheck --mode=patch drivers/usb/
```

The report mode is the default. You can select another one with the --mode=<mode> option explained above.

Debugging Coccinelle SmPL patches

Using coccicheck is best as it provides in the spatch command line include options matching the options used when we compile the kernel. You can learn what these options are by using verbose option, you could then manually run Coccinelle with debug options added.

Alternatively you can debug running Coccinelle against SmPL patches by asking for stderr to be redirected to stderr, by default stderr is redirected to /dev/null, if you'd like to capture stderr you can specify the --debug=file.err option to coccicheck. For instance:

```
rm -f cocci.err
./scripts/coccicheck --mode=patch --debug=cocci.err
cat cocci.err
```

Debugging support is only supported when using Coccinelle >= 1.0.2.

Additional Flags

Additional flags can be passed to spatch through the SPFLAGS variable. This works as Coccinelle respects the last flags given to it when options are in conflict.

```
./scripts/coccicheck --sp-flag="--use-glimpse"
```

Coccinelle supports idutils as well but requires coccinelle >= 1.0.6. When no ID file is specified coccinelle assumes your ID database file is in the file .id-utils.index on the top level of the kernel, coccinelle carries a script scripts/idutils_index.sh which creates the database with:

```
mkid -i C --output .id-utils.index
```

If you have another database filename you can also just symlink with this name.

```
./scripts/coccicheck --sp-flag="--use-idutils"
```

Alternatively you can specify the database filename explicitly, for instance:

```
./scripts/coccicheck --sp-flag="--use-idutils /full-path/to/ID"
```

Sometimes coccinelle doesn't recognize or parse complex macro variables due to insufficient definition. Therefore, to make it parsable we explicitly provide the prototype of the complex macro using the ---macro-file-builtins <headerfile.h> flag.

The <headerfile.h> should contain the complete prototype of the complex macro from which spatch engine can extract the type information required during transformation.

For example:

Z_SYSCALL_HANDLER is not recognized by coccinelle. Therefore, we put its prototype in a header file, say for example mymacros.h.

```
$ cat mymacros.h
#define Z_SYSCALL_HANDLER int xxx
```

Now we pass the header file mymacros.h during transformation:

```
./scripts/coccicheck --sp-flag="---macro-file-builtins mymacros.h"
```

See `spatch --help` to learn more about spatch options.

Note that the `--use-glimpse` and `--use-idutils` options require external tools for indexing the code. None of them is thus active by default. However, by indexing the code with one of these tools, and according to the cocci file used, spatch could proceed the entire code base more quickly.

SmPL patch specific options

SmPL patches can have their own requirements for options passed to Coccinelle. SmPL patch specific options can be provided by providing them at the top of the SmPL patch, for instance:

```
// Options: --no-includes --include-headers
```

Proposing new semantic patches

New semantic patches can be proposed and submitted by kernel developers. For sake of clarity, they should be organized in the sub-directories of `scripts/coccinelle/`.

The cocci script should have the following properties:

- The script **must** have `report` mode.
- The first few lines should state the purpose of the script using `///` comments . Usually, this message would be used as the commit log when proposing a patch based on the script.

Example

```
/// Use ARRAY_SIZE instead of dividing sizeof array with sizeof an element
```

- A more detailed information about the script with exceptional cases or false positives (if any) can be listed using `//#` comments.

Example

```
//# This makes an effort to find cases where ARRAY_SIZE can be used such as
//# where there is a division of sizeof the array by the sizeof its first
//# element or by any indexed element or the element type. It replaces the
//# division of the two sizeofs by ARRAY_SIZE.
```

- Confidence: It is a property defined to specify the accuracy level of the script. It can be either `High`, `Moderate` or `Low` depending upon the number of false positives observed.

Example

```
// Confidence: High
```

- Virtual rules: These are required to support the various modes framed in the script. The virtual rule specified in the script should have the corresponding mode handling rule.

Example

```
virtual context

@depends on context@
type T;
T[] E;
@@
(
* (sizeof(E)/sizeof(*E))
|
* (sizeof(E)/sizeof(E[...]))
|
* (sizeof(E)/sizeof(T))
)
```

Detailed description of the `report` mode

`report` generates a list in the following format:

```
file:line:column-column: message
```

Example Running:

```
./scripts/coccicheck --mode=report --cocci=scripts/coccinelle/array_size.cocci
```

will execute the following part of the SmPL script:

```
<smpl>

@r depends on (org || report)@
type T;
T[] E;
position p;
@@
(
(sizeof(E)@p /sizeof(*E))
|
(sizeof(E)@p /sizeof(E[...]))
|
(sizeof(E)@p /sizeof(T))
)

@script:python depends on report@
p << r.p;
@@

msg="WARNING: Use ARRAY_SIZE"
coccilib.report.print_report(p[0], msg)

</smpl>
```

This SmPL excerpt generates entries on the standard output, as illustrated below:

```
ext/hal/nxp/mcux/drivers/lpc/fsl_wwdt.c:66:49-50: WARNING: Use ARRAY_SIZE
ext/hal/nxp/mcux/drivers/lpc/fsl_ctimer.c:74:53-54: WARNING: Use ARRAY_SIZE
ext/hal/nxp/mcux/drivers/imx/fsl_dcp.c:944:45-46: WARNING: Use ARRAY_SIZE
```

Detailed description of the `patch` mode

When the `patch` mode is available, it proposes a fix for each problem identified.

Example Running:

```
./scripts/coccicheck --mode=patch --cocci=scripts/coccinelle/misc/array_size.cocci
```

will execute the following part of the SmPL script:

```
<smpl>

@depends on patch@
type T;
T[] E;
@@
(
- (sizeof(E)/sizeof(*E))
+ ARRAY_SIZE(E)
|
- (sizeof(E)/sizeof(E[...]))
+ ARRAY_SIZE(E)
|
- (sizeof(E)/sizeof(T))
+ ARRAY_SIZE(E)
)

</smpl>
```

This SmPL excerpt generates patch hunks on the standard output, as illustrated below:

```
diff -u -p a/ext/lib/encoding/tinycbor/src/cborvalidation.c b/ext/lib/encoding/tinycbor/src/
↪cborvalidation.c
--- a/ext/lib/encoding/tinycbor/src/cborvalidation.c
+++ b/ext/lib/encoding/tinycbor/src/cborvalidation.c
@@ -325,7 +325,7 @@ static inline CborError validate_number(
static inline CborError validate_tag(CborValue *it, CborTag tag, int flags, int␣
↪recursionLeft)
{
  CborType type = cbor_value_get_type(it);
-    const size_t knownTagCount = sizeof(knownTagData) / sizeof(knownTagData[0]);
+    const size_t knownTagCount = ARRAY_SIZE(knownTagData);
   const struct KnownTagData *tagData = knownTagData;
   const struct KnownTagData * const knownTagDataEnd = knownTagData + knownTagCount;
```

Detailed description of the `context` mode

`context` highlights lines of interest and their context in a diff-like style.

> **Note**
>
> The diff-like output generated is NOT an applicable patch. The intent of the `context` mode is to highlight the important lines (annotated with minus, `-`) and gives some surrounding context lines around. This output can be used with the diff mode of Emacs to review the code.

Example Running:

```
./scripts/coccicheck --mode=context --cocci=scripts/coccinelle/array_size.cocci
```

will execute the following part of the SmPL script:

```
<smpl>

@depends on context@
type T;
T[] E;
@@
(
* (sizeof(E)/sizeof(*E))
|
* (sizeof(E)/sizeof(E[...]))
|
* (sizeof(E)/sizeof(T))
)

</smpl>
```

This SmPL excerpt generates diff hunks on the standard output, as illustrated below:

```
diff -u -p ext/lib/encoding/tinycbor/src/cborvalidation.c /tmp/nothing/ext/lib/encoding/
↪tinycbor/src/cborvalidation.c
--- ext/lib/encoding/tinycbor/src/cborvalidation.c
+++ /tmp/nothing/ext/lib/encoding/tinycbor/src/cborvalidation.c
@@ -325,7 +325,6 @@ static inline CborError validate_number(
static inline CborError validate_tag(CborValue *it, CborTag tag, int flags, int␣
↪recursionLeft)
{
  CborType type = cbor_value_get_type(it);
-    const size_t knownTagCount = sizeof(knownTagData) / sizeof(knownTagData[0]);
   const struct KnownTagData *tagData = knownTagData;
   const struct KnownTagData * const knownTagDataEnd = knownTagData + knownTagCount;
```

Detailed description of the `org` mode

`org` generates a report in the Org mode format of Emacs.

Example Running:

```
./scripts/coccicheck --mode=org --cocci=scripts/coccinelle/misc/array_size.cocci
```

will execute the following part of the SmPL script:

```
<smpl>

@r depends on (org || report)@
type T;
T[] E;
position p;
@@
(
(sizeof(E)@p /sizeof(*E))
|
(sizeof(E)@p /sizeof(E[...]))
|
(sizeof(E)@p /sizeof(T))
)
```

(continues on next page)

(continued from previous page)

```
@script:python depends on org@
p << r.p;
@@
coccilib.org.print_todo(p[0], "WARNING should use ARRAY_SIZE")

</smpl>
```

This SmPL excerpt generates Org entries on the standard output, as illustrated below:

```
* TODO [[view:ext/lib/encoding/tinycbor/src/cborvalidation.c::face=ovl-
↪face1::linb=328::colb=52::cole=53][WARNING should use ARRAY_SIZE]]
```

Coccinelle Mailing List

Subscribe to the coccinelle mailing list:

- https://systeme.lip6.fr/mailman/listinfo/cocci

Archives:

- https://lore.kernel.org/cocci/
- https://systeme.lip6.fr/pipermail/cocci/

2.15.3 Visual Studio Code

Visual Studio Code (VS Code for short) is a popular cross-platform IDE that supports C projects and has a rich set of extensions.

This guide describes the process of setting up VS Code for Zephyr's blinky sample in VS Code.

The instructions have been tested on Linux, but the steps should be the same for macOS and Windows, just make sure to adjust the paths if needed.

Get VS Code

Download VS Code and install it.

Install the required extensions through the *Extensions* marketplace in the left panel. Search for the C/C++ Extension Pack and install it.

Initialize a new workspace

This guide gives details on how to configure the blinky sample application, but the instructions would be similar for any Zephyr project and *workspace layout*.

Before you start, make sure you have a working Zephyr development environment, as per the instructions in the *Getting Started Guide*.

Open the project in VS Code

1. In VS Code, select *File* ▸ *Open Folder* from the main menu.
2. Navigate to your Zephyr workspace and select it (i.e. the `zephyrproject` folder in your HOME directory if you have followed the Getting Started instructions).

3. If prompted, enable workspace trust.

Generate compile commands

In order to support code navigation and linting capabilities, you must compile your project once to generate the `compile_commands.json` file that will provide the C/C++ extension with the required information (ex. include paths):

```
west build -b native_sim/native/64 samples/basic/blinky
```

Configure the C/C++ extension

You'll now need to point to the generated `compile_commands.json` file to enable linting and code navigation in VS Code.

1. Go to the *File* ▸ *Preferences* ▸ *Settings* in the VS Code top menu.
2. Search for the parameter *C_Cpp* > *Default: Compile Commands* and set its value to: `zephyr/build/compile_commands.json`.

   Linting errors in the code should now be resolved, and you should be able to navigate through the code.

Additional resources

There are many other extensions that can be useful when working with Zephyr and VS Code. While this guide does not cover them yet, you may refer to their documentation to set them up:

Contribution tooling

- Checkpatch Extension
- EditorConfig Extension

Documentation languages extensions

- reStructuredText Extension Pack

IDE extensions

- CMake Extension documentation
- nRF Kconfig Extension
- nRF DeviceTree Extension
- GNU Linker Map files Extension

Additional guides

- How to Develop Zephyr Apps with a Modern, Visual IDE

Note

Please be aware that these extensions might not all have the same level of quality and maintenance.

Chapter 3

Kernel

3.1 Kernel Services

The Zephyr kernel lies at the heart of every Zephyr application. It provides a low footprint, high performance, multi-threaded execution environment with a rich set of available features. The rest of the Zephyr ecosystem, including device drivers, networking stack, and application-specific code, uses the kernel's features to create a complete application.

The configurable nature of the kernel allows you to incorporate only those features needed by your application, making it ideal for systems with limited amounts of memory (as little as 2 KB!) or with simple multi-threading requirements (such as a set of interrupt handlers and a single background task). Examples of such systems include: embedded sensor hubs, environmental sensors, simple LED wearable, and store inventory tags.

Applications requiring more memory (50 to 900 KB), multiple communication devices (like Wi-Fi and Bluetooth Low Energy), and complex multi-threading, can also be developed using the Zephyr kernel. Examples of such systems include: fitness wearables, smart watches, and IoT wireless gateways.

3.1.1 Scheduling, Interrupts, and Synchronization

These pages cover basic kernel services related to thread scheduling and synchronization.

Threads

Note

There is also limited support for using *Operation without Threads*.

- *Lifecycle*
  - *Thread Creation*
  - *Thread Termination*
  - *Thread Aborting*
  - *Thread Suspension*

- *Thread States*
- *Thread Stack objects*
  - *Kernel-only Stacks*
  - *Thread stacks*
- *Thread Priorities*
  - *Meta-IRQ Priorities*
- *Thread Options*
- *Thread Custom Data*
- *Implementation*
  - *Spawning a Thread*
  - *Dropping Permissions*
  - *Terminating a Thread*
- *Runtime Statistics*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

This section describes kernel services for creating, scheduling, and deleting independently executable threads of instructions.

A *thread* is a kernel object that is used for application processing that is too lengthy or too complex to be performed by an ISR.

Any number of threads can be defined by an application (limited only by available RAM). Each thread is referenced by a *thread id* that is assigned when the thread is spawned.

A thread has the following key properties:

- A **stack area**, which is a region of memory used for the thread's stack. The **size** of the stack area can be tailored to conform to the actual needs of the thread's processing. Special macros exist to create and work with stack memory regions.
- A **thread control block** for private kernel bookkeeping of the thread's metadata. This is an instance of type `k_thread`.
- An **entry point function**, which is invoked when the thread is started. Up to 3 **argument values** can be passed to this function.
- A **scheduling priority**, which instructs the kernel's scheduler how to allocate CPU time to the thread. (See *Scheduling*.)
- A set of **thread options**, which allow the thread to receive special treatment by the kernel under specific circumstances. (See *Thread Options*.)
- A **start delay**, which specifies how long the kernel should wait before starting the thread.
- An **execution mode**, which can either be supervisor or user mode. By default, threads run in supervisor mode and allow access to privileged CPU instructions, the entire memory address space, and peripherals. User mode threads have a reduced set of privileges. This depends on the `CONFIG_USERSPACE` option. See *User Mode*.

Lifecycle

Thread Creation A thread must be created before it can be used. The kernel initializes the thread control block as well as one end of the stack portion. The remainder of the thread's stack is typically left uninitialized.

Specifying a start delay of *K_NO_WAIT* instructs the kernel to start thread execution immediately. Alternatively, the kernel can be instructed to delay execution of the thread by specifying a timeout value – for example, to allow device hardware used by the thread to become available.

The kernel allows a delayed start to be canceled before the thread begins executing. A cancellation request has no effect if the thread has already started. A thread whose delayed start was successfully canceled must be re-spawned before it can be used.

Thread Termination Once a thread is started it typically executes forever. However, a thread may synchronously end its execution by returning from its entry point function. This is known as **termination**.

A thread that terminates is responsible for releasing any shared resources it may own (such as mutexes and dynamically allocated memory) prior to returning, since the kernel does *not* reclaim them automatically.

In some cases a thread may want to sleep until another thread terminates. This can be accomplished with the *k_thread_join()* API. This will block the calling thread until either the timeout expires, the target thread self-exits, or the target thread aborts (either due to a *k_thread_abort()* call or triggering a fatal error).

Once a thread has terminated, the kernel guarantees that no use will be made of the thread struct. The memory of such a struct can then be re-used for any purpose, including spawning a new thread. Note that the thread must be fully terminated, which presents race conditions where a thread's own logic signals completion which is seen by another thread before the kernel processing is complete. Under normal circumstances, application code should use *k_thread_join()* or *k_thread_abort()* to synchronize on thread termination state and not rely on signaling from within application logic.

Thread Aborting A thread may asynchronously end its execution by **aborting**. The kernel automatically aborts a thread if the thread triggers a fatal error condition, such as dereferencing a null pointer.

A thread can also be aborted by another thread (or by itself) by calling *k_thread_abort()*. However, it is typically preferable to signal a thread to terminate itself gracefully, rather than aborting it.

As with thread termination, the kernel does not reclaim shared resources owned by an aborted thread.

Note

The kernel does not currently make any claims regarding an application's ability to respawn a thread that aborts.

Thread Suspension A thread can be prevented from executing for an indefinite period of time if it becomes **suspended**. The function *k_thread_suspend()* can be used to suspend any thread, including the calling thread. Suspending a thread that is already suspended has no additional effect.

Once suspended, a thread cannot be scheduled until another thread calls *k_thread_resume()* to remove the suspension.

Note

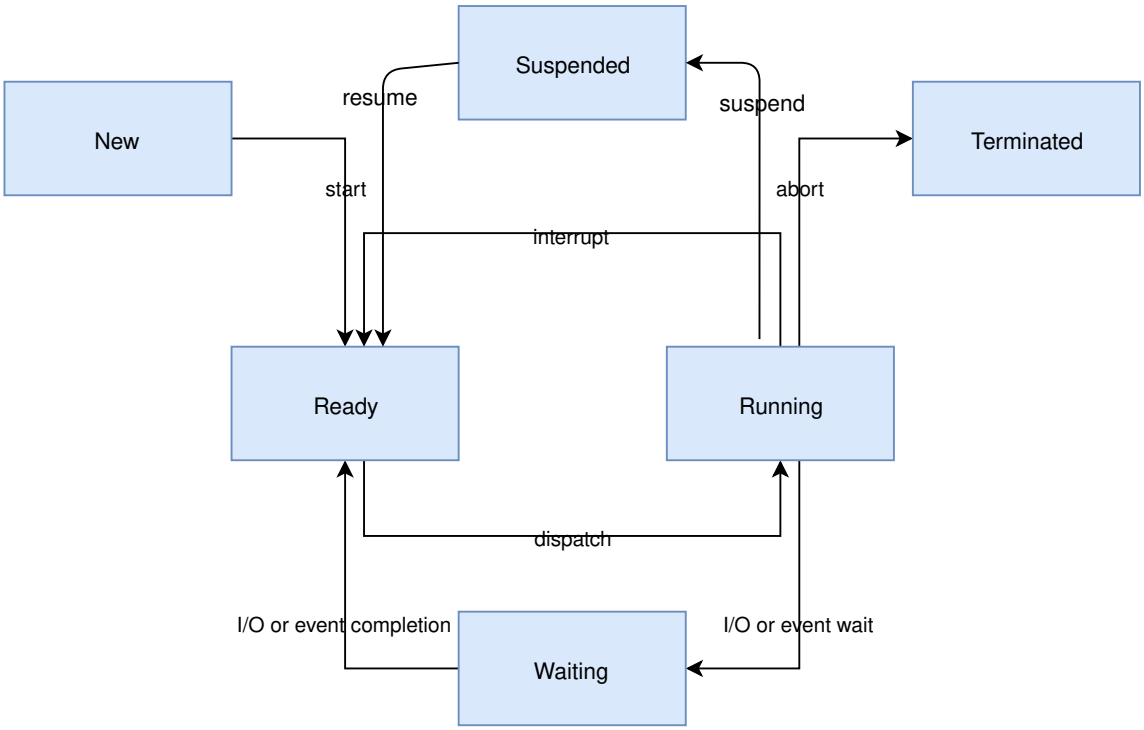
A thread can prevent itself from executing for a specified period of time using *k_sleep()*. However, this is different from suspending a thread since a sleeping thread becomes executable automatically when the time limit is reached.

Thread States A thread that has no factors that prevent its execution is deemed to be **ready**, and is eligible to be selected as the current thread.

A thread that has one or more factors that prevent its execution is deemed to be **unready**, and cannot be selected as the current thread.

The following factors make a thread unready:

- The thread has not been started.
- The thread is waiting for a kernel object to complete an operation. (For example, the thread is taking a semaphore that is unavailable.)
- The thread is waiting for a timeout to occur.
- The thread has been suspended.
- The thread has terminated or aborted.



Note

Although the diagram above may appear to suggest that both **Ready** and **Running** are distinct thread states, that is not the correct interpretation. **Ready** is a thread state, and **Running** is a schedule state that only applies to **Ready** threads.

Thread Stack objects Every thread requires its own stack buffer for the CPU to push context. Depending on configuration, there are several constraints that must be met:

- There may need to be additional memory reserved for memory management structures

- If guard-based stack overflow detection is enabled, a small write- protected memory management region must immediately precede the stack buffer to catch overflows.
- If userspace is enabled, a separate fixed-size privilege elevation stack must be reserved to serve as a private kernel stack for handling system calls.
- If userspace is enabled, the thread's stack buffer must be appropriately sized and aligned such that a memory protection region may be programmed to exactly fit.

The alignment constraints can be quite restrictive, for example some MPUs require their regions to be of some power of two in size, and aligned to its own size.

Because of this, portable code can't simply pass an arbitrary character buffer to *k_thread_create()*. Special macros exist to instantiate stacks, prefixed with K_KERNEL_STACK and K_THREAD_STACK.

Kernel-only Stacks If it is known that a thread will never run in user mode, or the stack is being used for special contexts like handling interrupts, it is best to define stacks using the K_KERNEL_STACK macros.

These stacks save memory because an MPU region will never need to be programmed to cover the stack buffer itself, and the kernel will not need to reserve additional room for the privilege elevation stack, or memory management data structures which only pertain to user mode threads.

Attempts from user mode to use stacks declared in this way will result in a fatal error for the caller.

If CONFIG_USERSPACE is not enabled, the set of K_THREAD_STACK macros have an identical effect to the K_KERNEL_STACK macros.

Thread stacks If it is known that a stack will need to host user threads, or if this cannot be determined, define the stack with K_THREAD_STACK macros. This may use more memory but the stack object is suitable for hosting user threads.

If CONFIG_USERSPACE is not enabled, the set of K_THREAD_STACK macros have an identical effect to the K_KERNEL_STACK macros.

Thread Priorities A thread's priority is an integer value, and can be either negative or non-negative. Numerically lower priorities takes precedence over numerically higher values. For example, the scheduler gives thread A of priority 4 *higher* priority over thread B of priority 7; likewise thread C of priority -2 has higher priority than both thread A and thread B.

The scheduler distinguishes between two classes of threads, based on each thread's priority.

- A *cooperative thread* has a negative priority value. Once it becomes the current thread, a cooperative thread remains the current thread until it performs an action that makes it unready.
- A *preemptible thread* has a non-negative priority value. Once it becomes the current thread, a preemptible thread may be supplanted at any time if a cooperative thread, or a preemptible thread of higher or equal priority, becomes ready.

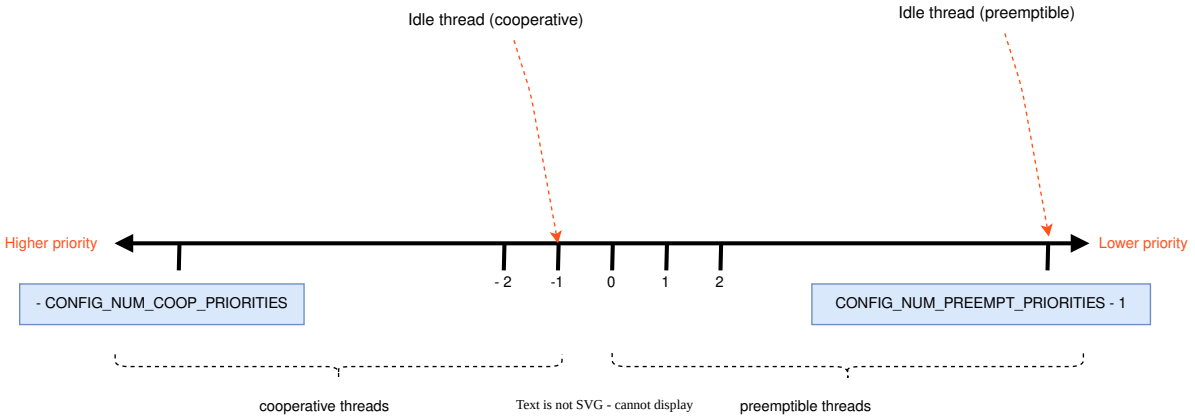
A thread's initial priority value can be altered up or down after the thread has been started. Thus it is possible for a preemptible thread to become a cooperative thread, and vice versa, by changing its priority.

Note

The scheduler does not make heuristic decisions to re-prioritize threads. Thread priorities are set and changed only at the application's request.

The kernel supports a virtually unlimited number of thread priority levels. The configuration options CONFIG_NUM_COOP_PRIORITIES and CONFIG_NUM_PREEMPT_PRIORITIES specify the number of priority levels for each class of thread, resulting in the following usable priority ranges:

- cooperative threads: (-CONFIG_NUM_COOP_PRIORITIES) to -1
- preemptive threads: 0 to (CONFIG_NUM_PREEMPT_PRIORITIES - 1)



For example, configuring 5 cooperative priorities and 10 preemptive priorities results in the ranges -5 to -1 and 0 to 9, respectively.

Meta-IRQ Priorities When enabled (see CONFIG_NUM_METAIRQ_PRIORITIES), there is a special subclass of cooperative priorities at the highest (numerically lowest) end of the priority space: meta-IRQ threads. These are scheduled according to their normal priority, but also have the special ability to preempt all other threads (and other meta-IRQ threads) at lower priorities, even if those threads are cooperative and/or have taken a scheduler lock. Meta-IRQ threads are still threads, however, and can still be interrupted by any hardware interrupt.

This behavior makes the act of unblocking a meta-IRQ thread (by any means, e.g. creating it, calling k_sem_give(), etc.) into the equivalent of a synchronous system call when done by a lower priority thread, or an ARM-like "pended IRQ" when done from true interrupt context. The intent is that this feature will be used to implement interrupt "bottom half" processing and/or "tasklet" features in driver subsystems. The thread, once woken, will be guaranteed to run before the current CPU returns into application code.

Unlike similar features in other OSes, meta-IRQ threads are true threads and run on their own stack (which must be allocated normally), not the per-CPU interrupt stack. Design work to enable the use of the IRQ stack on supported architectures is pending.

Note that because this breaks the promise made to cooperative threads by the Zephyr API (namely that the OS won't schedule other thread until the current thread deliberately blocks), it should be used only with great care from application code. These are not simply very high priority threads and should not be used as such.

Thread Options The kernel supports a small set of *thread options* that allow a thread to receive special treatment under specific circumstances. The set of options associated with a thread are specified when the thread is spawned.

A thread that does not require any thread option has an option value of zero. A thread that requires a thread option specifies it by name, using the | character as a separator if multiple options are needed (i.e. combine options using the bitwise OR operator).

The following thread options are supported.

K_ESSENTIAL
: This option tags the thread as an *essential thread*. This instructs the kernel to treat the termination or aborting of the thread as a fatal system error.

By default, the thread is not considered to be an essential thread.

K_SSE_REGS
: This x86-specific option indicate that the thread uses the CPU's SSE registers. Also see *K_FP_REGS*.

  By default, the kernel does not attempt to save and restore the contents of these registers when scheduling the thread.

K_FP_REGS
: This option indicate that the thread uses the CPU's floating point registers. This instructs the kernel to take additional steps to save and restore the contents of these registers when scheduling the thread. (For more information see *Floating Point Services*.)

  By default, the kernel does not attempt to save and restore the contents of this register when scheduling the thread.

K_USER
: If CONFIG_USERSPACE is enabled, this thread will be created in user mode and will have reduced privileges. See *User Mode*. Otherwise this flag does nothing.

K_INHERIT_PERMS
: If CONFIG_USERSPACE is enabled, this thread will inherit all kernel object permissions that the parent thread had, except the parent thread object. See *User Mode*.

Thread Custom Data Every thread has a 32-bit *custom data* area, accessible only by the thread itself, and may be used by the application for any purpose it chooses. The default custom data value for a thread is zero.

> **Note**
>
> Custom data support is not available to ISRs because they operate within a single shared kernel interrupt handling context.

By default, thread custom data support is disabled. The configuration option CONFIG_THREAD_CUSTOM_DATA can be used to enable support.

The *k_thread_custom_data_set()* and *k_thread_custom_data_get()* functions are used to write and read a thread's custom data, respectively. A thread can only access its own custom data, and not that of another thread.

The following code uses the custom data feature to record the number of times each thread calls a specific routine.

> **Note**
>
> Obviously, only a single routine can use this technique, since it monopolizes the use of the custom data feature.

```
int call_tracking_routine(void)
{
    uint32_t call_count;

    if (k_is_in_isr()) {
        /* ignore any call made by an ISR */
    } else {
        call_count = (uint32_t)k_thread_custom_data_get();
        call_count++;
        k_thread_custom_data_set((void *)call_count);
```

(continues on next page)

(continued from previous page)

```
    }

    /* do rest of routine's processing */
    ...
}
```

Use thread custom data to allow a routine to access thread-specific information, by using the custom data as a pointer to a data structure owned by the thread.

Implementation

Spawning a Thread A thread is spawned by defining its stack area and its thread control block, and then calling *k_thread_create()*.

The stack area must be defined using *K_THREAD_STACK_DEFINE* or *K_KERNEL_STACK_DEFINE* to ensure it is properly set up in memory.

The size parameter for the stack must be one of three values:

- The original requested stack size passed to K_THREAD_STACK or K_KERNEL_STACK family of stack instantiation macros.
- For a stack object defined with the K_THREAD_STACK family of macros, the return value of *K_THREAD_STACK_SIZEOF()* for that' object.
- For a stack object defined with the K_KERNEL_STACK family of macros, the return value of *K_KERNEL_STACK_SIZEOF()* for that object.

The thread spawning function returns its thread id, which can be used to reference the thread.

The following code spawns a thread that starts immediately.

```
#define MY_STACK_SIZE 500
#define MY_PRIORITY 5

extern void my_entry_point(void *, void *, void *);

K_THREAD_STACK_DEFINE(my_stack_area, MY_STACK_SIZE);
struct k_thread my_thread_data;

k_tid_t my_tid = k_thread_create(&my_thread_data, my_stack_area,
                                 K_THREAD_STACK_SIZEOF(my_stack_area),
                                 my_entry_point,
                                 NULL, NULL, NULL,
                                 MY_PRIORITY, 0, K_NO_WAIT);
```

Alternatively, a thread can be declared at compile time by calling *K_THREAD_DEFINE*. Observe that the macro defines the stack area, control block, and thread id variables automatically.

The following code has the same effect as the code segment above.

```
#define MY_STACK_SIZE 500
#define MY_PRIORITY 5

extern void my_entry_point(void *, void *, void *);

K_THREAD_DEFINE(my_tid, MY_STACK_SIZE,
                my_entry_point, NULL, NULL, NULL,
                MY_PRIORITY, 0, 0);
```

Note

The delay parameter to *k_thread_create()* is a *k_timeout_t* value, so *K_NO_WAIT* means to start the thread immediately. The corresponding parameter to *K_THREAD_DEFINE* is a duration in integral milliseconds, so the equivalent argument is 0.

User Mode Constraints This section only applies if CONFIG_USERSPACE is enabled, and a user thread tries to create a new thread. The *k_thread_create()* API is still used, but there are additional constraints which must be met or the calling thread will be terminated:

- The calling thread must have permissions granted on both the child thread and stack parameters; both are tracked by the kernel as kernel objects.
- The child thread and stack objects must be in an uninitialized state, i.e. it is not currently running and the stack memory is unused.
- The stack size parameter passed in must be equal to or less than the bounds of the stack object when it was declared.
- The *K_USER* option must be used, as user threads can only create other user threads.
- The *K_ESSENTIAL* option must not be used, user threads may not be considered essential threads.
- The priority of the child thread must be a valid priority value, and equal to or lower than the parent thread.

Dropping Permissions If CONFIG_USERSPACE is enabled, a thread running in supervisor mode may perform a one-way transition to user mode using the *k_thread_user_mode_enter()* API. This is a one-way operation which will reset and zero the thread's stack memory. The thread will be marked as non-essential.

Terminating a Thread A thread terminates itself by returning from its entry point function.

The following code illustrates the ways a thread can terminate.

```
void my_entry_point(int unused1, int unused2, int unused3)
{
    while (1) {
        ...
        if (<some condition>) {
            return; /* thread terminates from mid-entry point function */
        }
        ...
    }

    /* thread terminates at end of entry point function */
}
```

If CONFIG_USERSPACE is enabled, aborting a thread will additionally mark the thread and stack objects as uninitialized so that they may be re-used.

Runtime Statistics Thread runtime statistics can be gathered and retrieved if CONFIG_THREAD_RUNTIME_STATS is enabled, for example, total number of execution cycles of a thread.

By default, the runtime statistics are gathered using the default kernel timer. For some architectures, SoCs or boards, there are timers with higher resolution available via timing functions. Using of these timers can be enabled via CONFIG_THREAD_RUNTIME_STATS_USE_TIMING_FUNCTIONS.

Here is an example:

```
k_thread_runtime_stats_t rt_stats_thread;

k_thread_runtime_stats_get(k_current_get(), &rt_stats_thread);

printk("Cycles: %llu\n", rt_stats_thread.execution_cycles);
```

Suggested Uses Use threads to handle processing that cannot be handled in an ISR.

Use separate threads to handle logically distinct processing operations that can execute in parallel.

Configuration Options Related configuration options:

- CONFIG_MAIN_THREAD_PRIORITY
- CONFIG_MAIN_STACK_SIZE
- CONFIG_IDLE_STACK_SIZE
- CONFIG_THREAD_CUSTOM_DATA
- CONFIG_NUM_COOP_PRIORITIES
- CONFIG_NUM_PREEMPT_PRIORITIES
- CONFIG_TIMESLICING
- CONFIG_TIMESLICE_SIZE
- CONFIG_TIMESLICE_PRIORITY
- CONFIG_USERSPACE

Related code samples

Basic Synchronization
Manipulate basic kernel synchronization primitives.

Basic thread manipulation
Spawn multiple threads that blink LEDs and print information to the console.

Dumb HTTP server (multi-threaded)
Implement a simple HTTP server supporting simultaneous connections using BSD sockets.

API Reference

group thread_apis

Defines

K_ESSENTIAL
system thread that must not abort

K_FP_IDX

FPU registers are managed by context switch.

This option indicates that the thread uses the CPU's floating point registers. This instructs the kernel to take additional steps to save and restore the contents of these registers when scheduling the thread. No effect if CONFIG_FPU_SHARING is not enabled.

K_FP_REGS

K_USER

user mode thread

This thread has dropped from supervisor mode to user mode and consequently has additional restrictions

K_INHERIT_PERMS

Inherit Permissions.

Indicates that the thread being created should inherit all kernel object permissions from the thread that created it. No effect if CONFIG_USERSPACE is not enabled.

K_CALLBACK_STATE

Callback item state.

This is a single bit of state reserved for "callback manager" utilities (p4wq initially) who need to track operations invoked from within a user-provided callback they have been invoked. Effectively it serves as a tiny bit of zero-overhead TLS data.

K_DSP_IDX

DSP registers are managed by context switch.

This option indicates that the thread uses the CPU's DSP registers. This instructs the kernel to take additional steps to save and restore the contents of these registers when scheduling the thread. No effect if CONFIG_DSP_SHARING is not enabled.

K_DSP_REGS

K_AGU_IDX

AGU registers are managed by context switch.

This option indicates that the thread uses the ARC processor's XY memory and DSP feature. Often used with CONFIG_ARC_AGU_SHARING . No effect if CONFIG_ARC_AGU_SHARING is not enabled.

K_AGU_REGS

K_SSE_REGS

FP and SSE registers are managed by context switch on x86.

This option indicates that the thread uses the x86 CPU's floating point and SSE registers. This instructs the kernel to take additional steps to save and restore the contents of these registers when scheduling the thread. No effect if CONFIG_X86_SSE is not enabled.

`k_thread_access_grant`(thread, ...)

Grant a thread access to a set of kernel objects.

This is a convenience function. For the provided thread, grant access to the remaining arguments, which must be pointers to kernel objects.

The thread object must be initialized (i.e. running). The objects don't need to be. Note that NULL shouldn't be passed as an argument.

Parameters

- `thread` – Thread to grant access to objects
- `...` – list of kernel object pointers

`K_THREAD_DEFINE`(name, stack_size, entry, p1, p2, p3, prio, options, delay)

Statically define and initialize a thread.

The thread may be scheduled for immediate execution or a delayed start.

Thread options are architecture-specific, and can include K_ESSENTIAL, K_FP_REGS, and K_SSE_REGS. Multiple options may be specified by separating them using "|" (the logical OR operator).

The ID of the thread can be accessed using:

```
extern const k_tid_t <name>;
```

Note

Static threads with zero delay should not normally have MetaIRQ priority levels. This can preempt the system initialization handling (depending on the priority of the main thread) and cause surprising ordering side effects. It will not affect anything in the OS per se, but consider it bad practice. Use a SYS_INIT() callback if you need to run code before entrance to the application main().

Parameters

- `name` – Name of the thread.
- `stack_size` – Stack size in bytes.
- `entry` – Thread entry function.
- `p1` – 1st entry point parameter.
- `p2` – 2nd entry point parameter.
- `p3` – 3rd entry point parameter.
- `prio` – Thread priority.
- `options` – Thread options.
- `delay` – Scheduling delay (in milliseconds), zero for no delay.

`K_KERNEL_THREAD_DEFINE`(name, stack_size, entry, p1, p2, p3, prio, options, delay)

Statically define and initialize a thread intended to run only in kernel mode.

The thread may be scheduled for immediate execution or a delayed start.

Thread options are architecture-specific, and can include K_ESSENTIAL, K_FP_REGS, and K_SSE_REGS. Multiple options may be specified by separating them using "|" (the logical OR operator).

The ID of the thread can be accessed using:

```
extern const k_tid_t <name>;
```

Note

Threads defined by this can only run in kernel mode, and cannot be transformed into user thread via *k_thread_user_mode_enter()*.

Warning

Depending on the architecture, the stack size (`stack_size`) may need to be multiples of CONFIG_MMU_PAGE_SIZE (if MMU) or in power-of-two size (if MPU).

Parameters

- `name` – Name of the thread.
- `stack_size` – Stack size in bytes.
- `entry` – Thread entry function.
- `p1` – 1st entry point parameter.
- `p2` – 2nd entry point parameter.
- `p3` – 3rd entry point parameter.
- `prio` – Thread priority.
- `options` – Thread options.
- `delay` – Scheduling delay (in milliseconds), zero for no delay.

Typedefs

typedef void (***k_thread_user_cb_t**)(const struct *k_thread* *thread, void *user_data)

Functions

void **k_thread_foreach**(*k_thread_user_cb_t* user_cb, void *user_data)

Iterate over all the threads in the system.

This routine iterates over all the threads in the system and calls the user_cb function for each thread.

Note

`CONFIG_THREAD_MONITOR` must be set for this function to be effective.

Note

This API uses *k_spin_lock* to protect the _kernel.threads list which means creation of new threads and terminations of existing threads are blocked until this API returns.

Parameters

- **user_cb** – Pointer to the user callback function.
- **user_data** – Pointer to user data.

void **k_thread_foreach_unlocked**(*k_thread_user_cb_t* user_cb, void *user_data)

Iterate over all the threads in the system without locking.

This routine works exactly the same like *k_thread_foreach* but unlocks interrupts when user_cb is executed.

Note

CONFIG_THREAD_MONITOR must be set for this function to be effective.

Note

This API uses *k_spin_lock* only when accessing the _kernel.threads queue elements. It unlocks it during user callback function processing. If a new task is created when this `foreach` function is in progress, the added new task would not be included in the enumeration. If a task is aborted during this enumeration, there would be a race here and there is a possibility that this aborted task would be included in the enumeration.

Note

If the task is aborted and the memory occupied by its *k_thread* structure is reused when this `k_thread_foreach_unlocked` is in progress it might even lead to the system behave unstable. This function may never return, as it would follow some `next` task pointers treating given pointer as a pointer to the *k_thread* structure while it is something different right now. Do not reuse the memory that was occupied by *k_thread* structure of aborted task if it was aborted after this function was called in any context.

Parameters

- **user_cb** – Pointer to the user callback function.
- **user_data** – Pointer to user data.

k_thread_stack_t ***k_thread_stack_alloc**(size_t size, int flags)

Dynamically allocate a thread stack.

Relevant stack creation flags include:

- *K_USER* allocate a userspace thread (requires CONFIG_USERSPACE=y)

See also

CONFIG_DYNAMIC_THREAD

Parameters

- **size** – Stack size in bytes.

- **flags** – Stack creation flags, or 0.

Return values

- **the** – allocated thread stack on success.
- **NULL** – on failure.

int **k_thread_stack_free**(k_thread_stack_t *stack)

Free a dynamically allocated thread stack.

See also

CONFIG_DYNAMIC_THREAD

Parameters

- **stack** – Pointer to the thread stack.

Return values

- **0** – on success.
- **-EBUSY** – if the thread stack is in use.
- **-EINVAL** – if `stack` is invalid.
- **-ENOSYS** – if dynamic thread stack allocation is disabled

k_tid_t **k_thread_create**(struct *k_thread* *new_thread, k_thread_stack_t *stack, size_t stack_size, k_thread_entry_t entry, void *p1, void *p2, void *p3, int prio, uint32_t options, *k_timeout_t* delay)

Create a thread.

This routine initializes a thread, then schedules it for execution.

The new thread may be scheduled for immediate execution or a delayed start. If the newly spawned thread does not have a delayed start the kernel scheduler may preempt the current thread to allow the new thread to execute.

Thread options are architecture-specific, and can include K_ESSENTIAL, K_FP_REGS, and K_SSE_REGS. Multiple options may be specified by separating them using "|" (the logical OR operator).

Stack objects passed to this function must be originally defined with either of these macros in order to be portable:

- *K_THREAD_STACK_DEFINE()* - For stacks that may support either user or supervisor threads.
- *K_KERNEL_STACK_DEFINE()* - For stacks that may support supervisor threads only. These stacks use less memory if CONFIG_USERSPACE is enabled.

The stack_size parameter has constraints. It must either be:

- The original size value passed to *K_THREAD_STACK_DEFINE()* or *K_KERNEL_STACK_DEFINE()*
- The return value of *K_THREAD_STACK_SIZEOF(stack)* if the stack was defined with *K_THREAD_STACK_DEFINE()*

- The return value of *K_KERNEL_STACK_SIZEOF(stack)* if the stack was defined with *K_KERNEL_STACK_DEFINE()*.

Using other values, or sizeof(stack) may produce undefined behavior.

Parameters

- `new_thread` – Pointer to uninitialized struct *k_thread*
- `stack` – Pointer to the stack space.
- `stack_size` – Stack size in bytes.
- `entry` – Thread entry function.
- `p1` – 1st entry point parameter.
- `p2` – 2nd entry point parameter.
- `p3` – 3rd entry point parameter.
- `prio` – Thread priority.
- `options` – Thread options.
- `delay` – Scheduling delay, or K_NO_WAIT (for no delay).

Returns

ID of new thread.

FUNC_NORETURN void **k_thread_user_mode_enter**(k_thread_entry_t entry, void *p1, void *p2, void *p3)

Drop a thread's privileges permanently to user mode.

This allows a supervisor thread to be re-used as a user thread. This function does not return, but control will transfer to the provided entry point as if this was a new user thread.

The implementation ensures that the stack buffer contents are erased. Any thread-local storage will be reverted to a pristine state.

Memory domain membership, resource pool assignment, kernel object permissions, priority, and thread options are preserved.

A common use of this function is to re-use the main thread as a user thread once all supervisor mode-only tasks have been completed.

Parameters

- `entry` – Function to start executing from
- `p1` – 1st entry point parameter
- `p2` – 2nd entry point parameter
- `p3` – 3rd entry point parameter

static inline void **k_thread_heap_assign**(struct *k_thread* *thread, struct *k_heap* *heap)

Assign a resource memory pool to a thread.

By default, threads have no resource pool assigned unless their parent thread has a resource pool, in which case it is inherited. Multiple threads may be assigned to the same memory pool.

Changing a thread's resource pool will not migrate allocations from the previous pool.

Parameters

- `thread` – Target thread to assign a memory pool for resource requests.
- `heap` – Heap object to use for resources, or NULL if the thread should no longer have a memory pool.

int **k_thread_join**(struct *k_thread* *thread, *k_timeout_t* timeout)

Sleep until a thread exits.

The caller will be put to sleep until the target thread exits, either due to being aborted, self-exiting, or taking a fatal error. This API returns immediately if the thread isn't running.

This API may only be called from ISRs with a K_NO_WAIT timeout, where it can be useful as a predicate to detect when a thread has aborted.

Parameters

- **thread** – Thread to wait to exit
- **timeout** – upper bound time to wait for the thread to exit.

Return values

- **0** – success, target thread has exited or wasn't running
- **-EBUSY** – returned without waiting
- **-EAGAIN** – waiting period timed out
- **-EDEADLK** – target thread is joining on the caller, or target thread is the caller

int32_t **k_sleep**(*k_timeout_t* timeout)

Put the current thread to sleep.

This routine puts the current thread to sleep for *duration*, specified as a *k_timeout_t* object.

Note

if *timeout* is set to K_FOREVER then the thread is suspended.

Parameters

- **timeout** – Desired duration of sleep.

Returns

Zero if the requested time has elapsed or if the thread was woken up by the *k_wakeup* call, the time left to sleep rounded up to the nearest millisecond.

static inline int32_t **k_msleep**(int32_t ms)

Put the current thread to sleep.

This routine puts the current thread to sleep for *duration* milliseconds.

Parameters

- **ms** – Number of milliseconds to sleep.

Returns

Zero if the requested time has elapsed or if the thread was woken up by the *k_wakeup* call, the time left to sleep rounded up to the nearest millisecond.

int32_t **k_usleep**(int32_t us)

Put the current thread to sleep with microsecond resolution.

This function is unlikely to work as expected without kernel tuning. In particular, because the lower bound on the duration of a sleep is the duration of a tick, CONFIG_SYS_CLOCK_TICKS_PER_SEC must be adjusted to achieve the resolution desired. The implications of doing this must be understood before attempting to use *k_usleep()*. Use with caution.

Parameters

- `us` – Number of microseconds to sleep.

Returns
Zero if the requested time has elapsed or if the thread was woken up by the *k_wakeup* call, the time left to sleep rounded up to the nearest microsecond.

void `k_busy_wait`(uint32_t usec_to_wait)

Cause the current thread to busy wait.

This routine causes the current thread to execute a "do nothing" loop for *usec_to_wait* microseconds.

Note

The clock used for the microsecond-resolution delay here may be skewed relative to the clock used for system timeouts like *k_sleep()*. For example k_busy_wait(1000) may take slightly more or less time than k_sleep(K_MSEC(1)), with the offset dependent on clock tolerances.

Note

In case when `CONFIG_SYSTEM_CLOCK_SLOPPY_IDLE` and `CONFIG_PM` options are enabled, this function may not work. The timer/clock used for delay processing may be disabled/inactive.

bool `k_can_yield`(void)

Check whether it is possible to yield in the current context.

This routine checks whether the kernel is in a state where it is possible to yield or call blocking API's. It should be used by code that needs to yield to perform correctly, but can feasibly be called from contexts where that is not possible. For example in the PRE_KERNEL initialization step, or when being run from the idle thread.

Returns
True if it is possible to yield in the current context, false otherwise.

void `k_yield`(void)

Yield the current thread.

This routine causes the current thread to yield execution to another thread of the same or higher priority. If there are no other ready threads of the same or higher priority, the routine returns immediately.

void `k_wakeup`(k_tid_t thread)

Wake up a sleeping thread.

This routine prematurely wakes up *thread* from sleeping.

If *thread* is not currently sleeping, the routine has no effect.

Parameters

- `thread` – ID of thread to wake.

__attribute_const__ k_tid_t `k_sched_current_thread_query`(void)

Query thread ID of the current thread.

This unconditionally queries the kernel via a system call.

Note

Use *k_current_get()* unless absolutely sure this is necessary. This should only be used directly where the thread local variable cannot be used or may contain invalid values if thread local storage (TLS) is enabled. If TLS is not enabled, this is the same as *k_current_get()*.

Returns
ID of current thread.

__attribute_const__ static inline k_tid_t **k_current_get**(void)

Get thread ID of the current thread.

Returns
ID of current thread.

void **k_thread_abort**(k_tid_t thread)

Abort a thread.

This routine permanently stops execution of *thread*. The thread is taken off all kernel queues it is part of (i.e. the ready queue, the timeout queue, or a kernel object wait queue). However, any kernel resources the thread might currently own (such as mutexes or memory blocks) are not released. It is the responsibility of the caller of this routine to ensure all necessary cleanup is performed.

After *k_thread_abort()* returns, the thread is guaranteed not to be running or to become runnable anywhere on the system. Normally this is done via blocking the caller (in the same manner as *k_thread_join()*), but in interrupt context on SMP systems the implementation is required to spin for threads that are running on other CPUs.

Parameters

- **thread** – ID of thread to abort.

void **k_thread_start**(k_tid_t thread)

Start an inactive thread.

If a thread was created with K_FOREVER in the delay parameter, it will not be added to the scheduling queue until this function is called on it.

Parameters

- **thread** – thread to start

k_ticks_t **k_thread_timeout_expires_ticks**(const struct *k_thread* *thread)

Get time when a thread wakes up, in system ticks.

This routine computes the system uptime when a waiting thread next executes, in units of system ticks. If the thread is not waiting, it returns current system time.

k_ticks_t **k_thread_timeout_remaining_ticks**(const struct *k_thread* *thread)

Get time remaining before a thread wakes up, in system ticks.

This routine computes the time remaining before a waiting thread next executes, in units of system ticks. If the thread is not waiting, it returns zero.

int **k_thread_priority_get**(k_tid_t thread)

Get a thread's priority.

This routine gets the priority of *thread*.

Parameters

- **thread** – ID of thread whose priority is needed.

Returns
Priority of *thread*.

void **k_thread_priority_set**(k_tid_t thread, int prio)

Set a thread's priority.

This routine immediately changes the priority of *thread*.

Rescheduling can occur immediately depending on the priority *thread* is set to:

- If its priority is raised above the priority of the caller of this function, and the caller is preemptible, *thread* will be scheduled in.
- If the caller operates on itself, it lowers its priority below that of other threads in the system, and the caller is preemptible, the thread of highest priority will be scheduled in.

Priority can be assigned in the range of -CONFIG_NUM_COOP_PRIORITIES to CONFIG_NUM_PREEMPT_PRIORITIES-1, where -CONFIG_NUM_COOP_PRIORITIES is the highest priority.

Warning

Changing the priority of a thread currently involved in mutex priority inheritance may result in undefined behavior.

Parameters

- **thread** – ID of thread whose priority is to be set.
- **prio** – New priority.

void **k_thread_deadline_set**(k_tid_t thread, int deadline)

Set deadline expiration time for scheduler.

This sets the "deadline" expiration as a time delta from the current time, in the same units used by *k_cycle_get_32()*. The scheduler (when deadline scheduling is enabled) will choose the next expiring thread when selecting between threads at the same static priority. Threads at different priorities will be scheduled according to their static priority.

Note

Deadlines are stored internally using 32 bit unsigned integers. The number of cycles between the "first" deadline in the scheduler queue and the "last" deadline must be less than 2^31 (i.e a signed non-negative quantity). Failure to adhere to this rule may result in scheduled threads running in an incorrect deadline order.

Note

Despite the API naming, the scheduler makes no guarantees the thread WILL be scheduled within that deadline, nor does it take extra metadata (like e.g. the "runtime" and "period" parameters in Linux sched_setattr()) that allows the kernel to validate the scheduling for achievability. Such features could be implemented above this call, which is simply input to the priority selection logic.

Note

You should enable CONFIG_SCHED_DEADLINE in your project configuration.

Parameters

- **thread** – A thread on which to set the deadline
- **deadline** – A time delta, in cycle units

int **k_thread_cpu_mask_clear**(k_tid_t thread)

Sets all CPU enable masks to zero.

After this returns, the thread will no longer be schedulable on any CPUs. The thread must not be currently runnable.

Note

You should enable CONFIG_SCHED_CPU_MASK in your project configuration.

Parameters

- **thread** – Thread to operate upon

Returns

Zero on success, otherwise error code

int **k_thread_cpu_mask_enable_all**(k_tid_t thread)

Sets all CPU enable masks to one.

After this returns, the thread will be schedulable on any CPU. The thread must not be currently runnable.

Note

You should enable CONFIG_SCHED_CPU_MASK in your project configuration.

Parameters

- **thread** – Thread to operate upon

Returns

Zero on success, otherwise error code

int **k_thread_cpu_mask_enable**(k_tid_t thread, int cpu)

Enable thread to run on specified CPU.

The thread must not be currently runnable.

Note

You should enable CONFIG_SCHED_CPU_MASK in your project configuration.

Parameters

- **thread** – Thread to operate upon
- **cpu** – CPU index

Returns
Zero on success, otherwise error code

int `k_thread_cpu_mask_disable`(k_tid_t thread, int cpu)
Prevent thread to run on specified CPU.

The thread must not be currently runnable.

> **Note**
> You should enable `CONFIG_SCHED_CPU_MASK` in your project configuration.

Parameters
- `thread` – Thread to operate upon
- `cpu` – CPU index

Returns
Zero on success, otherwise error code

int `k_thread_cpu_pin`(k_tid_t thread, int cpu)
Pin a thread to a CPU.

Pin a thread to a CPU by first clearing the cpu mask and then enabling the thread on the selected CPU.

Parameters
- `thread` – Thread to operate upon
- `cpu` – CPU index

Returns
Zero on success, otherwise error code

void `k_thread_suspend`(k_tid_t thread)
Suspend a thread.

This routine prevents the kernel scheduler from making *thread* the current thread. All other internal operations on *thread* are still performed; for example, kernel objects it is waiting on are still handed to it. Note that any existing timeouts (e.g. *k_sleep()*, or a timeout argument to *k_sem_take()* et. al.) will be canceled. On resume, the thread will begin running immediately and return from the blocked call.

When the target thread is active on another CPU, the caller will block until the target thread is halted (suspended or aborted). But if the caller is in an interrupt context, it will spin waiting for that target thread active on another CPU to halt.

If *thread* is already suspended, the routine has no effect.

Parameters
- `thread` – ID of thread to suspend.

void `k_thread_resume`(k_tid_t thread)
Resume a suspended thread.

This routine allows the kernel scheduler to make *thread* the current thread, when it is next eligible for that role.

If *thread* is not currently suspended, the routine has no effect.

Parameters
- `thread` – ID of thread to resume.

void **k_sched_time_slice_set**(int32_t slice, int prio)

Set time-slicing period and scope.

This routine specifies how the scheduler will perform time slicing of preemptible threads.

To enable time slicing, *slice* must be non-zero. The scheduler ensures that no thread runs for more than the specified time limit before other threads of that priority are given a chance to execute. Any thread whose priority is higher than *prio* is exempted, and may execute as long as desired without being preempted due to time slicing.

Time slicing only limits the maximum amount of time a thread may continuously execute. Once the scheduler selects a thread for execution, there is no minimum guaranteed time the thread will execute before threads of greater or equal priority are scheduled.

When the current thread is the only one of that priority eligible for execution, this routine has no effect; the thread is immediately rescheduled after the slice period expires.

To disable timeslicing, set both *slice* and *prio* to zero.

Parameters

- **slice** – Maximum time slice length (in milliseconds).
- **prio** – Highest thread priority level eligible for time slicing.

void **k_thread_time_slice_set**(struct *k_thread* *th, int32_t slice_ticks,
k_thread_timeslice_fn_t expired, void *data)

Set thread time slice.

As for k_sched_time_slice_set, but (when CONFIG_TIMESLICE_PER_THREAD=y) sets the timeslice for a specific thread. When non-zero, this timeslice will take precedence over the global value.

When such a thread's timeslice expires, the configured callback will be called before the thread is removed/re-added to the run queue. This callback will occur in interrupt context, and the specified thread is guaranteed to have been preempted by the currently-executing ISR. Such a callback is free to, for example, modify the thread priority or slice time for future execution, suspend the thread, etc...

Note

Unlike the older API, the time slice parameter here is specified in ticks, not milliseconds. Ticks have always been the internal unit, and not all platforms have integer conversions between the two.

Note

Threads with a non-zero slice time set will be timesliced always, even if they are higher priority than the maximum timeslice priority set via *k_sched_time_slice_set()*.

Note

The callback notification for slice expiration happens, as it must, while the thread is still "current", and thus it happens before any registered timeouts at this tick. This has the somewhat confusing side effect that the tick time (c.f. *k_uptime_get()*) does not yet reflect the expired ticks. Applications wishing to make fine-grained

timing decisions within this callback should use the cycle API, or derived facilities like k_thread_runtime_stats_get().

Parameters

- `th` – A valid, initialized thread
- `slice_ticks` – Maximum timeslice, in ticks
- `expired` – Callback function called on slice expiration
- `data` – Parameter for the expiration handler

void `k_sched_lock`(void)

Lock the scheduler.

This routine prevents the current thread from being preempted by another thread by instructing the scheduler to treat it as a cooperative thread. If the thread subsequently performs an operation that makes it unready, it will be context switched out in the normal manner. When the thread again becomes the current thread, its non-preemptible status is maintained.

This routine can be called recursively.

Owing to clever implementation details, scheduler locks are extremely fast for non-userspace threads (just one byte inc/decrement in the thread struct).

Note

This works by elevating the thread priority temporarily to a cooperative priority, allowing cheap synchronization vs. other preemptible or cooperative threads running on the current CPU. It does not prevent preemption or asynchrony of other types. It does not prevent threads from running on other CPUs when CONFIG_SMP=y. It does not prevent interrupts from happening, nor does it prevent threads with MetaIRQ priorities from preempting the current thread. In general this is a historical API not well-suited to modern applications, use with care.

void `k_sched_unlock`(void)

Unlock the scheduler.

This routine reverses the effect of a previous call to *k_sched_lock()*. A thread must call the routine once for each time it called *k_sched_lock()* before the thread becomes preemptible.

void `k_thread_custom_data_set`(void *value)

Set current thread's custom data.

This routine sets the custom data for the current thread to @ value.

Custom data is not used by the kernel itself, and is freely available for a thread to use as it sees fit. It can be used as a framework upon which to build thread-local storage.

Parameters

- `value` – New custom data value.

void *`k_thread_custom_data_get`(void)

Get current thread's custom data.

This routine returns the custom data for the current thread.

Returns

Current custom data value.

int k_thread_name_set(k_tid_t thread, const char *str)

Set current thread name.

Set the name of the thread to be used when CONFIG_THREAD_MONITOR is enabled for tracing and debugging.

Parameters

- thread – Thread to set name, or NULL to set the current thread
- str – Name string

Return values

- 0 – on success
- -EFAULT – Memory access error with supplied string
- -ENOSYS – Thread name configuration option not enabled
- -EINVAL – Thread name too long

const char *k_thread_name_get(k_tid_t thread)

Get thread name.

Get the name of a thread

Parameters

- thread – Thread ID

Return values

Thread – name, or NULL if configuration not enabled

int k_thread_name_copy(k_tid_t thread, char *buf, size_t size)

Copy the thread name into a supplied buffer.

Parameters

- thread – Thread to obtain name information
- buf – Destination buffer
- size – Destination buffer size

Return values

- -ENOSPC – Destination buffer too small
- -EFAULT – Memory access error
- -ENOSYS – Thread name feature not enabled
- 0 – Success

const char *k_thread_state_str(k_tid_t thread_id, char *buf, size_t buf_size)

Get thread state string.

This routine generates a human friendly string containing the thread's state, and copies as much of it as possible into *buf*.

Parameters

- thread_id – Thread ID
- buf – Buffer into which to copy state strings
- buf_size – Size of the buffer

Return values

Pointer – to *buf* if data was copied, else a pointer to "".

struct **k_thread**

#include <*thread.h*> Thread Structure.

Public Members

struct _callee_saved **callee_saved**

defined by the architecture, but all archs need these

void ***init_data**

static thread init data

_wait_q_t **join_queue**

threads waiting in *k_thread_join()*

struct __thread_entry **entry**

thread entry and parameters description

struct *k_thread* ***next_thread**

next item in list of all threads

void ***custom_data**

crude thread-local storage

struct _thread_stack_info **stack_info**

Stack Info.

struct _mem_domain_info **mem_domain_info**

memory domain info of the thread

k_thread_stack_t ***stack_obj**

Base address of thread stack.

If memory mapped stack (CONFIG_THREAD_STACK_MEM_MAPPED) is enabled, this is the physical address of the stack.

void ***syscall_frame**

current syscall frame pointer

int **swap_retval**

z_swap() return value

void ***switch_handle**

Context handle returned via *arch_switch()*

struct *k_heap* ***resource_pool**

resource pool

_wait_q_t **halt_queue**

threads waiting in *k_thread_suspend()*

struct _thread_arch **arch**

arch-specifics: must always be at the end

group **thread_stack_api**

Thread Stack APIs.

Defines

K_KERNEL_STACK_DECLARE(sym, size)

Declare a reference to a thread stack.

This macro declares the symbol of a thread stack defined elsewhere in the current scope.

Parameters

- **sym** – Thread stack symbol name
- **size** – Size of the stack memory region

K_KERNEL_STACK_ARRAY_DECLARE(sym, nmemb, size)

Declare a reference to a thread stack array.

This macro declares the symbol of a thread stack array defined elsewhere in the current scope.

Parameters

- **sym** – Thread stack symbol name
- **nmemb** – Number of stacks defined
- **size** – Size of the stack memory region

K_KERNEL_PINNED_STACK_ARRAY_DECLARE(sym, nmemb, size)

Declare a reference to a pinned thread stack array.

This macro declares the symbol of a pinned thread stack array defined elsewhere in the current scope.

Parameters

- **sym** – Thread stack symbol name
- **nmemb** – Number of stacks defined
- **size** – Size of the stack memory region

K_KERNEL_STACK_DEFINE(sym, size)

Define a toplevel kernel stack memory region.

This defines a region of memory for use as a thread stack, for threads that exclusively run in supervisor mode. This is also suitable for declaring special stacks for interrupt or exception handling.

Stacks defined with this macro may not host user mode threads.

It is legal to precede this definition with the 'static' keyword.

It is NOT legal to take the sizeof(sym) and pass that to the stackSize parameter of *k_thread_create()*, it may not be the same as the 'size' parameter. Use *K_KERNEL_STACK_SIZEOF()* instead.

The total amount of memory allocated may be increased to accommodate fixed-size stack overflow guards.

Parameters

- **sym** – Thread stack symbol name
- **size** – Size of the stack memory region

K_KERNEL_PINNED_STACK_DEFINE(sym, size)

Define a toplevel kernel stack memory region in pinned section.

See *K_KERNEL_STACK_DEFINE()* for more information and constraints.

This puts the stack into the pinned noinit linker section if CONFIG_LINKER_USE_PINNED_SECTION is enabled, or else it would put the stack into the same section as *K_KERNEL_STACK_DEFINE()*.

Parameters

- **sym** – Thread stack symbol name
- **size** – Size of the stack memory region

K_KERNEL_STACK_ARRAY_DEFINE(sym, nmemb, size)

Define a toplevel array of kernel stack memory regions.

Stacks defined with this macro may not host user mode threads.

Parameters

- **sym** – Kernel stack array symbol name
- **nmemb** – Number of stacks to define
- **size** – Size of the stack memory region

K_KERNEL_PINNED_STACK_ARRAY_DEFINE(sym, nmemb, size)

Define a toplevel array of kernel stack memory regions in pinned section.

See *K_KERNEL_STACK_ARRAY_DEFINE()* for more information and constraints.

This puts the stack into the pinned noinit linker section if CONFIG_LINKER_USE_PINNED_SECTION is enabled, or else it would put the stack into the same section as *K_KERNEL_STACK_ARRAY_DEFINE()*.

Parameters

- **sym** – Kernel stack array symbol name
- **nmemb** – Number of stacks to define
- **size** – Size of the stack memory region

K_KERNEL_STACK_MEMBER(sym, size)

Define an embedded stack memory region.

Used for kernel stacks embedded within other data structures.

Stacks defined with this macro may not host user mode threads.

Parameters

- **sym** – Thread stack symbol name
- **size** – Size of the stack memory region

K_KERNEL_STACK_SIZEOF(sym)

K_THREAD_STACK_DECLARE(sym, size)

Declare a reference to a thread stack.

This macro declares the symbol of a thread stack defined elsewhere in the current scope.

Parameters

- **sym** – Thread stack symbol name
- **size** – Size of the stack memory region

K_THREAD_STACK_ARRAY_DECLARE(sym, nmemb, size)

Declare a reference to a thread stack array.

This macro declares the symbol of a thread stack array defined elsewhere in the current scope.

Parameters

- **sym** – Thread stack symbol name
- **nmemb** – Number of stacks defined
- **size** – Size of the stack memory region

K_THREAD_STACK_SIZEOF(sym)

Return the size in bytes of a stack memory region.

Convenience macro for passing the desired stack size to *k_thread_create()* since the underlying implementation may actually create something larger (for instance a guard area).

The value returned here is not guaranteed to match the 'size' parameter passed to K_THREAD_STACK_DEFINE and may be larger, but is always safe to pass to *k_thread_create()* for the associated stack object.

Parameters

- **sym** – Stack memory symbol

Returns

Size of the stack buffer

K_THREAD_STACK_DEFINE(sym, size)

Define a toplevel thread stack memory region.

This defines a region of memory suitable for use as a thread's stack.

This is the generic, historical definition. Align to Z_THREAD_STACK_OBJ_ALIGN and put in 'noinit' section so that it isn't zeroed at boot

The defined symbol will always be a k_thread_stack_t which can be passed to *k_thread_create()*, but should otherwise not be manipulated. If the buffer inside needs to be examined, examine thread->stack_info for the associated thread object to obtain the boundaries.

It is legal to precede this definition with the 'static' keyword.

It is NOT legal to take the sizeof(sym) and pass that to the stackSize parameter of *k_thread_create()*, it may not be the same as the 'size' parameter. Use *K_THREAD_STACK_SIZEOF()* instead.

Some arches may round the size of the usable stack region up to satisfy alignment constraints. *K_THREAD_STACK_SIZEOF()* will return the aligned size.

Parameters

- **sym** – Thread stack symbol name
- **size** – Size of the stack memory region

K_THREAD_PINNED_STACK_DEFINE(sym, size)

Define a toplevel thread stack memory region in pinned section.

This defines a region of memory suitable for use as a thread's stack.

This is the generic, historical definition. Align to Z_THREAD_STACK_OBJ_ALIGN and put in 'noinit' section so that it isn't zeroed at boot

The defined symbol will always be a k_thread_stack_t which can be passed to *k_thread_create()*, but should otherwise not be manipulated. If the buffer inside needs to be examined, examine thread->stack_info for the associated thread object to obtain the boundaries.

It is legal to precede this definition with the 'static' keyword.

It is NOT legal to take the sizeof(sym) and pass that to the stackSize parameter of *k_thread_create()*, it may not be the same as the 'size' parameter. Use *K_THREAD_STACK_SIZEOF()* instead.

Some arches may round the size of the usable stack region up to satisfy alignment constraints. *K_THREAD_STACK_SIZEOF()* will return the aligned size.

This puts the stack into the pinned noinit linker section if CONFIG_LINKER_USE_PINNED_SECTION is enabled, or else it would put the stack into the same section as *K_THREAD_STACK_DEFINE()*.

Parameters

- **sym** – Thread stack symbol name
- **size** – Size of the stack memory region

K_THREAD_STACK_LEN(size)

Calculate size of stacks to be allocated in a stack array.

This macro calculates the size to be allocated for the stacks inside a stack array. It accepts the indicated "size" as a parameter and if required, pads some extra bytes (e.g. for MPU scenarios). Refer K_THREAD_STACK_ARRAY_DEFINE definition to see how this is used. The returned size ensures each array member will be aligned to the required stack base alignment.

Parameters

- **size** – Size of the stack memory region

Returns

Appropriate size for an array member

K_THREAD_STACK_ARRAY_DEFINE(sym, nmemb, size)

Define a toplevel array of thread stack memory regions.

Create an array of equally sized stacks. See K_THREAD_STACK_DEFINE definition for additional details and constraints.

This is the generic, historical definition. Align to Z_THREAD_STACK_OBJ_ALIGN and put in 'noinit' section so that it isn't zeroed at boot

Parameters

- **sym** – Thread stack symbol name
- **nmemb** – Number of stacks to define
- **size** – Size of the stack memory region

K_THREAD_PINNED_STACK_ARRAY_DEFINE(sym, nmemb, size)

Define a toplevel array of thread stack memory regions in pinned section.

Create an array of equally sized stacks. See K_THREAD_STACK_DEFINE definition for additional details and constraints.

This is the generic, historical definition. Align to Z_THREAD_STACK_OBJ_ALIGN and put in 'noinit' section so that it isn't zeroed at boot

This puts the stack into the pinned noinit linker section if CONFIG_LINKER_USE_PINNED_SECTION is enabled, or else it would put the stack into the same section as *K_THREAD_STACK_DEFINE()*.

Parameters

- **sym** – Thread stack symbol name
- **nmemb** – Number of stacks to define
- **size** – Size of the stack memory region

K_THREAD_STACK_MEMBER(sym, size)

Define an embedded stack memory region.

Used for stacks embedded within other data structures. Use is highly discouraged but in some cases necessary. For memory protection scenarios, it is very important that any RAM preceding this member not be writable by threads else a stack overflow will lead to silent corruption. In other words, the containing data structure should live in RAM owned by the kernel.

A user thread can only be started with a stack defined in this way if the thread starting it is in supervisor mode.

Deprecated:

This is now deprecated, as stacks defined in this way are not usable from user mode. Use K_KERNEL_STACK_MEMBER.

Parameters

- **sym** – Thread stack symbol name
- **size** – Size of the stack memory region

Scheduling

The kernel's priority-based scheduler allows an application's threads to share the CPU.

Concepts The scheduler determines which thread is allowed to execute at any point in time; this thread is known as the **current thread**.

There are various points in time when the scheduler is given an opportunity to change the identity of the current thread. These points are called **reschedule points**. Some potential reschedule points are:

- transition of a thread from running state to a suspended or waiting state, for example by *k_sem_take()* or *k_sleep()*.
- transition of a thread to the *ready state*, for example by *k_sem_give()* or *k_thread_start()*
- return to thread context after processing an interrupt
- when a running thread invokes *k_yield()*

A thread **sleeps** when it voluntarily initiates an operation that transitions itself to a suspended or waiting state.

Whenever the scheduler changes the identity of the current thread, or when execution of the current thread is replaced by an ISR, the kernel first saves the current thread's CPU register values. These register values get restored when the thread later resumes execution.

Scheduling Algorithm The kernel's scheduler selects the highest priority ready thread to be the current thread. When multiple ready threads of the same priority exist, the scheduler chooses the one that has been waiting longest.

A thread's relative priority is primarily determined by its static priority. However, when both earliest-deadline-first scheduling is enabled (`CONFIG_SCHED_DEADLINE`) and a choice of threads have equal static priority, then the thread with the earlier deadline is considered to have the higher priority. Thus, when earliest-deadline-first scheduling is enabled, two threads are only considered to have the same priority when both their static priorities and deadlines are equal. The routine *`k_thread_deadline_set()`* is used to set a thread's deadline.

Note

Execution of ISRs takes precedence over thread execution, so the execution of the current thread may be replaced by an ISR at any time unless interrupts have been masked. This applies to both cooperative threads and preemptive threads.

The kernel can be built with one of several choices for the ready queue implementation, offering different choices between code size, constant factor runtime overhead and performance scaling when many threads are added.

- Simple linked-list ready queue (`CONFIG_SCHED_DUMB`)

  The scheduler ready queue will be implemented as a simple unordered list, with very fast constant time performance for single threads and very low code size. This implementation should be selected on systems with constrained code size that will never see more than a small number (3, maybe) of runnable threads in the queue at any given time. On most platforms (that are not otherwise using the red/black tree) this results in a savings of ~2k of code size.

- Red/black tree ready queue (`CONFIG_SCHED_SCALABLE`)

  The scheduler ready queue will be implemented as a red/black tree. This has rather slower constant-time insertion and removal overhead, and on most platforms (that are not otherwise using the red/black tree somewhere) requires an extra ~2kb of code. The resulting behavior will scale cleanly and quickly into the many thousands of threads.

  Use this for applications needing many concurrent runnable threads (> 20 or so). Most applications won't need this ready queue implementation.

- Traditional multi-queue ready queue (`CONFIG_SCHED_MULTIQ`)

  When selected, the scheduler ready queue will be implemented as the classic/textbook array of lists, one per priority.

  This corresponds to the scheduler algorithm used in Zephyr versions prior to 1.12.

  It incurs only a tiny code size overhead vs. the "dumb" scheduler and runs in O(1) time in almost all circumstances with very low constant factor. But it requires a fairly large RAM budget to store those list heads, and the limited features make it incompatible with features like deadline scheduling that need to sort threads more finely, and SMP affinity which need to traverse the list of threads.

  Typical applications with small numbers of runnable threads probably want the DUMB scheduler.

The wait_q abstraction used in IPC primitives to pend threads for later wakeup shares the same backend data structure choices as the scheduler, and can use the same options.

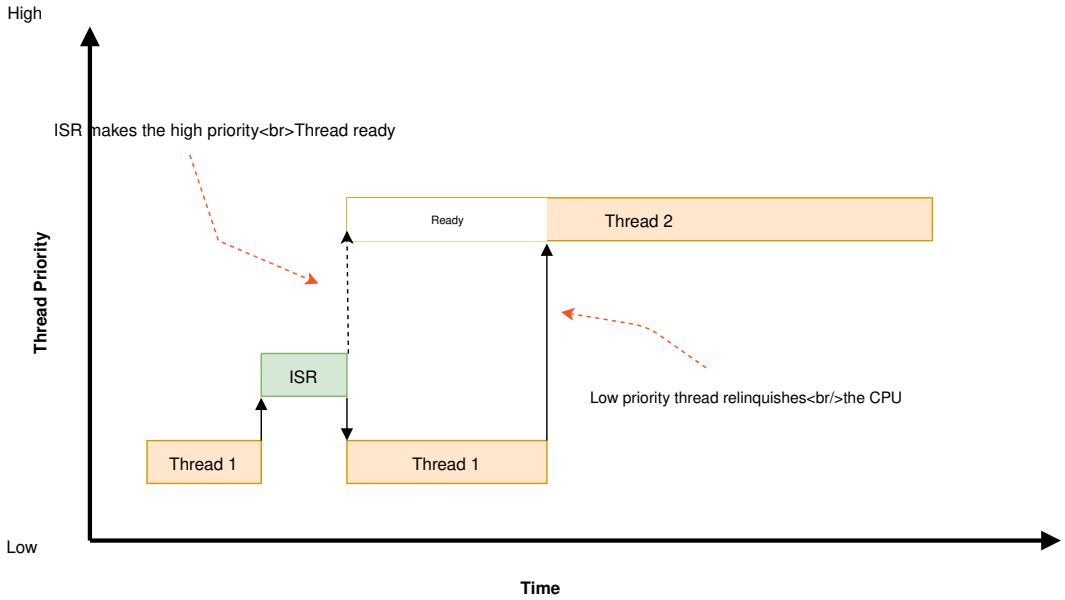
- Scalable wait_q implementation (`CONFIG_WAITQ_SCALABLE`)

  When selected, the wait_q will be implemented with a balanced tree. Choose this if you expect to have many threads waiting on individual primitives. There is a ~2kb code size increase over `CONFIG_WAITQ_DUMB` (which may be shared with `CONFIG_SCHED_SCALABLE`) if the red/black tree is not used elsewhere in the application, and pend/unpend operations on "small" queues will be somewhat slower (though this is not generally a performance path).

- Simple linked-list wait_q (`CONFIG_WAITQ_DUMB`)

  When selected, the wait_q will be implemented with a doubly-linked list. Choose this if you expect to have only a few threads blocked on any single IPC primitive.

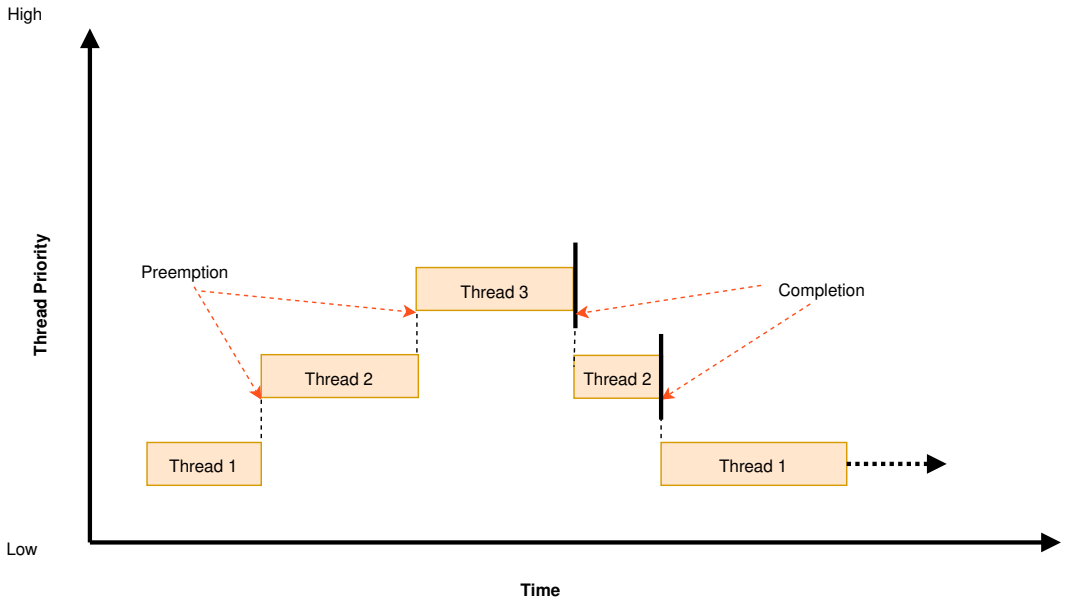
Cooperative Time Slicing Once a cooperative thread becomes the current thread, it remains the current thread until it performs an action that makes it unready. Consequently, if a cooperative thread performs lengthy computations, it may cause an unacceptable delay in the scheduling of other threads, including those of higher priority and equal priority.



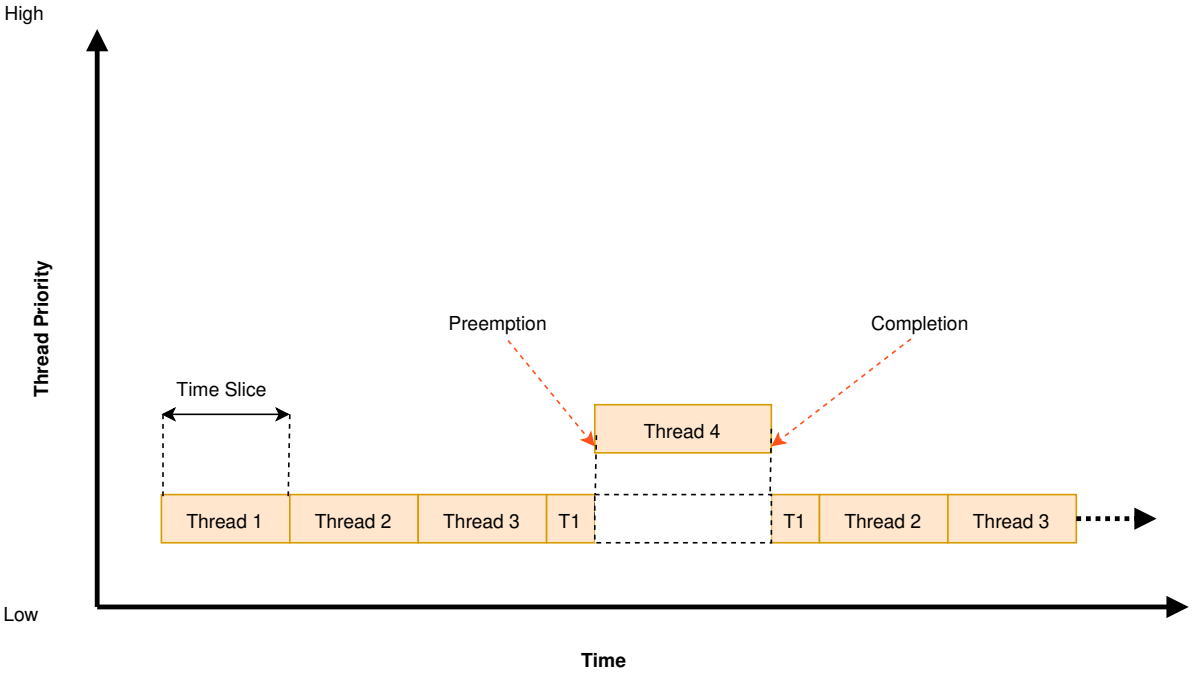
To overcome such problems, a cooperative thread can voluntarily relinquish the CPU from time to time to permit other threads to execute. A thread can relinquish the CPU in two ways:

- Calling *k_yield()* puts the thread at the back of the scheduler's prioritized list of ready threads, and then invokes the scheduler. All ready threads whose priority is higher or equal to that of the yielding thread are then allowed to execute before the yielding thread is rescheduled. If no such ready threads exist, the scheduler immediately reschedules the yielding thread without context switching.
- Calling *k_sleep()* makes the thread unready for a specified time period. Ready threads of *all* priorities are then allowed to execute; however, there is no guarantee that threads whose priority is lower than that of the sleeping thread will actually be scheduled before the sleeping thread becomes ready once again.

Preemptive Time Slicing Once a preemptive thread becomes the current thread, it remains the current thread until a higher priority thread becomes ready, or until the thread performs an action that makes it unready. Consequently, if a preemptive thread performs lengthy computations, it may cause an unacceptable delay in the scheduling of other threads, including those of equal priority.



To overcome such problems, a preemptive thread can perform cooperative time slicing (as described above), or the scheduler's time slicing capability can be used to allow other threads of the same priority to execute.



The scheduler divides time into a series of **time slices**, where slices are measured in system clock ticks. The time slice size is configurable, but this size can be changed while the application is running.

At the end of every time slice, the scheduler checks to see if the current thread is preemptible and, if so, implicitly invokes `k_yield()` on behalf of the thread. This gives other ready threads of the same priority the opportunity to execute before the current thread is scheduled again. If no threads of equal priority are ready, the current thread remains the current thread.

Threads with a priority higher than specified limit are exempt from preemptive time slicing, and are never preempted by a thread of equal priority. This allows an application to use preemptive time slicing only when dealing with lower priority threads that are less time-sensitive.

Note

The kernel's time slicing algorithm does *not* ensure that a set of equal-priority threads receive an equitable amount of CPU time, since it does not measure the amount of time a thread actually gets to execute. However, the algorithm *does* ensure that a thread never executes for longer than a single time slice without being required to yield.

Scheduler Locking A preemptible thread that does not wish to be preempted while performing a critical operation can instruct the scheduler to temporarily treat it as a cooperative thread by calling `k_sched_lock()`. This prevents other threads from interfering while the critical operation is being performed.

Once the critical operation is complete the preemptible thread must call `k_sched_unlock()` to restore its normal, preemptible status.

If a thread calls `k_sched_lock()` and subsequently performs an action that makes it unready, the scheduler will switch the locking thread out and allow other threads to execute. When the locking thread again becomes the current thread, its non-preemptible status is maintained.

Note

Locking out the scheduler is a more efficient way for a preemptible thread to prevent preemption than changing its priority level to a negative value.

Thread Sleeping A thread can call `k_sleep()` to delay its processing for a specified time period. During the time the thread is sleeping the CPU is relinquished to allow other ready threads to execute. Once the specified delay has elapsed the thread becomes ready and is eligible to be scheduled once again.

A sleeping thread can be woken up prematurely by another thread using `k_wakeup()`. This technique can sometimes be used to permit the secondary thread to signal the sleeping thread that something has occurred *without* requiring the threads to define a kernel synchronization object, such as a semaphore. Waking up a thread that is not sleeping is allowed, but has no effect.

Busy Waiting A thread can call `k_busy_wait()` to perform a `busy wait` that delays its processing for a specified time period *without* relinquishing the CPU to another ready thread.

A busy wait is typically used instead of thread sleeping when the required delay is too short to warrant having the scheduler context switch from the current thread to another thread and then back again.

Suggested Uses Use cooperative threads for device drivers and other performance-critical work.

Use cooperative threads to implement mutually exclusion without the need for a kernel object, such as a mutex.

Use preemptive threads to give priority to time-sensitive processing over less time-sensitive processing.

CPU Idling

Although normally reserved for the idle thread, in certain special applications, a thread might want to make the CPU idle.

- *Concepts*
- *Implementation*
  - *Making the CPU idle*
  - *Making the CPU idle in an atomic fashion*
- *Suggested Uses*
- *API Reference*

Concepts Making the CPU idle causes the kernel to pause all operations until an event, normally an interrupt, wakes up the CPU. In a regular system, the idle thread is responsible for this. However, in some constrained systems, it is possible that another thread takes this duty.

Implementation

Making the CPU idle Making the CPU idle is simple: call the k_cpu_idle() API. The CPU will stop executing instructions until an event occurs. Most likely, the function will be called within a loop. Note that in certain architectures, upon return, k_cpu_idle() unconditionally unmasks interrupts.

```
static k_sem my_sem;

void my_isr(void *unused)
{
    k_sem_give(&my_sem);
}

int main(void)
{
    k_sem_init(&my_sem, 0, 1);

    /* wait for semaphore from ISR, then do related work */

    for (;;) {

        /* wait for ISR to trigger work to perform */
        if (k_sem_take(&my_sem, K_NO_WAIT) == 0) {

            /* ... do processing */

        }

        /* put CPU to sleep to save power */
        k_cpu_idle();
    }
}
```

Making the CPU idle in an atomic fashion It is possible that there is a need to do some work atomically before making the CPU idle. In such a case, k_cpu_atomic_idle() should be used instead.

In fact, there is a race condition in the previous example: the interrupt could occur between the time the semaphore is taken, finding out it is not available and making the CPU idle again. In

some systems, this can cause the CPU to idle until *another* interrupt occurs, which might be *never*, thus hanging the system completely. To prevent this, k_cpu_atomic_idle() should have been used, like in this example.

```
static k_sem my_sem;

void my_isr(void *unused)
{
    k_sem_give(&my_sem);
}

int main(void)
{
    k_sem_init(&my_sem, 0, 1);

    for (;;) {

        unsigned int key = irq_lock();

        /*
         * Wait for semaphore from ISR; if acquired, do related work, then
         * go to next loop iteration (the semaphore might have been given
         * again); else, make the CPU idle.
         */

        if (k_sem_take(&my_sem, K_NO_WAIT) == 0) {

            irq_unlock(key);

            /* ... do processing */


        } else {
            /* put CPU to sleep to save power */
            k_cpu_atomic_idle(key);
        }
    }
}
```

Suggested Uses Use k_cpu_atomic_idle() when a thread has to do some real work in addition to idling the CPU to wait for an event. See example above.

Use k_cpu_idle() only when a thread is only responsible for idling the CPU, i.e. not doing any real work, like in this example below.

```
int main(void)
{
    /* ... do some system/application initialization */


    /* thread is only used for CPU idling from this point on */
    for (;;) {
        k_cpu_idle();
    }
}
```

Note

Do not use these APIs unless absolutely necessary. In a normal system, the idle thread takes care of power management, including CPU idling.

API Reference

group `cpu_idle_apis`

Functions

static inline void **k_cpu_idle**(void)

Make the CPU idle.

This function makes the CPU idle until an event wakes it up.

In a regular system, the idle thread should be the only thread responsible for making the CPU idle and triggering any type of power management. However, in some more constrained systems, such as a single-threaded system, the only thread would be responsible for this if needed.

> **Note**
>
> In some architectures, before returning, the function unmasks interrupts unconditionally.

static inline void **k_cpu_atomic_idle**(unsigned int key)

Make the CPU idle in an atomic fashion.

Similar to *k_cpu_idle()*, but must be called with interrupts locked.

Enabling interrupts and entering a low-power mode will be atomic, i.e. there will be no period of time where interrupts are enabled before the processor enters a low-power mode.

After waking up from the low-power mode, the interrupt lockout state will be restored as if by *irq_unlock(key)*.

Parameters

- `key` – Interrupt locking key obtained from *irq_lock()*.

System Threads

- *Implementation*
  - *Writing a main() function*
- *Suggested Uses*

A *system thread* is a thread that the kernel spawns automatically during system initialization.

The kernel spawns the following system threads:

Main thread

This thread performs kernel initialization, then calls the application's `main()` function (if one is defined).

By default, the main thread uses the highest configured preemptible thread priority (i.e. 0). If the kernel is not configured to support preemptible threads, the main thread uses the lowest configured cooperative thread priority (i.e. -1).

The main thread is an essential thread while it is performing kernel initialization or executing the application's `main()` function; this means a fatal system error is raised if the thread aborts. If `main()` is not defined, or if it executes and then does a normal return, the main thread terminates normally and no error is raised.

Idle thread
: This thread executes when there is no other work for the system to do. If possible, the idle thread activates the board's power management support to save power; otherwise, the idle thread simply performs a "do nothing" loop. The idle thread remains in existence as long as the system is running and never terminates.

  The idle thread always uses the lowest configured thread priority. If this makes it a cooperative thread, the idle thread repeatedly yields the CPU to allow the application's other threads to run when they need to.

  The idle thread is an essential thread, which means a fatal system error is raised if the thread aborts.

Additional system threads may also be spawned, depending on the kernel and board configuration options specified by the application. For example, enabling the system workqueue spawns a system thread that services the work items submitted to it. (See *Workqueue Threads*.)

Implementation

Writing a main() function An application-supplied `main()` function begins executing once kernel initialization is complete. The kernel does not pass any arguments to the function.

The following code outlines a trivial `main()` function. The function used by a real application can be as complex as needed.

```
int main(void)
{
    /* initialize a semaphore */
    ...

    /* register an ISR that gives the semaphore */
    ...

    /* monitor the semaphore forever */
    while (1) {
        /* wait for the semaphore to be given by the ISR */
        ...
        /* do whatever processing is now needed */
        ...
    }
}
```

Suggested Uses Use the main thread to perform thread-based processing in an application that only requires a single thread, rather than defining an additional application-specific thread.

Workqueue Threads

- *Work Item Lifecycle*
- *Delayable Work*

- *Triggered Work*
- *System Workqueue*
- *How to Use Workqueues*
- *Workqueue Best Practices*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

A *workqueue* is a kernel object that uses a dedicated thread to process work items in a first in, first out manner. Each work item is processed by calling the function specified by the work item. A workqueue is typically used by an ISR or a high-priority thread to offload non-urgent processing to a lower-priority thread so it does not impact time-sensitive processing.

Any number of workqueues can be defined (limited only by available RAM). Each workqueue is referenced by its memory address.

A workqueue has the following key properties:

- A **queue** of work items that have been added, but not yet processed.
- A **thread** that processes the work items in the queue. The priority of the thread is configurable, allowing it to be either cooperative or preemptive as required.

Regardless of workqueue thread priority the workqueue thread will yield between each submitted work item, to prevent a cooperative workqueue from starving other threads.

A workqueue must be initialized before it can be used. This sets its queue to empty and spawns the workqueue's thread. The thread runs forever, but sleeps when no work items are available.

Note

The behavior described here is changed from the Zephyr workqueue implementation used prior to release 2.6. Among the changes are:

- Precise tracking of the status of cancelled work items, so that the caller need not be concerned that an item may be processing when the cancellation returns. Checking of return values on cancellation is still required.
- Direct submission of delayable work items to the queue with `K_NO_WAIT` rather than always going through the timeout API, which could introduce delays.
- The ability to wait until a work item has completed or a queue has been drained.
- Finer control of behavior when scheduling a delayable work item, specifically allowing a previous deadline to remain unchanged when a work item is scheduled again.
- Safe handling of work item resubmission when the item is being processed on another workqueue.

Using the return values of `k_work_busy_get()` or `k_work_is_pending()`, or measurements of remaining time until delayable work is scheduled, should be avoided to prevent race conditions of the type observed with the previous implementation. See also *Workqueue Best Practices*.

Work Item Lifecycle Any number of **work items** can be defined. Each work item is referenced by its memory address.

A work item is assigned a **handler function**, which is the function executed by the workqueue's thread when the work item is processed. This function accepts a single argument, which is the address of the work item itself. The work item also maintains information about its status.

A work item must be initialized before it can be used. This records the work item's handler function and marks it as not pending.

A work item may be **queued** (`K_WORK_QUEUED`) by submitting it to a workqueue by an ISR or a thread. Submitting a work item appends the work item to the workqueue's queue. Once the workqueue's thread has processed all of the preceding work items in its queue the thread will remove the next work item from the queue and invoke the work item's handler function. Depending on the scheduling priority of the workqueue's thread, and the work required by other items in the queue, a queued work item may be processed quickly or it may remain in the queue for an extended period of time.

A delayable work item may be **scheduled** (`K_WORK_DELAYED`) to a workqueue; see *Delayable Work*.

A work item will be **running** (`K_WORK_RUNNING`) when it is running on a work queue, and may also be **canceling** (`K_WORK_CANCELING`) if it started running before a thread has requested that it be cancelled.

A work item can be in multiple states; for example it can be:

- running on a queue;
- marked canceling (because a thread used *k_work_cancel_sync()* to wait until the work item completed);
- queued to run again on the same queue;
- scheduled to be submitted to a (possibly different) queue

all simultaneously. A work item that is in any of these states is **pending** (*k_work_is_pending()*) or **busy** (*k_work_busy_get()*).

A handler function can use any kernel API available to threads. However, operations that are potentially blocking (e.g. taking a semaphore) must be used with care, since the workqueue cannot process subsequent work items in its queue until the handler function finishes executing.

The single argument that is passed to a handler function can be ignored if it is not required. If the handler function requires additional information about the work it is to perform, the work item can be embedded in a larger data structure. The handler function can then use the argument value to compute the address of the enclosing data structure with *CONTAINER_OF*, and thereby obtain access to the additional information it needs.

A work item is typically initialized once and then submitted to a specific workqueue whenever work needs to be performed. If an ISR or a thread attempts to submit a work item that is already queued the work item is not affected; the work item remains in its current place in the workqueue's queue, and the work is only performed once.

A handler function is permitted to re-submit its work item argument to the workqueue, since the work item is no longer queued at that time. This allows the handler to execute work in stages, without unduly delaying the processing of other work items in the workqueue's queue.

Important

A pending work item *must not* be altered until the item has been processed by the workqueue thread. This means a work item must not be re-initialized while it is busy. Furthermore, any additional information the work item's handler function needs to perform its work must not be altered until the handler function has finished executing.

Delayable Work An ISR or a thread may need to schedule a work item that is to be processed only after a specified period of time, rather than immediately. This can be done by **scheduling**

a **delayable work item** to be submitted to a workqueue at a future time.

A delayable work item contains a standard work item but adds fields that record when and where the item should be submitted.

A delayable work item is initialized and scheduled to a workqueue in a similar manner to a standard work item, although different kernel APIs are used. When the schedule request is made the kernel initiates a timeout mechanism that is triggered after the specified delay has elapsed. Once the timeout occurs the kernel submits the work item to the specified workqueue, where it remains queued until it is processed in the standard manner.

Note that work handler used for delayable still receives a pointer to the underlying non-delayable work structure, which is not publicly accessible from *k_work_delayable*. To get access to an object that contains the delayable work object use this idiom:

```
static void work_handler(struct k_work *work)
{
        struct k_work_delayable *dwork = k_work_delayable_from_work(work);
        struct work_context *ctx = CONTAINER_OF(dwork, struct work_context,
                                                timed_work);
        ...
```

Triggered Work The *k_work_poll_submit()* interface schedules a triggered work item in response to a **poll event** (see *Polling API*), that will call a user-defined function when a monitored resource becomes available or poll signal is raised, or a timeout occurs. In contrast to *k_poll()*, the triggered work does not require a dedicated thread waiting or actively polling for a poll event.

A triggered work item is a standard work item that has the following added properties:

- A pointer to an array of poll events that will trigger work item submissions to the workqueue
- A size of the array containing poll events.

A triggered work item is initialized and submitted to a workqueue in a similar manner to a standard work item, although dedicated kernel APIs are used. When a submit request is made, the kernel begins observing kernel objects specified by the poll events. Once at least one of the observed kernel object's changes state, the work item is submitted to the specified workqueue, where it remains queued until it is processed in the standard manner.

Important

The triggered work item as well as the referenced array of poll events have to be valid and cannot be modified for a complete triggered work item lifecycle, from submission to work item execution or cancellation.

An ISR or a thread may **cancel** a triggered work item it has submitted as long as it is still waiting for a poll event. In such case, the kernel stops waiting for attached poll events and the specified work is not executed. Otherwise the cancellation cannot be performed.

System Workqueue The kernel defines a workqueue known as the *system workqueue*, which is available to any application or kernel code that requires workqueue support. The system workqueue is optional, and only exists if the application makes use of it.

Important

Additional workqueues should only be defined when it is not possible to submit new work items to the system workqueue, since each new workqueue incurs a significant cost in memory footprint. A new workqueue can be justified if it is not possible for its work items to co-exist with existing system workqueue work items without an unacceptable impact; for example, if the new work items perform blocking operations that would delay other system workqueue processing to an unacceptable degree.

How to Use Workqueues

Defining and Controlling a Workqueue A workqueue is defined using a variable of type *k_work_q*. The workqueue is initialized by defining the stack area used by its thread, initializing the *k_work_q*, either zeroing its memory or calling *k_work_queue_init()*, and then calling *k_work_queue_start()*. The stack area must be defined using *K_THREAD_STACK_DEFINE* to ensure it is properly set up in memory.

The following code defines and initializes a workqueue:

```
#define MY_STACK_SIZE 512
#define MY_PRIORITY 5

K_THREAD_STACK_DEFINE(my_stack_area, MY_STACK_SIZE);

struct k_work_q my_work_q;

k_work_queue_init(&my_work_q);

k_work_queue_start(&my_work_q, my_stack_area,
                   K_THREAD_STACK_SIZEOF(my_stack_area), MY_PRIORITY,
                   NULL);
```

In addition the queue identity and certain behavior related to thread rescheduling can be controlled by the optional final parameter; see *k_work_queue_start()* for details.

The following API can be used to interact with a workqueue:

- *k_work_queue_drain()* can be used to block the caller until the work queue has no items left. Work items resubmitted from the workqueue thread are accepted while a queue is draining, but work items from any other thread or ISR are rejected. The restriction on submitting more work can be extended past the completion of the drain operation in order to allow the blocking thread to perform additional work while the queue is "plugged". Note that draining a queue has no effect on scheduling or processing delayable items, but if the queue is plugged and the deadline expires the item will silently fail to be submitted.
- *k_work_queue_unplug()* removes any previous block on submission to the queue due to a previous drain operation.

Submitting a Work Item A work item is defined using a variable of type *k_work*. It must be initialized by calling *k_work_init()*, unless it is defined using *K_WORK_DEFINE* in which case initialization is performed at compile-time.

An initialized work item can be submitted to the system workqueue by calling *k_work_submit()*, or to a specified workqueue by calling *k_work_submit_to_queue()*.

The following code demonstrates how an ISR can offload the printing of error messages to the system workqueue. Note that if the ISR attempts to resubmit the work item while it is still queued, the work item is left unchanged and the associated error message will not be printed.

```
struct device_info {
    struct k_work work;
    char name[16]
} my_device;

void my_isr(void *arg)
{
    ...
    if (error detected) {
        k_work_submit(&my_device.work);
    }
    ...
}

void print_error(struct k_work *item)
{
    struct device_info *the_device =
        CONTAINER_OF(item, struct device_info, work);
    printk("Got error on device %s\n", the_device->name);
}

/* initialize name info for a device */
strcpy(my_device.name, "FOO_dev");

/* initialize work item for printing device's error messages */
k_work_init(&my_device.work, print_error);

/* install my_isr() as interrupt handler for the device (not shown) */
...
```

The following API can be used to check the status of or synchronize with the work item:

- *k_work_busy_get()* returns a snapshot of flags indicating work item state. A zero value indicates the work is not scheduled, submitted, being executed, or otherwise still being referenced by the workqueue infrastructure.
- *k_work_is_pending()* is a helper that indicates `true` if and only if the work is scheduled, queued, or running.
- *k_work_flush()* may be invoked from threads to block until the work item has completed. It returns immediately if the work is not pending.
- *k_work_cancel()* attempts to prevent the work item from being executed. This may or may not be successful. This is safe to invoke from ISRs.
- *k_work_cancel_sync()* may be invoked from threads to block until the work completes; it will return immediately if the cancellation was successful or not necessary (the work wasn't submitted or running). This can be used after *k_work_cancel()* is invoked (from an ISR)' to confirm completion of an ISR-initiated cancellation.

Scheduling a Delayable Work Item A delayable work item is defined using a variable of type *k_work_delayable*. It must be initialized by calling *k_work_init_delayable()*.

For delayed work there are two common use cases, depending on whether a deadline should be extended if a new event occurs. An example is collecting data that comes in asynchronously, e.g. characters from a UART associated with a keyboard. There are two APIs that submit work after a delay:

- *k_work_schedule()* (or *k_work_schedule_for_queue()*) schedules work to be executed at a specific time or after a delay. Further attempts to schedule the same item with this API before the delay completes will not change the time at which the item will be submitted to its queue. Use this if the policy is to keep collecting data until a specified delay since the **first** unprocessed data was received;

- *k_work_reschedule()* (or *k_work_reschedule_for_queue()*) unconditionally sets the deadline for the work, replacing any previous incomplete delay and changing the destination queue if necessary. Use this if the policy is to keep collecting data until a specified delay since the **last** unprocessed data was received.

If the work item is not scheduled both APIs behave the same. If *K_NO_WAIT* is specified as the delay the behavior is as if the item was immediately submitted directly to the target queue, without waiting for a minimal timeout (unless *k_work_schedule()* is used and a previous delay has not completed).

Both also have variants that allow control of the queue used for submission.

The helper function *k_work_delayable_from_work()* can be used to get a pointer to the containing *k_work_delayable* from a pointer to *k_work* that is passed to a work handler function.

The following additional API can be used to check the status of or synchronize with the work item:

- *k_work_delayable_busy_get()* is the analog to *k_work_busy_get()* for delayable work.
- *k_work_delayable_is_pending()* is the analog to *k_work_is_pending()* for delayable work.
- *k_work_flush_delayable()* is the analog to *k_work_flush()* for delayable work.
- *k_work_cancel_delayable()* is the analog to *k_work_cancel()* for delayable work; similarly with *k_work_cancel_delayable_sync()*.

Synchronizing with Work Items While the state of both regular and delayable work items can be determined from any context using *k_work_busy_get()* and *k_work_delayable_busy_get()* some use cases require synchronizing with work items after they've been submitted. *k_work_flush()*, *k_work_cancel_sync()*, and *k_work_cancel_delayable_sync()* can be invoked from thread context to wait until the requested state has been reached.

These APIs must be provided with a *k_work_sync* object that has no application-inspectable components but is needed to provide the synchronization objects. These objects should not be allocated on a stack if the code is expected to work on architectures with `CONFIG_KERNEL_COHERENCE`.

Workqueue Best Practices

Avoid Race Conditions Sometimes the data a work item must process is naturally thread-safe, for example when it's put into a `k_queue` by some thread and processed in the work thread. More often external synchronization is required to avoid data races: cases where the work thread might inspect or manipulate shared state that's being accessed by another thread or interrupt. Such state might be a flag indicating that work needs to be done, or a shared object that is filled by an ISR or thread and read by the work handler.

For simple flags *Atomic Services* may be sufficient. In other cases spin locks (*k_spinlock*) or thread-aware locks (`k_sem`, *k_mutex* , ...) may be used to ensure data races don't occur.

If the selected lock mechanism can *sleep* then allowing the work thread to sleep will starve other work queue items, which may need to make progress in order to get the lock released. Work handlers should try to take the lock with its no-wait path. For example:

```
static void work_handler(struct work *work)
{
        struct work_context *parent = CONTAINER_OF(work, struct work_context,
                                                   work_item);

        if (k_mutex_lock(&parent->lock, K_NO_WAIT) != 0) {
                /* NB: Submit will fail if the work item is being cancelled. */
                (void)k_work_submit(work);
```

(continues on next page)

(continued from previous page)

```
                return;
        }

        /* do stuff under lock */
        k_mutex_unlock(&parent->lock);
        /* do stuff without lock */
}
```

Be aware that if the lock is held by a thread with a lower priority than the work queue the resubmission may starve the thread that would release the lock, causing the application to fail. Where the idiom above is required a delayable work item is preferred, and the work should be (re-)scheduled with a non-zero delay to allow the thread holding the lock to make progress.

Note that submitting from the work handler can fail if the work item had been cancelled. Generally this is acceptable, since the cancellation will complete once the handler finishes. If it is not, the code above must take other steps to notify the application that the work could not be performed.

Work items in isolation are self-locking, so you don't need to hold an external lock just to submit or schedule them. Even if you use external state protected by such a lock to prevent further resubmission, it's safe to do the resubmit as long as you're sure that eventually the item will take its lock and check that state to determine whether it should do anything. Where a delayable work item is being rescheduled in its handler due to inability to take the lock some other self-locking state, such as an atomic flag set by the application/driver when the cancel is initiated, would be required to detect the cancellation and avoid the cancelled work item being submitted again after the deadline.

Check Return Values All work API functions return status of the underlying operation, and in many cases it is important to verify that the intended result was obtained.

- Submitting a work item (*k_work_submit_to_queue()*) can fail if the work is being cancelled or the queue is not accepting new items. If this happens the work will not be executed, which could cause a subsystem that is animated by work handler activity to become non-responsive.
- Asynchronous cancellation (*k_work_cancel()* or *k_work_cancel_delayable()*) can complete while the work item is still being run by a handler. Proceeding to manipulate state shared with the work handler will result in data races that can cause failures.

Many race conditions have been present in Zephyr code because the results of an operation were not checked.

There may be good reason to believe that a return value indicating that the operation did not complete as expected is not a problem. In those cases the code should clearly document this, by (1) casting the return value to `void` to indicate that the result is intentionally ignored, and (2) documenting what happens in the unexpected case. For example:

```
/* If this fails, the work handler will check pub->active and
 * exit without transmitting.
 */
(void)k_work_cancel_delayable(&pub->timer);
```

However in such a case the following code must still avoid data races, as it cannot guarantee that the work thread is not accessing work-related state.

Don't Optimize Prematurely The workqueue API is designed to be safe when invoked from multiple threads and interrupts. Attempts to externally inspect a work item's state and make decisions based on the result are likely to create new problems.

So when new work comes in, just submit it. Don't attempt to "optimize" by checking whether the work item is already submitted by inspecting snapshot state with *k_work_is_pending()* or *k_work_busy_get()*, or checking for a non-zero delay from *k_work_delayable_remaining_get()*. Those checks are fragile: a "busy" indication can be obsolete by the time the test is returned, and a "not-busy" indication can also be wrong if work is submitted from multiple contexts, or (for delayable work) if the deadline has completed but the work is still in queued or running state.

A general best practice is to always maintain in shared state some condition that can be checked by the handler to confirm whether there is work to be done. This way you can use the work handler as the standard cleanup path: rather than having to deal with cancellation and cleanup at points where items are submitted, you may be able to have everything done in the work handler itself.

A rare case where you could safely use *k_work_is_pending()* is as a check to avoid invoking *k_work_flush()* or *k_work_cancel_sync()*, if you are *certain* that nothing else might submit the work while you're checking (generally because you're holding a lock that prevents access to state used for submission).

Suggested Uses Use the system workqueue to defer complex interrupt-related processing from an ISR to a shared thread. This allows the interrupt-related processing to be done promptly without compromising the system's ability to respond to subsequent interrupts, and does not require the application to define and manage an additional thread to do the processing.

Configuration Options Related configuration options:

- CONFIG_SYSTEM_WORKQUEUE_STACK_SIZE
- CONFIG_SYSTEM_WORKQUEUE_PRIORITY
- CONFIG_SYSTEM_WORKQUEUE_NO_YIELD

API Reference

group workqueue_apis

Defines

K_WORK_DELAYABLE_DEFINE(work, work_handler)

Initialize a statically-defined delayable work item.

This macro can be used to initialize a statically-defined delayable work item, prior to its first use. For example,

```
static K_WORK_DELAYABLE_DEFINE(<dwork>, <work_handler>);
```

Note that if the runtime dependencies support initialization with *k_work_init_delayable()* using that will eliminate the initialized object in ROM that is produced by this macro and copied in at system startup.

Parameters

- **work** – Symbol name for delayable work item object
- **work_handler** – Function to invoke each time work item is processed.

K_WORK_USER_DEFINE(work, work_handler)

Initialize a statically-defined user work item.

This macro can be used to initialize a statically-defined user work item, prior to its first use. For example,

```
static K_WORK_USER_DEFINE(<work>, <work_handler>);
```

Parameters

- **work** – Symbol name for work item object
- **work_handler** – Function to invoke each time work item is processed.

K_WORK_DEFINE(work, work_handler)

Initialize a statically-defined work item.

This macro can be used to initialize a statically-defined workqueue work item, prior to its first use. For example,

```
static K_WORK_DEFINE(<work>, <work_handler>);
```

Parameters

- **work** – Symbol name for work item object
- **work_handler** – Function to invoke each time work item is processed.

Typedefs

typedef void (*k_work_handler_t)(struct *k_work* *work)

The signature for a work item handler function.

The function will be invoked by the thread animating a work queue.

Param work
the work item that provided the handler.

typedef void (*k_work_user_handler_t)(struct k_work_user *work)

Work item handler function type for user work queues.

A work item's handler function is executed by a user workqueue's thread when the work item is processed by the workqueue.

Param work
Address of the work item.

Enums

Values:

enumerator K_WORK_RUNNING = *BIT*(K_WORK_RUNNING_BIT)

Flag indicating a work item that is running under a work queue thread.

Accessed via *k_work_busy_get()*. May co-occur with other flags.

enumerator K_WORK_CANCELING = *BIT*(K_WORK_CANCELING_BIT)

Flag indicating a work item that is being canceled.

```
   Accessed via k_work_busy_get().  May co-occur with other flags.
```

enumerator K_WORK_QUEUED = *BIT*(K_WORK_QUEUED_BIT)

Flag indicating a work item that has been submitted to a queue but has not started running.

Accessed via *k_work_busy_get()*. May co-occur with other flags.

enumerator K_WORK_DELAYED = *BIT*(K_WORK_DELAYED_BIT)

Flag indicating a delayed work item that is scheduled for submission to a queue.

Accessed via *k_work_busy_get()*. May co-occur with other flags.

enumerator K_WORK_FLUSHING = *BIT*(K_WORK_FLUSHING_BIT)

Flag indicating a synced work item that is being flushed.

```
Accessed via k_work_busy_get().  May co-occur with other flags.
```

Functions

void **k_work_init**(struct *k_work* *work, *k_work_handler_t* handler)

Initialize a (non-delayable) work structure.

This must be invoked before submitting a work structure for the first time. It need not be invoked again on the same work structure. It can be re-invoked to change the associated handler, but this must be done when the work item is idle.

Function properties (list may not be complete)

isr-ok

Parameters

- **work** – the work structure to be initialized.
- **handler** – the handler to be invoked by the work item.

int **k_work_busy_get**(const struct *k_work* *work)

Busy state flags from the work item.

A zero return value indicates the work item appears to be idle.

Function properties (list may not be complete)

isr-ok

Note

This is a live snapshot of state, which may change before the result is checked. Use locks where appropriate.

Parameters

- **work** – pointer to the work item.

Returns

a mask of flags K_WORK_DELAYED, K_WORK_QUEUED, K_WORK_RUNNING, K_WORK_CANCELING, and K_WORK_FLUSHING.

static inline bool **k_work_is_pending**(const struct *k_work* *work)

Test whether a work item is currently pending.

Wrapper to determine whether a work item is in a non-idle dstate.

Function properties (list may not be complete)

isr-ok

> **Note**
>
> This is a live snapshot of state, which may change before the result is checked. Use locks where appropriate.

Parameters

- **work** – pointer to the work item.

Returns

true if and only if *k_work_busy_get()* returns a non-zero value.

int **k_work_submit_to_queue**(struct *k_work_q* *queue, struct *k_work* *work)

Submit a work item to a queue.

Function properties (list may not be complete)

isr-ok

Parameters

- **queue** – pointer to the work queue on which the item should run. If NULL the queue from the most recent submission will be used.
- **work** – pointer to the work item.

Return values

- **0** – if work was already submitted to a queue
- **1** – if work was not submitted and has been queued to queue
- **2** – if work was running and has been queued to the queue that was running it
- **-EBUSY** –
  - if work submission was rejected because the work item is cancelling; or
  - queue is draining; or
  - queue is plugged.
- **-EINVAL** – if queue is null and the work item has never been run.
- **-ENODEV** – if queue has not been started.

int **k_work_submit**(struct *k_work* *work)

Submit a work item to the system queue.

Function properties (list may not be complete)

isr-ok

Parameters

- `work` – pointer to the work item.

Returns
as with *k_work_submit_to_queue()*.

bool `k_work_flush`(struct *k_work* *work, struct *k_work_sync* *sync)

Wait for last-submitted instance to complete.

Resubmissions may occur while waiting, including chained submissions (from within the handler).

Note

Be careful of caller and work queue thread relative priority. If this function sleeps it will not return until the work queue thread completes the tasks that allow this thread to resume.

Note

Behavior is undefined if this function is invoked on `work` from a work queue running `work`.

Parameters

- `work` – pointer to the work item.
- `sync` – pointer to an opaque item containing state related to the pending cancellation. The object must persist until the call returns, and be accessible from both the caller thread and the work queue thread. The object must not be used for any other flush or cancel operation until this one completes. On architectures with CONFIG_KERNEL_COHERENCE the object must be allocated in coherent memory.

Return values

- `true` – if call had to wait for completion
- `false` – if work was already idle

int `k_work_cancel`(struct *k_work* *work)

Cancel a work item.

This attempts to prevent a pending (non-delayable) work item from being processed by removing it from the work queue. If the item is being processed, the work item will continue to be processed, but resubmissions are rejected until cancellation completes.

If this returns zero cancellation is complete, otherwise something (probably a work queue thread) is still referencing the item.

See also *k_work_cancel_sync()*.

Function properties (list may not be complete)
isr-ok

Parameters

- `work` – pointer to the work item.

Returns
the *k_work_busy_get()* status indicating the state of the item after all cancellation steps performed by this call are completed.

bool **k_work_cancel_sync**(struct *k_work* *work, struct *k_work_sync* *sync)

Cancel a work item and wait for it to complete.

Same as *k_work_cancel()* but does not return until cancellation is complete. This can be invoked by a thread after *k_work_cancel()* to synchronize with a previous cancellation.

On return the work structure will be idle unless something submits it after the cancellation was complete.

Note

Be careful of caller and work queue thread relative priority. If this function sleeps it will not return until the work queue thread completes the tasks that allow this thread to resume.

Note

Behavior is undefined if this function is invoked on `work` from a work queue running `work`.

Parameters

- **work** – pointer to the work item.
- **sync** – pointer to an opaque item containing state related to the pending cancellation. The object must persist until the call returns, and be accessible from both the caller thread and the work queue thread. The object must not be used for any other flush or cancel operation until this one completes. On architectures with CONFIG_KERNEL_COHERENCE the object must be allocated in coherent memory.

Return values

- **true** – if work was pending (call had to wait for cancellation of a running handler to complete, or scheduled or submitted operations were cancelled);
- **false** – otherwise

void **k_work_queue_init**(struct *k_work_q* *queue)

Initialize a work queue structure.

This must be invoked before starting a work queue structure for the first time. It need not be invoked again on the same work queue structure.

Function properties (list may not be complete)
isr-ok

Parameters

- **queue** – the queue structure to be initialized.

void **k_work_queue_start**(struct *k_work_q* *queue, k_thread_stack_t *stack, size_t stack_size, int prio, const struct *k_work_queue_config* *cfg)

Initialize a work queue.

This configures the work queue thread and starts it running. The function should not be re-invoked on a queue.

Parameters

- **queue** – pointer to the queue structure. It must be initialized in zeroed/bss memory or with *k_work_queue_init* before use.
- **stack** – pointer to the work thread stack area.
- **stack_size** – size of the work thread stack area, in bytes.
- **prio** – initial thread priority
- **cfg** – optional additional configuration parameters. Pass `NULL` if not required, to use the defaults documented in *k_work_queue_config*.

static inline k_tid_t **k_work_queue_thread_get**(struct *k_work_q* *queue)

Access the thread that animates a work queue.

This is necessary to grant a work queue thread access to things the work items it will process are expected to use.

Parameters

- **queue** – pointer to the queue structure.

Returns

the thread associated with the work queue.

int **k_work_queue_drain**(struct *k_work_q* *queue, bool plug)

Wait until the work queue has drained, optionally plugging it.

This blocks submission to the work queue except when coming from queue thread, and blocks the caller until no more work items are available in the queue.

If `plug` is true then submission will continue to be blocked after the drain operation completes until *k_work_queue_unplug()* is invoked.

Note that work items that are delayed are not yet associated with their work queue. They must be cancelled externally if a goal is to ensure the work queue remains empty. The `plug` feature can be used to prevent delayed items from being submitted after the drain completes.

Parameters

- **queue** – pointer to the queue structure.
- **plug** – if true the work queue will continue to block new submissions after all items have drained.

Return values

- **1** – if call had to wait for the drain to complete
- **0** – if call did not have to wait
- **negative** – if wait was interrupted or failed

int **k_work_queue_unplug**(struct *k_work_q* *queue)

Release a work queue to accept new submissions.

This releases the block on new submissions placed when *k_work_queue_drain()* is invoked with the `plug` option enabled. If this is invoked before the drain completes new items may be submitted as soon as the drain completes.

Function properties (list may not be complete)
isr-ok

Parameters

- `queue` – pointer to the queue structure.

Return values

- `0` – if successfully unplugged
- `-EALREADY` – if the work queue was not plugged.

void `k_work_init_delayable`(struct *k_work_delayable* *dwork, *k_work_handler_t* handler)

Initialize a delayable work structure.

This must be invoked before scheduling a delayable work structure for the first time. It need not be invoked again on the same work structure. It can be re-invoked to change the associated handler, but this must be done when the work item is idle.

Function properties (list may not be complete)
isr-ok

Parameters

- `dwork` – the delayable work structure to be initialized.
- `handler` – the handler to be invoked by the work item.

static inline struct *k_work_delayable* *`k_work_delayable_from_work`(struct *k_work* *work)

Get the parent delayable work structure from a work pointer.

This function is necessary when a `k_work_handler_t` function is passed to *k_work_schedule_for_queue()* and the handler needs to access data from the container of the containing *`k_work_delayable`*.

Parameters

- `work` – Address passed to the work handler

Returns
Address of the containing *`k_work_delayable`* structure.

int `k_work_delayable_busy_get`(const struct *k_work_delayable* *dwork)

Busy state flags from the delayable work item.

Function properties (list may not be complete)
isr-ok

Note

This is a live snapshot of state, which may change before the result can be inspected. Use locks where appropriate.

Parameters

- `dwork` – pointer to the delayable work item.

Returns
a mask of flags K_WORK_DELAYED, K_WORK_QUEUED, K_WORK_RUNNING, K_WORK_CANCELING, and K_WORK_FLUSHING. A zero return value indicates the work item appears to be idle.

static inline bool **k_work_delayable_is_pending**(const struct *k_work_delayable* *dwork)

Test whether a delayed work item is currently pending.

Wrapper to determine whether a delayed work item is in a non-idle state.

Function properties (list may not be complete)
isr-ok

> **Note**
> This is a live snapshot of state, which may change before the result can be inspected. Use locks where appropriate.

Parameters
- **dwork** – pointer to the delayable work item.

Returns
true if and only if *k_work_delayable_busy_get()* returns a non-zero value.

static inline *k_ticks_t* **k_work_delayable_expires_get**(const struct *k_work_delayable* *dwork)

Get the absolute tick count at which a scheduled delayable work will be submitted.

Function properties (list may not be complete)
isr-ok

> **Note**
> This is a live snapshot of state, which may change before the result can be inspected. Use locks where appropriate.

Parameters
- **dwork** – pointer to the delayable work item.

Returns
the tick count when the timer that will schedule the work item will expire, or the current tick count if the work is not scheduled.

static inline *k_ticks_t* **k_work_delayable_remaining_get**(const struct *k_work_delayable* *dwork)

Get the number of ticks until a scheduled delayable work will be submitted.

Function properties (list may not be complete)
isr-ok

Note

This is a live snapshot of state, which may change before the result can be inspected. Use locks where appropriate.

Parameters

- `dwork` – pointer to the delayable work item.

Returns

the number of ticks until the timer that will schedule the work item will expire, or zero if the item is not scheduled.

int `k_work_schedule_for_queue`(struct *k_work_q* *queue, struct *k_work_delayable* *dwork, *k_timeout_t* delay)

Submit an idle work item to a queue after a delay.

Unlike *k_work_reschedule_for_queue()* this is a no-op if the work item is already scheduled or submitted, even if `delay` is `K_NO_WAIT`.

Function properties (list may not be complete)

isr-ok

Parameters

- `queue` – the queue on which the work item should be submitted after the delay.
- `dwork` – pointer to the delayable work item.
- `delay` – the time to wait before submitting the work item. If `K_NO_WAIT` and the work is not pending this is equivalent to *k_work_submit_to_queue()*.

Return values

- `0` – if work was already scheduled or submitted.
- `1` – if work has been scheduled.
- `2` – if `delay` is `K_NO_WAIT` and work was running and has been queued to the queue that was running it.
- `-EBUSY` – if `delay` is `K_NO_WAIT` and *k_work_submit_to_queue()* fails with this code.
- `-EINVAL` – if `delay` is `K_NO_WAIT` and *k_work_submit_to_queue()* fails with this code.
- `-ENODEV` – if `delay` is `K_NO_WAIT` and *k_work_submit_to_queue()* fails with this code.

int `k_work_schedule`(struct *k_work_delayable* *dwork, *k_timeout_t* delay)

Submit an idle work item to the system work queue after a delay.

This is a thin wrapper around *k_work_schedule_for_queue()*, with all the API characteristics of that function.

Parameters

- `dwork` – pointer to the delayable work item.
- `delay` – the time to wait before submitting the work item. If `K_NO_WAIT` this is equivalent to *k_work_submit_to_queue()*.

Returns
as with *k_work_schedule_for_queue()*.

int **k_work_reschedule_for_queue**(struct *k_work_q* *queue, struct *k_work_delayable* *dwork, *k_timeout_t* delay)

Reschedule a work item to a queue after a delay.

Unlike *k_work_schedule_for_queue()* this function can change the deadline of a scheduled work item, and will schedule a work item that is in any state (e.g. is idle, submitted, or running). This function does not affect ("unsubmit") a work item that has been submitted to a queue.

Function properties (list may not be complete)
isr-ok

Note

If delay is `K_NO_WAIT` ("no delay") the return values are as with *k_work_submit_to_queue()*.

Parameters

- **queue** – the queue on which the work item should be submitted after the delay.
- **dwork** – pointer to the delayable work item.
- **delay** – the time to wait before submitting the work item. If `K_NO_WAIT` this is equivalent to *k_work_submit_to_queue()* after canceling any previous scheduled submission.

Return values

- **0** – if delay is `K_NO_WAIT` and work was already on a queue
- **1** – if
  - delay is `K_NO_WAIT` and work was not submitted but has now been queued to queue; or
  - delay not `K_NO_WAIT` and work has been scheduled
- **2** – if delay is `K_NO_WAIT` and work was running and has been queued to the queue that was running it
- **-EBUSY** – if `delay` is `K_NO_WAIT` and *k_work_submit_to_queue()* fails with this code.
- **-EINVAL** – if `delay` is `K_NO_WAIT` and *k_work_submit_to_queue()* fails with this code.
- **-ENODEV** – if `delay` is `K_NO_WAIT` and *k_work_submit_to_queue()* fails with this code.

int **k_work_reschedule**(struct *k_work_delayable* *dwork, *k_timeout_t* delay)

Reschedule a work item to the system work queue after a delay.

This is a thin wrapper around *k_work_reschedule_for_queue()*, with all the API characteristics of that function.

Parameters

- **dwork** – pointer to the delayable work item.

- `delay` – the time to wait before submitting the work item.

Returns
as with *k_work_reschedule_for_queue()*.

bool `k_work_flush_delayable`(struct *k_work_delayable* *dwork, struct *k_work_sync* *sync)

Flush delayable work.

If the work is scheduled, it is immediately submitted. Then the caller blocks until the work completes, as with *k_work_flush()*.

Note

Be careful of caller and work queue thread relative priority. If this function sleeps it will not return until the work queue thread completes the tasks that allow this thread to resume.

Note

Behavior is undefined if this function is invoked on `dwork` from a work queue running `dwork`.

Parameters

- `dwork` – pointer to the delayable work item.
- `sync` – pointer to an opaque item containing state related to the pending cancellation. The object must persist until the call returns, and be accessible from both the caller thread and the work queue thread. The object must not be used for any other flush or cancel operation until this one completes. On architectures with CONFIG_KERNEL_COHERENCE the object must be allocated in coherent memory.

Return values

- `true` – if call had to wait for completion
- `false` – if work was already idle

int `k_work_cancel_delayable`(struct *k_work_delayable* *dwork)

Cancel delayable work.

Similar to *k_work_cancel()* but for delayable work. If the work is scheduled or submitted it is canceled. This function does not wait for the cancellation to complete.

Function properties (list may not be complete)
isr-ok

Note

The work may still be running when this returns. Use *k_work_flush_delayable()* or *k_work_cancel_delayable_sync()* to ensure it is not running.

Note

Canceling delayable work does not prevent rescheduling it. It does prevent submitting it until the cancellation completes.

Parameters

- `dwork` – pointer to the delayable work item.

Returns

the *k_work_delayable_busy_get()* status indicating the state of the item after all cancellation steps performed by this call are completed.

bool `k_work_cancel_delayable_sync`(struct *k_work_delayable* *dwork, struct *k_work_sync* *sync)

Cancel delayable work and wait.

Like *k_work_cancel_delayable()* but waits until the work becomes idle.

Note

Canceling delayable work does not prevent rescheduling it. It does prevent submitting it until the cancellation completes.

Note

Be careful of caller and work queue thread relative priority. If this function sleeps it will not return until the work queue thread completes the tasks that allow this thread to resume.

Note

Behavior is undefined if this function is invoked on `dwork` from a work queue running `dwork`.

Parameters

- `dwork` – pointer to the delayable work item.
- `sync` – pointer to an opaque item containing state related to the pending cancellation. The object must persist until the call returns, and be accessible from both the caller thread and the work queue thread. The object must not be used for any other flush or cancel operation until this one completes. On architectures with CONFIG_KERNEL_COHERENCE the object must be allocated in coherent memory.

Return values

- `true` – if work was not idle (call had to wait for cancellation of a running handler to complete, or scheduled or submitted operations were cancelled);
- `false` – otherwise

static inline void **k_work_user_init**(struct k_work_user *work, *k_work_user_handler_t* handler)

Initialize a userspace work item.

This routine initializes a user workqueue work item, prior to its first use.

Parameters

- `work` – Address of work item.
- `handler` – Function to invoke each time work item is processed.

static inline bool **k_work_user_is_pending**(struct k_work_user *work)

Check if a userspace work item is pending.

This routine indicates if user work item *work* is pending in a workqueue's queue.

Function properties (list may not be complete)

isr-ok

Note

Checking if the work is pending gives no guarantee that the work will still be pending when this information is used. It is up to the caller to make sure that this information is used in a safe manner.

Parameters

- `work` – Address of work item.

Returns

true if work item is pending, or false if it is not pending.

static inline int **k_work_user_submit_to_queue**(struct k_work_user_q *work_q, struct k_work_user *work)

Submit a work item to a user mode workqueue.

Submits a work item to a workqueue that runs in user mode. A temporary memory allocation is made from the caller's resource pool which is freed once the worker thread consumes the *k_work* item. The workqueue thread must have memory access to the *k_work* item being submitted. The caller must have permission granted on the work_q parameter's queue object.

Function properties (list may not be complete)

isr-ok

Parameters

- `work_q` – Address of workqueue.
- `work` – Address of work item.

Return values

- `-EBUSY` – if the work item was already in some workqueue
- `-ENOMEM` – if no memory for thread resource pool allocation
- `0` – Success

void **k_work_user_queue_start**(struct k_work_user_q *work_q, k_thread_stack_t *stack, size_t stack_size, int prio, const char *name)

Start a workqueue in user mode.

This works identically to *k_work_queue_start()* except it is callable from user mode, and the worker thread created will run in user mode. The caller must have permissions granted on both the work_q parameter's thread and queue objects, and the same restrictions on priority apply as *k_thread_create()*.

Parameters

- **work_q** – Address of workqueue.
- **stack** – Pointer to work queue thread's stack space, as defined by *K_THREAD_STACK_DEFINE()*
- **stack_size** – Size of the work queue thread's stack (in bytes), which should either be the same constant passed to *K_THREAD_STACK_DEFINE()* or the value of *K_THREAD_STACK_SIZEOF()*.
- **prio** – Priority of the work queue's thread.
- **name** – optional thread name. If not null a copy is made into the thread's name buffer.

static inline k_tid_t **k_work_user_queue_thread_get**(struct k_work_user_q *work_q)

Access the user mode thread that animates a work queue.

This is necessary to grant a user mode work queue thread access to things the work items it will process are expected to use.

Parameters

- **work_q** – pointer to the user mode queue structure.

Returns
the user mode thread associated with the work queue.

void **k_work_poll_init**(struct k_work_poll *work, *k_work_handler_t* handler)

Initialize a triggered work item.

This routine initializes a workqueue triggered work item, prior to its first use.

Parameters

- **work** – Address of triggered work item.
- **handler** – Function to invoke each time work item is processed.

int **k_work_poll_submit_to_queue**(struct *k_work_q* *work_q, struct k_work_poll *work, struct *k_poll_event* *events, int num_events, *k_timeout_t* timeout)

Submit a triggered work item.

This routine schedules work item *work* to be processed by workqueue *work_q* when one of the given *events* is signaled. The routine initiates internal poller for the work item and then returns to the caller. Only when one of the watched events happen the work item is actually submitted to the workqueue and becomes pending.

Submitting a previously submitted triggered work item that is still waiting for the event cancels the existing submission and reschedules it the using the new event list. Note that this behavior is inherently subject to race conditions with the pre-existing triggered work item and work queue, so care must be taken to synchronize such resubmissions externally.

Function properties (list may not be complete)
isr-ok

Warning

Provided array of events as well as a triggered work item must be placed in persistent memory (valid until work handler execution or work cancellation) and cannot be modified after submission.

Parameters

- `work_q` – Address of workqueue.
- `work` – Address of delayed work item.
- `events` – An array of events which trigger the work.
- `num_events` – The number of events in the array.
- `timeout` – Timeout after which the work will be scheduled for execution even if not triggered.

Return values

- `0` – Work item started watching for events.
- `-EINVAL` – Work item is being processed or has completed its work.
- `-EADDRINUSE` – Work item is pending on a different workqueue.

int `k_work_poll_submit`(struct k_work_poll *work, struct *k_poll_event* *events, int num_events, *k_timeout_t* timeout)

Submit a triggered work item to the system workqueue.

This routine schedules work item *work* to be processed by system workqueue when one of the given *events* is signaled. The routine initiates internal poller for the work item and then returns to the caller. Only when one of the watched events happen the work item is actually submitted to the workqueue and becomes pending.

Submitting a previously submitted triggered work item that is still waiting for the event cancels the existing submission and reschedules it the using the new event list. Note that this behavior is inherently subject to race conditions with the pre-existing triggered work item and work queue, so care must be taken to synchronize such resubmissions externally.

Function properties (list may not be complete)
isr-ok

Warning

Provided array of events as well as a triggered work item must not be modified until the item has been processed by the workqueue.

Parameters

- `work` – Address of delayed work item.
- `events` – An array of events which trigger the work.
- `num_events` – The number of events in the array.

- `timeout` – Timeout after which the work will be scheduled for execution even if not triggered.

Return values

- `0` – Work item started watching for events.
- `-EINVAL` – Work item is being processed or has completed its work.
- `-EADDRINUSE` – Work item is pending on a different workqueue.

int `k_work_poll_cancel`(struct k_work_poll *work)

Cancel a triggered work item.

This routine cancels the submission of triggered work item *work*. A triggered work item can only be canceled if no event triggered work submission.

Function properties (list may not be complete)

isr-ok

Parameters

- `work` – Address of delayed work item.

Return values

- `0` – Work item canceled.
- `-EINVAL` – Work item is being processed or has completed its work.

struct `k_work`

#include <*kernel.h*> A structure used to submit work.

struct `k_work_delayable`

#include <*kernel.h*> A structure used to submit work after a delay.

struct `k_work_sync`

#include <*kernel.h*> A structure holding internal state for a pending synchronous operation on a work item or queue.

Instances of this type are provided by the caller for invocation of *k_work_flush()*, *k_work_cancel_sync()* and sibling flush and cancel APIs. A referenced object must persist until the call returns, and be accessible from both the caller thread and the work queue thread.

Note

If CONFIG_KERNEL_COHERENCE is enabled the object must be allocated in coherent memory; see *arch_mem_coherent()*. The stack on these architectures is generally not coherent. be stack-allocated. Violations are detected by runtime assertion.

struct `k_work_queue_config`

#include <*kernel.h*> A structure holding optional configuration items for a work queue.

This structure, and values it references, are not retained by *k_work_queue_start()*.

Public Members

const char *name

The name to be given to the work queue thread.

If left null the thread will not have a name.

bool no_yield

Control whether the work queue thread should yield between items.

Yielding between items helps guarantee the work queue thread does not starve other threads, including cooperative ones released by a work item. This is the default behavior.

Set this to true to prevent the work queue thread from yielding between items. This may be appropriate when a sequence of items should complete without yielding control.

bool essential

Control whether the work queue thread should be marked as essential thread.

struct k_work_q

#include <*kernel.h*> A structure used to hold work until it can be processed.

Operation without Threads

Thread support is not necessary in some applications:

- Bootloaders
- Simple event-driven applications
- Examples intended to demonstrate core functionality

Thread support can be disabled by setting CONFIG_MULTITHREADING to n. Since this configuration has a significant impact on Zephyr's functionality and testing of it has been limited, there are conditions on what can be expected to work in this configuration.

What Can be Expected to Work These core capabilities shall function correctly when CONFIG_MULTITHREADING is disabled:

- The *build system*
- The ability to boot the application to main()
- *Interrupt management*
- The system clock including *k_uptime_get()*
- Timers, i.e. k_timer()
- Non-sleeping delays e.g. *k_busy_wait()*.
- Sleeping *k_cpu_idle()*.
- Pre main() drivers and subsystems initialization e.g. SYS_INIT.
- *Memory Management*
- Specifically identified drivers in certain subsystems, listed below.

The expectations above affect selection of other features; for example CONFIG_SYS_CLOCK_EXISTS cannot be set to n.

What Cannot be Expected to Work Functionality that will not work with CONFIG_MULTITHREADING includes majority of the kernel API:

- *Threads*
- *Scheduling*
- *Workqueue Threads*
- *Polling API*
- *Semaphores*
- *Mutexes*
- *Condition Variables*
- *Data Passing*

Subsystem Behavior Without Thread Support The sections below list driver and functional subsystems that are expected to work to some degree when CONFIG_MULTITHREADING is disabled. Subsystems that are not listed here should not be expected to work.

Some existing drivers within the listed subsystems do not work when threading is disabled, but are within scope based on their subsystem, or may be sufficiently isolated that supporting them on a particular platform is low-impact. Enhancements to add support to existing capabilities that were not originally implemented to work with threads disabled will be considered.

Flash The *Flash* is expected to work for all SoC flash peripheral drivers. Bus-accessed devices like serial memories may not be supported.

List/table of supported drivers to go here

GPIO The *General-Purpose Input/Output (GPIO)* is expected to work for all SoC GPIO peripheral drivers. Bus-accessed devices like GPIO extenders may not be supported.

List/table of supported drivers to go here

UART A subset of the *Universal Asynchronous Receiver-Transmitter (UART)* is expected to work for all SoC UART peripheral drivers.

- Applications that select CONFIG_UART_INTERRUPT_DRIVEN may work, depending on driver implementation.
- Applications that select CONFIG_UART_ASYNC_API may work, depending on driver implementation.
- Applications that do not select either CONFIG_UART_ASYNC_API or CONFIG_UART_INTERRUPT_DRIVEN are expected to work.

List/table of supported drivers to go here, including which API options are supported

Interrupts

An *interrupt service routine* (ISR) is a function that executes asynchronously in response to a hardware or software interrupt. An ISR normally preempts the execution of the current thread, allowing the response to occur with very low overhead. Thread execution resumes only once all ISR work has been completed.

- *Concepts*
  - *Multi-level Interrupt Handling*
  - *Preventing Interruptions*
  - *Offloading ISR Work*
  - *Sharing interrupt lines*
- *Implementation*
  - *Defining a regular ISR*
  - *Defining a 'direct' ISR*
  - *Sharing an interrupt line*
  - *Dynamically disconnecting an ISR*
  - *Implementation Details*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

Concepts Any number of ISRs can be defined (limited only by available RAM), subject to the constraints imposed by underlying hardware.

An ISR has the following key properties:

- An **interrupt request (IRQ) signal** that triggers the ISR.
- A **priority level** associated with the IRQ.
- An **interrupt service routine** that is invoked to handle the interrupt.
- An **argument value** that is passed to that function.

An IDT (Interrupt Descriptor Table) or a vector table is used to associate a given interrupt source with a given ISR. Only a single ISR can be associated with a specific IRQ at any given time.

Multiple ISRs can utilize the same function to process interrupts, allowing a single function to service a device that generates multiple types of interrupts or to service multiple devices (usually of the same type). The argument value passed to an ISR's function allows the function to determine which interrupt has been signaled.

The kernel provides a default ISR for all unused IDT entries. This ISR generates a fatal system error if an unexpected interrupt is signaled.

The kernel supports **interrupt nesting**. This allows an ISR to be preempted in mid-execution if a higher priority interrupt is signaled. The lower priority ISR resumes execution once the higher priority ISR has completed its processing.

An ISR executes in the kernel's **interrupt context**. This context has its own dedicated stack area (or, on some architectures, stack areas). The size of the interrupt context stack must be capable of handling the execution of multiple concurrent ISRs if interrupt nesting support is enabled.

Important

Many kernel APIs can be used only by threads, and not by ISRs. In cases where a routine may be invoked by both threads and ISRs the kernel provides the `k_is_in_isr()` API to allow the

routine to alter its behavior depending on whether it is executing as part of a thread or as part of an ISR.

Multi-level Interrupt Handling A hardware platform can support more interrupt lines than natively-provided through the use of one or more nested interrupt controllers. Sources of hardware interrupts are combined into one line that is then routed to the parent controller.

If nested interrupt controllers are supported, `CONFIG_MULTI_LEVEL_INTERRUPTS` should be enabled, and `CONFIG_2ND_LEVEL_INTERRUPTS` and `CONFIG_3RD_LEVEL_INTERRUPTS` configured as well, based on the hardware architecture.

A unique 32-bit interrupt number is assigned with information embedded in it to select and invoke the correct Interrupt Service Routine (ISR). Each interrupt level is given a byte within this 32-bit number, providing support for up to four interrupt levels using this arch, as illustrated and explained below:

```
         9            2   0
  - - - - - - - - - - - - -        (LEVEL 1)
  5      |        A   |
- - - - - - -        - - - - - - -  (LEVEL 2)
  |   C                     B
- - - - - - -                      (LEVEL 3)
       D
```

There are three interrupt levels shown here.

- '-' means interrupt line and is numbered from 0 (right most).
- LEVEL 1 has 12 interrupt lines, with two lines (2 and 9) connected to nested controllers and one device 'A' on line 4.
- One of the LEVEL 2 controllers has interrupt line 5 connected to a LEVEL 3 nested controller and one device 'C' on line 3.
- The other LEVEL 2 controller has no nested controllers but has one device 'B' on line 2.
- The LEVEL 3 controller has one device 'D' on line 2.

Here's how unique interrupt numbers are generated for each hardware interrupt. Let's consider four interrupts shown above as A, B, C, and D:

```
A -> 0x00000004
B -> 0x00000302
C -> 0x00000409
D -> 0x00030609
```

Note

The bit positions for LEVEL 2 and onward are offset by 1, as 0 means that interrupt number is not present for that level. For our example, the LEVEL 3 controller has device D on line 2, connected to the LEVEL 2 controller's line 5, that is connected to the LEVEL 1 controller's line 9 (2 -> 5 -> 9). Because of the encoding offset for LEVEL 2 and onward, device D is given the number 0x00030609.

Preventing Interruptions In certain situations it may be necessary for the current thread to prevent ISRs from executing while it is performing time-sensitive or critical section operations.

A thread may temporarily prevent all IRQ handling in the system using an **IRQ lock**. This lock can be applied even when it is already in effect, so routines can use it without having to know

if it is already in effect. The thread must unlock its IRQ lock the same number of times it was locked before interrupts can be once again processed by the kernel while the thread is running.

Important

The IRQ lock is thread-specific. If thread A locks out interrupts then performs an operation that puts itself to sleep (e.g. sleeping for N milliseconds), the thread's IRQ lock no longer applies once thread A is swapped out and the next ready thread B starts to run.

This means that interrupts can be processed while thread B is running unless thread B has also locked out interrupts using its own IRQ lock. (Whether interrupts can be processed while the kernel is switching between two threads that are using the IRQ lock is architecture-specific.)

When thread A eventually becomes the current thread once again, the kernel re-establishes thread A's IRQ lock. This ensures thread A won't be interrupted until it has explicitly unlocked its IRQ lock.

If thread A does not sleep but does make a higher-priority thread B ready, the IRQ lock will inhibit any preemption that would otherwise occur. Thread B will not run until the next *reschedule point* reached after releasing the IRQ lock.

Alternatively, a thread may temporarily **disable** a specified IRQ so its associated ISR does not execute when the IRQ is signaled. The IRQ must be subsequently **enabled** to permit the ISR to execute.

Important

Disabling an IRQ prevents *all* threads in the system from being preempted by the associated ISR, not just the thread that disabled the IRQ.

Zero Latency Interrupts Preventing interruptions by applying an IRQ lock may increase the observed interrupt latency. A high interrupt latency, however, may not be acceptable for certain low-latency use-cases.

The kernel addresses such use-cases by allowing interrupts with critical latency constraints to execute at a priority level that cannot be blocked by interrupt locking. These interrupts are defined as *zero-latency interrupts*. The support for zero-latency interrupts requires `CONFIG_ZERO_LATENCY_IRQS` to be enabled. In addition to that, the flag `IRQ_ZERO_LATENCY` must be passed to *`IRQ_CONNECT`* or *`IRQ_DIRECT_CONNECT`* macros to configure the particular interrupt with zero latency.

Zero-latency interrupts are expected to be used to manage hardware events directly, and not to interoperate with the kernel code at all. They should treat all kernel APIs as undefined behavior (i.e. an application that uses the APIs inside a zero-latency interrupt context is responsible for directly verifying correct behavior). Zero-latency interrupts may not modify any data inspected by kernel APIs invoked from normal Zephyr contexts and shall not generate exceptions that need to be handled synchronously (e.g. kernel panic).

Important

Zero-latency interrupts are supported on an architecture-specific basis. The feature is currently implemented in the ARM Cortex-M architecture variant.

Offloading ISR Work An ISR should execute quickly to ensure predictable system operation. If time consuming processing is required the ISR should offload some or all processing to a thread, thereby restoring the kernel's ability to respond to other interrupts.

The kernel supports several mechanisms for offloading interrupt-related processing to a thread.

- An ISR can signal a helper thread to do interrupt-related processing using a kernel object, such as a FIFO, LIFO, or semaphore.
- An ISR can instruct the system workqueue thread to execute a work item. (See *Workqueue Threads*.)

When an ISR offloads work to a thread, there is typically a single context switch to that thread when the ISR completes, allowing interrupt-related processing to continue almost immediately. However, depending on the priority of the thread handling the offload, it is possible that the currently executing cooperative thread or other higher-priority threads may execute before the thread handling the offload is scheduled.

Sharing interrupt lines In the case of some hardware platforms, the same interrupt lines may be used by different IPs. For example, interrupt 17 may be used by a DMA controller to signal that a data transfer has been completed or by a DAI controller to signal that the transfer FIFO has reached its watermark. To make this work, one would have to either employ some special logic or find a workaround (for example, using the shared_irq interrupt controller), which doesn't scale very well.

To solve this problem, one may use shared interrupts, which can be enabled using `CONFIG_SHARED_INTERRUPTS`. Whenever an attempt to register a second ISR/argument pair on the same interrupt line is made (using *IRQ_CONNECT* or *irq_connect_dynamic()*), the interrupt line will become shared, meaning the two ISR/argument pairs (previous one and the one that has just been registered) will be invoked each time the interrupt is triggered. The entities that make use of an interrupt line in the shared interrupt context are known as clients. The maximum number of allowed clients for an interrupt is controlled by `CONFIG_SHARED_IRQ_MAX_NUM_CLIENTS`.

Interrupt sharing is transparent to the user. As such, the user may register interrupts using *IRQ_CONNECT* and *irq_connect_dynamic()* as they normally would. The interrupt sharing is taken care of behind the scenes.

Enabling the shared interrupt support and dynamic interrupt support will allow users to dynamically disconnect ISRs using *irq_disconnect_dynamic()*. After an ISR is disconnected, whenever the interrupt line for which it was register gets triggered, the ISR will no longer get invoked.

Please note that enabling `CONFIG_SHARED_INTERRUPTS` will result in a non-negligible increase in the binary size. Use with caution.

Implementation

Defining a regular ISR An ISR is defined at runtime by calling *IRQ_CONNECT*. It must then be enabled by calling *irq_enable()*.

> **Important**
>
> IRQ_CONNECT() is not a C function and does some inline assembly magic behind the scenes. All its arguments must be known at build time. Drivers that have multiple instances may need to define per-instance config functions to configure each instance of the interrupt.

The following code defines and enables an ISR.

```
#define MY_DEV_IRQ  24       /* device uses IRQ 24 */
#define MY_DEV_PRIO  2       /* device uses interrupt priority 2 */
/* argument passed to my_isr(), in this case a pointer to the device */
#define MY_ISR_ARG  DEVICE_GET(my_device)
#define MY_IRQ_FLAGS 0       /* IRQ flags */

void my_isr(void *arg)
{
   ... /* ISR code */
}

void my_isr_installer(void)
{
   ...
   IRQ_CONNECT(MY_DEV_IRQ, MY_DEV_PRIO, my_isr, MY_ISR_ARG, MY_IRQ_FLAGS);
   irq_enable(MY_DEV_IRQ);
   ...
}
```

Since the *IRQ_CONNECT* macro requires that all its parameters be known at build time, in some cases this may not be acceptable. It is also possible to install interrupts at runtime with *irq_connect_dynamic()*. It is used in exactly the same way as *IRQ_CONNECT*:

```
void my_isr_installer(void)
{
   ...
   irq_connect_dynamic(MY_DEV_IRQ, MY_DEV_PRIO, my_isr, MY_ISR_ARG,
                       MY_IRQ_FLAGS);
   irq_enable(MY_DEV_IRQ);
   ...
}
```

Dynamic interrupts require the CONFIG_DYNAMIC_INTERRUPTS option to be enabled. Removing or re-configuring a dynamic interrupt is currently unsupported.

Defining a 'direct' ISR Regular Zephyr interrupts introduce some overhead which may be unacceptable for some low-latency use-cases. Specifically:

- The argument to the ISR is retrieved and passed to the ISR
- If power management is enabled and the system was idle, all the hardware will be resumed from low-power state before the ISR is executed, which can be very time-consuming
- Although some architectures will do this in hardware, other architectures need to switch to the interrupt stack in code
- After the interrupt is serviced, the OS then performs some logic to potentially make a scheduling decision.

Zephyr supports so-called 'direct' interrupts, which are installed via *IRQ_DIRECT_CONNECT*. These direct interrupts have some special implementation requirements and a reduced feature set; see the definition of *IRQ_DIRECT_CONNECT* for details.

The following code demonstrates a direct ISR:

```
#define MY_DEV_IRQ  24       /* device uses IRQ 24 */
#define MY_DEV_PRIO  2       /* device uses interrupt priority 2 */
/* argument passed to my_isr(), in this case a pointer to the device */
#define MY_IRQ_FLAGS 0       /* IRQ flags */

ISR_DIRECT_DECLARE(my_isr)
{
```

(continues on next page)

(continued from previous page)

```
   do_stuff();
   ISR_DIRECT_PM(); /* PM done after servicing interrupt for best latency */
   return 1; /* We should check if scheduling decision should be made */
}

void my_isr_installer(void)
{
   ...
   IRQ_DIRECT_CONNECT(MY_DEV_IRQ, MY_DEV_PRIO, my_isr, MY_IRQ_FLAGS);
   irq_enable(MY_DEV_IRQ);
   ...
}
```

Installation of dynamic direct interrupts is supported on an architecture-specific basis. (The feature is currently implemented in ARM Cortex-M architecture variant. Dynamic direct interrupts feature is exposed to the user via an ARM-only API.)

Sharing an interrupt line The following code defines two ISRs using the same interrupt number.

```
#define MY_DEV_IRQ 24              /* device uses INTID 24 */
#define MY_DEV_IRQ_PRIO 2          /* device uses interrupt priority 2 */
/*  this argument may be anything */
#define MY_FST_ISR_ARG INT_TO_POINTER(1)
/*  this argument may be anything */
#define MY_SND_ISR_ARG INT_TO_POINTER(2)
#define MY_IRQ_FLAGS 0             /* IRQ flags */

void my_first_isr(void *arg)
{
   ... /* some magic happens here */
}

void my_second_isr(void *arg)
{
   ... /* even more magic happens here */
}

void my_isr_installer(void)
{
   ...
   IRQ_CONNECT(MY_DEV_IRQ, MY_DEV_IRQ_PRIO, my_first_isr, MY_FST_ISR_ARG, MY_IRQ_FLAGS);
   IRQ_CONNECT(MY_DEV_IRQ, MY_DEV_IRQ_PRIO, my_second_isr, MY_SND_ISR_ARG, MY_IRQ_FLAGS);
   ...
}
```

The same restrictions regarding *IRQ_CONNECT* described in *Defining a regular ISR* are applicable here. If CONFIG_SHARED_INTERRUPTS is disabled, the above code will generate a build error. Otherwise, the above code will result in the two ISRs being invoked each time interrupt 24 is triggered.

If CONFIG_SHARED_IRQ_MAX_NUM_CLIENTS is set to a value lower than 2 (current number of clients), a build error will be generated.

If dynamic interrupts are enabled, *irq_connect_dynamic()* will allow sharing interrupts during runtime. Exceeding the configured maximum number of allowed clients will result in a failed assertion.

Dynamically disconnecting an ISR The following code defines two ISRs using the same interrupt number. The second ISR is disconnected during runtime.

```
#define MY_DEV_IRQ 24              /* device uses INTID 24 */
#define MY_DEV_IRQ_PRIO 2          /* device uses interrupt priority 2 */
/*  this argument may be anything */
#define MY_FST_ISR_ARG INT_TO_POINTER(1)
/*  this argument may be anything */
#define MY_SND_ISR_ARG INT_TO_POINTER(2)
#define MY_IRQ_FLAGS 0             /* IRQ flags */

void my_first_isr(void *arg)
{
   ... /* some magic happens here */
}

void my_second_isr(void *arg)
{
   ... /* even more magic happens here */
}

void my_isr_installer(void)
{
   ...
   IRQ_CONNECT(MY_DEV_IRQ, MY_DEV_IRQ_PRIO, my_first_isr, MY_FST_ISR_ARG, MY_IRQ_FLAGS);
   IRQ_CONNECT(MY_DEV_IRQ, MY_DEV_IRQ_PRIO, my_second_isr, MY_SND_ISR_ARG, MY_IRQ_FLAGS);
   ...
}

void my_isr_uninstaller(void)
{
   ...
   irq_disconnect_dynamic(MY_DEV_IRQ, MY_DEV_IRQ_PRIO, my_first_isr, MY_FST_ISR_ARG, MY_IRQ_
↪FLAGS);
   ...
}
```

The *irq_disconnect_dynamic()* call will result in interrupt 24 becoming unshared, meaning the system will act as if the first *IRQ_CONNECT* call never happened. This behaviour is only allowed if CONFIG_DYNAMIC_INTERRUPTS is enabled, otherwise a linker error will be generated.

Implementation Details Interrupt tables are set up at build time using some special build tools. The details laid out here apply to all architectures except x86, which are covered in the *x86 Details* section below.

The invocation of *IRQ_CONNECT* will declare an instance of struct _isr_list which is placed in a special .intList section. This section is placed in compiled code on precompilation stages only. It is meant to be used by Zephyr script to generate interrupt tables and is removed from the final build. The script implements different parsers to process the data from .intList section and produce the required output.

The default parser generates C arrays filled with arguments and interrupt handlers in a form of addresses directly taken from .intList section entries. It works with all the architectures and compilers (with the exception mentioned above). The limitation of this parser is the fact that after the arrays are generated it is expected for the code not to relocate. Any relocation on this stage may lead to the situation where the entry in the interrupt array is no longer pointing to the function that was expected. It means that this parser, being more compatible is limiting us from using Link Time Optimization.

The local isr declaration parser uses different approach to construct the same arrays at binnary level. All the entries to the arrays are declared and defined locally, directly in the file where *IRQ_CONNECT* is used. They are placed in a section with the unique, synthesized name. The name

of the section is then placed in .intList section and it is used to create linker script to properly place the created entry in the right place in the memory. This parser is now limited to the supported architectures and toolchains but in reward it keeps the information about object relations for linker thus allowing the Link Time Optimization.

Implementation using C arrays This is the default configuration available for all Zephyr supported architectures.

Any invocation of *IRQ_CONNECT* will declare an instance of struct _isr_list which is placed in a special .intList section:

```
struct _isr_list {
    /** IRQ line number */
    int32_t irq;
    /** Flags for this IRQ, see ISR_FLAG_* definitions */
    int32_t flags;
    /** ISR to call */
    void *func;
    /** Parameter for non-direct IRQs */
    void *param;
};
```

Zephyr is built in two phases; the first phase of the build produces ${ZEPHYR_PREBUILT_EXECUTABLE}.elf which contains all the entries in the .intList section preceded by a header:

```
struct {
    void *spurious_irq_handler;
    void *sw_irq_handler;
    uint32_t num_isrs;
    uint32_t num_vectors;
    struct _isr_list isrs[];  <- of size num_isrs
};
```

This data consisting of the header and instances of struct _isr_list inside ${ZEPHYR_PREBUILT_EXECUTABLE}.elf is then used by the gen_isr_tables.py script to generate a C file defining a vector table and software ISR table that are then compiled and linked into the final application.

The priority level of any interrupt is not encoded in these tables, instead *IRQ_CONNECT* also has a runtime component which programs the desired priority level of the interrupt to the interrupt controller. Some architectures do not support the notion of interrupt priority, in which case the priority argument is ignored.

Vector Table A vector table is generated when CONFIG_GEN_IRQ_VECTOR_TABLE is enabled. This data structure is used natively by the CPU and is simply an array of function pointers, where each element n corresponds to the IRQ handler for IRQ line n, and the function pointers are:

1. For 'direct' interrupts declared with *IRQ_DIRECT_CONNECT*, the handler function will be placed here.
2. For regular interrupts declared with *IRQ_CONNECT*, the address of the common software IRQ handler is placed here. This code does common kernel interrupt bookkeeping and looks up the ISR and parameter from the software ISR table.
3. For interrupt lines that are not configured at all, the address of the spurious IRQ handler will be placed here. The spurious IRQ handler causes a system fatal error if encountered.

Some architectures (such as the Nios II internal interrupt controller) have a common entry point for all interrupts and do not support a vector table, in which case the CONFIG_GEN_IRQ_VECTOR_TABLE option should be disabled.

Some architectures may reserve some initial vectors for system exceptions and declare this in a table elsewhere, in which case CONFIG_GEN_IRQ_START_VECTOR needs to be set to properly offset the indices in the table.

SW ISR Table This is an array of struct _isr_table_entry:

```
struct _isr_table_entry {
    void *arg;
    void (*isr)(void *);
};
```

This is used by the common software IRQ handler to look up the ISR and its argument and execute it. The active IRQ line is looked up in an interrupt controller register and used to index this table.

Shared SW ISR Table This is an array of struct z_shared_isr_table_entry:

```
struct z_shared_isr_table_entry {
    struct _isr_table_entry clients[CONFIG_SHARED_IRQ_MAX_NUM_CLIENTS];
    size_t client_num;
};
```

This table keeps track of the registered clients for each of the interrupt lines. Whenever an interrupt line becomes shared, `z_shared_isr()` will replace the currently registered ISR in _sw_isr_table. This special ISR will iterate through the list of registered clients and invoke the ISRs.

Implementation using linker script This way of prepare and parse .isrList section to implement interrupt vectors arrays is called local isr declaration. The name comes from the fact that all the entries to the arrays that would create interrupt vectors are created locally in place of invocation of *IRQ_CONNECT* macro. Then automatically generated linker scripts are used to place it in the right place in the memory.

This option requires enabling by the choose of `ISR_TABLES_LOCAL_DECLARATION`. If this configuration is supported by the used architecture and toolchaing the `ISR_TABLES_LOCAL_DECLARATION_SUPPORTED` is set. See details of this option for the information about currently supported configurations.

Any invocation of *IRQ_CONNECT* or *IRQ_DIRECT_CONNECT* will declare an instance of struct _isr_list_sname which is placde in a special .intList section:

```
struct _isr_list_sname {
    /** IRQ line number */
    int32_t irq;
    /** Flags for this IRQ, see ISR_FLAG_* definitions */
    int32_t flags;
    /** The section name */
    const char sname[];
};
```

Note that the section name is placed in flexible array member. It means that the size of the initialized structure will warry depending on the structure name length. The whole entry is used by the script during the build of the application and has all the information needed for proper interrupt placement.

Beside of the _isr_list_sname the *IRQ_CONNECT* macro generates an entry that would be the part of the interrupt array:

```
struct _isr_table_entry {
    const void *arg;
    void (*isr)(const void *);
};
```

This array is placed in a section with the name saved in _isr_list_sname structure.

The values created by *IRQ_DIRECT_CONNECT* macro depends on the architecture. It can be changed to variable that points to a interrupt handler:

```
static uintptr_t <unique name> = ((uintptr_t)func);
```

Or to actually naked function that implements a jump to the interrupt handler:

```
static void <unique name>(void)
{
    __asm(ARCH_IRQ_VECTOR_JUMP_CODE(func));
}
```

Similar like for *IRQ_CONNECT*, the created variable or function is placed in a section, saved in _isr_list_sname section.

Files generated by the script The interrupt tables generator script creates 3 files: isr_tables.c, isr_tables_swi.ld, and isr_tables_vt.ld.

The isr_tables.c will contain all the structures for interrupts, direct interrupts and shared interrupts (if enabled). This file implements only all the structures that are not implemented by the application, leaving a comment where the interrupt not implemented here can be found.

Then two linker files are used. The isr_tables_vt.ld file is included in place where the interrupt vectors are required to be placed in the selected architecture. The isr_tables_swi.ld file describes the placement of the software interrupt table elements. The separated file is required as it might be placed in writable on nonwritable section, depending on the current configuration.

x86 Details The x86 architecture has a special type of vector table called the Interrupt Descriptor Table (IDT) which must be laid out in a certain way per the x86 processor documentation. It is still fundamentally a vector table, and the *arch/x86/gen_idt.py* tool uses the .intList section to create it. However, on APIC-based systems the indexes in the vector table do not correspond to the IRQ line. The first 32 vectors are reserved for CPU exceptions, and all remaining vectors (up to index 255) correspond to the priority level, in groups of 16. In this scheme, interrupts of priority level 0 will be placed in vectors 32-47, level 1 48-63, and so forth. When the *arch/x86/gen_idt.py* tool is constructing the IDT, when it configures an interrupt it will look for a free vector in the appropriate range for the requested priority level and set the handler there.

On x86 when an interrupt or exception vector is executed by the CPU, there is no foolproof way to determine which vector was fired, so a software ISR table indexed by IRQ line is not used. Instead, the *IRQ_CONNECT* call creates a small assembly language function which calls the common interrupt code in `_interrupt_enter()` with the ISR and parameter as arguments. It is the address of this assembly interrupt stub which gets placed in the IDT. For interrupts declared with *IRQ_DIRECT_CONNECT* the parameterless ISR is placed directly in the IDT.

On systems where the position in the vector table corresponds to the interrupt's priority level, the interrupt controller needs to know at runtime what vector is associated with an IRQ line. *arch/x86/gen_idt.py* additionally creates an _irq_to_interrupt_vector array which maps an IRQ line to its configured vector in the IDT. This is used at runtime by *IRQ_CONNECT* to program the IRQ-to-vector association in the interrupt controller.

For dynamic interrupts, the build must generate some 4-byte dynamic interrupt stubs, one stub per dynamic interrupt in use. The number of stubs is controlled by the `CONFIG_X86_DYNAMIC_IRQ_STUBS` option. Each stub pushes an unique identifier which is then used

to fetch the appropriate handler function and parameter out of a table populated when the dynamic interrupt was connected.

Going Beyond the Default Supported Number of Interrupts When generating interrupts in the multi-level configuration, 8-bits per level is the default mask used when determining which level a given interrupt code belongs to. This can become a problem when dealing with CPUs that support more than 255 interrupts per single aggregator. In this case it may be desirable to override these defaults and use a custom number of bits per level. Regardless of how many bits used for each level, the sum of the total bits used between all levels must sum to be less than or equal to 32-bits, fitting into a single 32-bit integer. To modify the bit total per level, override the default 8 in *Kconfig.multilevel* by setting `CONFIG_1ST_LEVEL_INTERRUPT_BITS` for the first level, `CONFIG_2ND_LEVEL_INTERRUPT_BITS` for the second level and `CONFIG_3RD_LEVEL_INTERRUPT_BITS` for the third level. These masks control the length of the bit masks and shift to apply when generating interrupt values, when checking the interrupts level and converting interrupts to a different level. The logic controlling this can be found in `irq_multilevel.h`

Suggested Uses Use a regular or direct ISR to perform interrupt processing that requires a very rapid response, and can be done quickly without blocking.

> **Note**
>
> Interrupt processing that is time consuming, or involves blocking, should be handed off to a thread. See *Offloading ISR Work* for a description of various techniques that can be used in an application.

Configuration Options Related configuration options:

- `CONFIG_ISR_STACK_SIZE`

Additional architecture-specific and device-specific configuration options also exist.

API Reference

group `isr_apis`

Defines

`IRQ_CONNECT`(irq_p, priority_p, isr_p, isr_param_p, flags_p)

Initialize an interrupt handler.

This routine initializes an interrupt handler for an IRQ. The IRQ must be subsequently enabled before the interrupt handler begins servicing interrupts.

> **Warning**
>
> Although this routine is invoked at run-time, all of its arguments must be computable by the compiler at build time.

Parameters

- `irq_p` – IRQ line number.
- `priority_p` – Interrupt priority.

- `isr_p` – Address of interrupt service routine.
- `isr_param_p` – Parameter passed to interrupt service routine.
- `flags_p` – Architecture-specific IRQ configuration flags..

`IRQ_DIRECT_CONNECT`(irq_p, priority_p, isr_p, flags_p)

Initialize a 'direct' interrupt handler.

This routine initializes an interrupt handler for an IRQ. The IRQ must be subsequently enabled via *irq_enable()* before the interrupt handler begins servicing interrupts.

These ISRs are designed for performance-critical interrupt handling and do not go through common interrupt handling code. They must be implemented in such a way that it is safe to put them directly in the vector table. For ISRs written in C, The *ISR_DIRECT_DECLARE()* macro will do this automatically. For ISRs written in assembly it is entirely up to the developer to ensure that the right steps are taken.

This type of interrupt currently has a few limitations compared to normal Zephyr interrupts:

- No parameters are passed to the ISR.
- No stack switch is done, the ISR will run on the interrupted context's stack, unless the architecture automatically does the stack switch in HW.
- Interrupt locking state is unchanged from how the HW sets it when the ISR runs. On arches that enter ISRs with interrupts locked, they will remain locked.
- Scheduling decisions are now optional, controlled by the return value of ISRs implemented with the *ISR_DIRECT_DECLARE()* macro
- The call into the OS to exit power management idle state is now optional. Normal interrupts always do this before the ISR is run, but when it runs is now controlled by the placement of a *ISR_DIRECT_PM()* macro, or omitted entirely.

Warning

Although this routine is invoked at run-time, all of its arguments must be computable by the compiler at build time.

Parameters

- `irq_p` – IRQ line number.
- `priority_p` – Interrupt priority.
- `isr_p` – Address of interrupt service routine.
- `flags_p` – Architecture-specific IRQ configuration flags.

`ISR_DIRECT_HEADER`()

Common tasks before executing the body of an ISR.

This macro must be at the beginning of all direct interrupts and performs minimal architecture-specific tasks before the ISR itself can run. It takes no arguments and has no return value.

`ISR_DIRECT_FOOTER`(check_reschedule)

Common tasks before exiting the body of an ISR.

This macro must be at the end of all direct interrupts and performs minimal architecture-specific tasks like EOI. It has no return value.

In a normal interrupt, a check is done at end of interrupt to invoke z_swap() logic if the current thread is preemptible and there is another thread ready to run in the kernel's ready queue cache. This is now optional and controlled by the check_reschedule argument. If unsure, set to nonzero. On systems that do stack switching and nested interrupt tracking in software, z_swap() should only be called if this was a non-nested interrupt.

Parameters

- **check_reschedule** – If nonzero, additionally invoke scheduling logic

ISR_DIRECT_PM()

Perform power management idle exit logic.

This macro may optionally be invoked somewhere in between IRQ_DIRECT_HEADER() and IRQ_DIRECT_FOOTER() invocations. It performs tasks necessary to exit power management idle state. It takes no parameters and returns no arguments. It may be omitted, but be careful!

ISR_DIRECT_DECLARE(name)

Helper macro to declare a direct interrupt service routine.

This will declare the function in a proper way and automatically include the *ISR_DIRECT_FOOTER()* and *ISR_DIRECT_HEADER()* macros. The function should return nonzero status if a scheduling decision should potentially be made. See *ISR_DIRECT_FOOTER()* for more details on the scheduling decision.

For architectures that support 'regular' and 'fast' interrupt types, where these interrupt types require different assembly language handling of registers by the ISR, this will always generate code for the 'fast' interrupt type.

Example usage:

```
ISR_DIRECT_DECLARE(my_isr)
{
        bool done = do_stuff();
        ISR_DIRECT_PM(); // done after do_stuff() due to latency concerns
        if (!done) {
            return 0; // don't bother checking if we have to z_swap()
        }

        k_sem_give(some_sem);
        return 1;
 }
```

Parameters

- **name** – symbol name of the ISR

irq_lock()

Lock interrupts.

This routine disables all interrupts on the CPU. It returns an unsigned integer "lock-out key", which is an architecture-dependent indicator of whether interrupts were locked prior to the call. The lock-out key must be passed to *irq_unlock()* to re-enable interrupts.

This routine can be called recursively, as long as the caller keeps track of each lock-out key that is generated. Interrupts are re-enabled by passing each of the keys to *irq_unlock()* in the reverse order they were acquired. (That is, each call to *irq_lock()* must be balanced by a corresponding call to *irq_unlock()*.)

This routine can only be invoked from supervisor mode. Some architectures (for example, ARM) will fail silently if invoked from user mode instead of generating an exception.

Note

This routine must also serve as a memory barrier to ensure the uniprocessor implementation of spinlocks is correct.

Note

This routine can be called by ISRs or by threads. If it is called by a thread, the interrupt lock is thread-specific; this means that interrupts remain disabled only while the thread is running. If the thread performs an operation that allows another thread to run (for example, giving a semaphore or sleeping for N milliseconds), the interrupt lock no longer applies and interrupts may be re-enabled while other processing occurs. When the thread once again becomes the current thread, the kernel re-establishes its interrupt lock; this ensures the thread won't be interrupted until it has explicitly released the interrupt lock it established.

Warning

The lock-out key should never be used to manually re-enable interrupts or to inspect or manipulate the contents of the CPU's interrupt bits.

Returns

An architecture-dependent lock-out key representing the "interrupt disable state" prior to the call.

`irq_unlock`(key)

Unlock interrupts.

This routine reverses the effect of a previous call to *irq_lock()* using the associated lock-out key. The caller must call the routine once for each time it called *irq_lock()*, supplying the keys in the reverse order they were acquired, before interrupts are enabled.

This routine can only be invoked from supervisor mode. Some architectures (for example, ARM) will fail silently if invoked from user mode instead of generating an exception.

Note

This routine must also serve as a memory barrier to ensure the uniprocessor implementation of spinlocks is correct.

Note

Can be called by ISRs.

Parameters

- **key** – Lock-out key generated by *irq_lock()*.

irq_enable(irq)

Enable an IRQ.

This routine enables interrupts from source *irq*.

Parameters

- **irq** – IRQ line.

irq_disable(irq)

Disable an IRQ.

This routine disables interrupts from source *irq*.

Parameters

- **irq** – IRQ line.

irq_is_enabled(irq)

Get IRQ enable state.

This routine indicates if interrupts from source *irq* are enabled.

Parameters

- **irq** – IRQ line.

Returns

interrupt enable state, true or false

Functions

static inline int **irq_connect_dynamic**(unsigned int irq, unsigned int priority, void (*routine)(const void *parameter), const void *parameter, uint32_t flags)

Configure a dynamic interrupt.

Use this instead of *IRQ_CONNECT()* if arguments cannot be known at build time.

Parameters

- **irq** – IRQ line number
- **priority** – Interrupt priority
- **routine** – Interrupt service routine
- **parameter** – ISR parameter
- **flags** – Arch-specific IRQ configuration flags

Returns

The vector assigned to this interrupt

static inline int **irq_disconnect_dynamic**(unsigned int irq, unsigned int priority, void (*routine)(const void *parameter), const void *parameter, uint32_t flags)

Disconnect a dynamic interrupt.

Use this in conjunction with shared interrupts to remove a routine/parameter pair from the list of clients using the same interrupt line. If the interrupt is not being shared then the associated _sw_isr_table entry will be replaced by (NULL, z_irq_spurious) (default entry).

Parameters

- `irq` – IRQ line number
- `priority` – Interrupt priority
- `routine` – Interrupt service routine
- `parameter` – ISR parameter
- `flags` – Arch-specific IRQ configuration flags

Returns
0 in case of success, negative value otherwise

bool `k_is_in_isr`(void)

Determine if code is running at interrupt level.

This routine allows the caller to customize its actions, depending on whether it is a thread or an ISR.

Function properties (list may not be complete)
isr-ok

Returns
false if invoked by a thread.

Returns
true if invoked by an ISR.

int `k_is_preempt_thread`(void)

Determine if code is running in a preemptible thread.

This routine allows the caller to customize its actions, depending on whether it can be preempted by another thread. The routine returns a 'true' value if all of the following conditions are met:

- The code is running in a thread, not at ISR.
- The thread's priority is in the preemptible range.
- The thread has not locked the scheduler.

Function properties (list may not be complete)
isr-ok

Returns
0 if invoked by an ISR or by a cooperative thread.

Returns
Non-zero if invoked by a preemptible thread.

static inline bool `k_is_pre_kernel`(void)

Test whether startup is in the before-main-task phase.

This routine allows the caller to customize its actions, depending on whether it being invoked before the kernel is fully active.

Function properties (list may not be complete)
isr-ok

Returns
true if invoked before post-kernel initialization

Returns
false if invoked during/after post-kernel initialization

Polling API

The polling API is used to wait concurrently for any one of multiple conditions to be fulfilled.

- *Concepts*
- *Implementation*
  - *Using k_poll()*
  - *Using k_poll_signal_raise()*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

Concepts The polling API's main function is *k_poll()*, which is very similar in concept to the POSIX *poll()* function, except that it operates on kernel objects rather than on file descriptors.

The polling API allows a single thread to wait concurrently for one or more conditions to be fulfilled without actively looking at each one individually.

There is a limited set of such conditions:

- a semaphore becomes available
- a kernel FIFO contains data ready to be retrieved
- a kernel message queue contains data ready to be retrieved
- a kernel pipe contains data ready to be retrieved
- a poll signal is raised

A thread that wants to wait on multiple conditions must define an array of **poll events**, one for each condition.

All events in the array must be initialized before the array can be polled on.

Each event must specify which **type** of condition must be satisfied so that its state is changed to signal the requested condition has been met.

Each event must specify what **kernel object** it wants the condition to be satisfied.

Each event must specify which **mode** of operation is used when the condition is satisfied.

Each event can optionally specify a **tag** to group multiple events together, to the user's discretion.

Apart from the kernel objects, there is also a **poll signal** pseudo-object type that be directly signaled.

The *k_poll()* function returns as soon as one of the conditions it is waiting for is fulfilled. It is possible for more than one to be fulfilled when *k_poll()* returns, if they were fulfilled before *k_poll()* was called, or due to the preemptive multi-threading nature of the kernel. The caller must look at the state of all the poll events in the array to figure out which ones were fulfilled and what actions to take.

Currently, there is only one mode of operation available: the object is not acquired. As an example, this means that when *k_poll()* returns and the poll event states that the semaphore is available, the caller of *k_poll()* must then invoke *k_sem_take()* to take ownership of the semaphore. If the semaphore is contested, there is no guarantee that it will be still available when *k_sem_take()* is called.

Implementation

Using k_poll() The main API is *k_poll()*, which operates on an array of poll events of type *k_poll_event*. Each entry in the array represents one event a call to *k_poll()* will wait for its condition to be fulfilled.

Poll events can be initialized using either the runtime initializers *K_POLL_EVENT_INITIALIZER()* or *k_poll_event_init()*, or the static initializer *K_POLL_EVENT_STATIC_INITIALIZER()*. An object that matches the **type** specified must be passed to the initializers. The **mode** *must* be set to *K_POLL_MODE_NOTIFY_ONLY*. The state *must* be set to *K_POLL_STATE_NOT_READY* (the initializers take care of this). The user **tag** is optional and completely opaque to the API: it is there to help a user to group similar events together. Being optional, it is passed to the static initializer, but not the runtime ones for performance reasons. If using runtime initializers, the user must set it separately in the *k_poll_event* data structure. If an event in the array is to be ignored, most likely temporarily, its type can be set to *K_POLL_TYPE_IGNORE*.

```
struct k_poll_event events[4] = {
    K_POLL_EVENT_STATIC_INITIALIZER(K_POLL_TYPE_SEM_AVAILABLE,
                                    K_POLL_MODE_NOTIFY_ONLY,
                                    &my_sem, 0),
    K_POLL_EVENT_STATIC_INITIALIZER(K_POLL_TYPE_FIFO_DATA_AVAILABLE,
                                    K_POLL_MODE_NOTIFY_ONLY,
                                    &my_fifo, 0),
    K_POLL_EVENT_STATIC_INITIALIZER(K_POLL_TYPE_MSGQ_DATA_AVAILABLE,
                                    K_POLL_MODE_NOTIFY_ONLY,
                                    &my_msgq, 0),
    K_POLL_EVENT_STATIC_INITIALIZER(K_POLL_TYPE_PIPE_DATA_AVAILABLE,
                                    K_POLL_MODE_NOTIFY_ONLY,
                                    &my_pipe, 0),
};
```

or at runtime

```
struct k_poll_event events[4];
void some_init(void)
{
    k_poll_event_init(&events[0],
                      K_POLL_TYPE_SEM_AVAILABLE,
                      K_POLL_MODE_NOTIFY_ONLY,
                      &my_sem);

    k_poll_event_init(&events[1],
                      K_POLL_TYPE_FIFO_DATA_AVAILABLE,
                      K_POLL_MODE_NOTIFY_ONLY,
                      &my_fifo);

    k_poll_event_init(&events[2],
                      K_POLL_TYPE_MSGQ_DATA_AVAILABLE,
                      K_POLL_MODE_NOTIFY_ONLY,
                      &my_msgq);

    k_poll_event_init(&events[3],
                      K_POLL_TYPE_PIPE_DATA_AVAILABLE,
```

(continues on next page)

(continued from previous page)

```
                    K_POLL_MODE_NOTIFY_ONLY,
                    &my_pipe);

    // tags are left uninitialized if unused
}
```

After the events are initialized, the array can be passed to *k_poll()*. A timeout can be specified to wait only for a specified amount of time, or the special values *K_NO_WAIT* and *K_FOREVER* to either not wait or wait until an event condition is satisfied and not sooner.

A list of pollers is offered on each semaphore or FIFO and as many events can wait in it as the app wants. Notice that the waiters will be served in first-come-first-serve order, not in priority order.

In case of success, *k_poll()* returns 0. If it times out, it returns -*EAGAIN*.

```
// assume there is no contention on this semaphore and FIFO
// -EADDRINUSE will not occur; the semaphore and/or data will be available

void do_stuff(void)
{
    rc = k_poll(events, ARRAY_SIZE(events), K_MSEC(1000));
    if (rc == 0) {
        if (events[0].state == K_POLL_STATE_SEM_AVAILABLE) {
            k_sem_take(events[0].sem, 0);
        } else if (events[1].state == K_POLL_STATE_FIFO_DATA_AVAILABLE) {
            data = k_fifo_get(events[1].fifo, 0);
            // handle data
        } else if (events[2].state == K_POLL_STATE_MSGQ_DATA_AVAILABLE) {
            ret = k_msgq_get(events[2].msgq, buf, K_NO_WAIT);
            // handle data
        } else if (events[3].state == K_POLL_STATE_PIPE_DATA_AVAILABLE) {
            ret = k_pipe_get(events[3].pipe, buf, bytes_to_read, &bytes_read, min_xfer, K_
↪NO_WAIT);
            // handle data
        }
    } else {
        // handle timeout
    }
}
```

When *k_poll()* is called in a loop, the events state must be reset to *K_POLL_STATE_NOT_READY* by the user.

```
void do_stuff(void)
{
    for(;;) {
        rc = k_poll(events, ARRAY_SIZE(events), K_FOREVER);
        if (events[0].state == K_POLL_STATE_SEM_AVAILABLE) {
            k_sem_take(events[0].sem, 0);
        } else if (events[1].state == K_POLL_STATE_FIFO_DATA_AVAILABLE) {
            data = k_fifo_get(events[1].fifo, 0);
            // handle data
        } else if (events[2].state == K_POLL_STATE_MSGQ_DATA_AVAILABLE) {
            ret = k_msgq_get(events[2].msgq, buf, K_NO_WAIT);
            // handle data
        } else if (events[3].state == K_POLL_STATE_PIPE_DATA_AVAILABLE) {
            ret = k_pipe_get(events[3].pipe, buf, bytes_to_read, &bytes_read, min_xfer, K_
↪NO_WAIT);
            // handle data
        }
```

(continues on next page)

(continued from previous page)

```
        events[0].state = K_POLL_STATE_NOT_READY;
        events[1].state = K_POLL_STATE_NOT_READY;
        events[2].state = K_POLL_STATE_NOT_READY;
        events[3].state = K_POLL_STATE_NOT_READY;
    }
}
```

Using k_poll_signal_raise() One of the types of events is *K_POLL_TYPE_SIGNAL*: this is a "direct" signal to a poll event. This can be seen as a lightweight binary semaphore only one thread can wait for.

A poll signal is a separate object of type *k_poll_signal* that must be attached to a k_poll_event, similar to a semaphore or FIFO. It must first be initialized either via *K_POLL_SIGNAL_INITIALIZER()* or *k_poll_signal_init()*.

```
struct k_poll_signal signal;
void do_stuff(void)
{
    k_poll_signal_init(&signal);
}
```

It is signaled via the *k_poll_signal_raise()* function. This function takes a user **result** parameter that is opaque to the API and can be used to pass extra information to the thread waiting on the event.

```
struct k_poll_signal signal;

// thread A
void do_stuff(void)
{
    k_poll_signal_init(&signal);

    struct k_poll_event events[1] = {
        K_POLL_EVENT_INITIALIZER(K_POLL_TYPE_SIGNAL,
                                 K_POLL_MODE_NOTIFY_ONLY,
                                 &signal),
    };

    k_poll(events, 1, K_FOREVER);

    int signaled, result;

    k_poll_signal_check(&signal, &signaled, &result);

    if (signaled && (result == 0x1337)) {
        // A-OK!
    } else {
        // weird error
    }
}

// thread B
void signal_do_stuff(void)
{
    k_poll_signal_raise(&signal, 0x1337);
}
```

If the signal is to be polled in a loop, *both* its event state must be reset to *K_POLL_STATE_NOT_READY* *and* its `result` must be reset using *k_poll_signal_reset()* on each iteration if it has been signaled.

```
struct k_poll_signal signal;
void do_stuff(void)
{
    k_poll_signal_init(&signal);

    struct k_poll_event events[1] = {
        K_POLL_EVENT_INITIALIZER(K_POLL_TYPE_SIGNAL,
                                 K_POLL_MODE_NOTIFY_ONLY,
                                 &signal),
    };

    for (;;) {
        k_poll(events, 1, K_FOREVER);

        int signaled, result;

        k_poll_signal_check(&signal, &signaled, &result);

        if (signaled && (result == 0x1337)) {
            // A-OK!
        } else {
            // weird error
        }

        k_poll_signal_reset(signal);
        events[0].state = K_POLL_STATE_NOT_READY;
    }
}
```

Note that poll signals are not internally synchronized. A *k_poll()* call that is passed a signal will return after any code in the system calls *k_poll_signal_raise()*. But if the signal is being externally managed and reset via *k_poll_signal_init()*, it is possible that by the time the application checks, the event state may no longer be equal to *K_POLL_STATE_SIGNALED*, and a (naive) application will miss events. Best practice is always to reset the signal only from within the thread invoking the *k_poll()* loop, or else to use some other event type which tracks event counts: semaphores and FIFOs are more error-proof in this sense because they can't "miss" events, architecturally.

Suggested Uses Use *k_poll()* to consolidate multiple threads that would be pending on one object each, saving possibly large amounts of stack space.

Use a poll signal as a lightweight binary semaphore if only one thread pends on it.

Note

Because objects are only signaled if no other thread is waiting for them to become available and only one thread can poll on a specific object, polling is best used when objects are not subject of contention between multiple threads, basically when a single thread operates as a main "server" or "dispatcher" for multiple objects and is the only one trying to acquire these objects.

Configuration Options Related configuration options:

- CONFIG_POLL

API Reference

group poll_apis

Defines

K_POLL_TYPE_IGNORE

K_POLL_TYPE_SIGNAL

K_POLL_TYPE_SEM_AVAILABLE

K_POLL_TYPE_DATA_AVAILABLE

K_POLL_TYPE_FIFO_DATA_AVAILABLE

K_POLL_TYPE_MSGQ_DATA_AVAILABLE

K_POLL_TYPE_PIPE_DATA_AVAILABLE

K_POLL_STATE_NOT_READY

K_POLL_STATE_SIGNALED

K_POLL_STATE_SEM_AVAILABLE

K_POLL_STATE_DATA_AVAILABLE

K_POLL_STATE_FIFO_DATA_AVAILABLE

K_POLL_STATE_MSGQ_DATA_AVAILABLE

K_POLL_STATE_PIPE_DATA_AVAILABLE

K_POLL_STATE_CANCELLED

K_POLL_SIGNAL_INITIALIZER(obj)

K_POLL_EVENT_INITIALIZER(_event_type, _event_mode, _event_obj)

K_POLL_EVENT_STATIC_INITIALIZER(_event_type, _event_mode, _event_obj, event_tag)

Enums

enum k_poll_modes

Values:

enumerator K_POLL_MODE_NOTIFY_ONLY = 0

enumerator K_POLL_NUM_MODES

Functions

void k_poll_event_init(struct *k_poll_event* *event, uint32_t type, int mode, void *obj)

Initialize one struct *k_poll_event* instance.

After this routine is called on a poll event, the event it ready to be placed in an event array to be passed to *k_poll()*.

Parameters

- event – The event to initialize.
- type – A bitfield of the types of event, from the K_POLL_TYPE_xxx values. Only values that apply to the same object being polled can be used together. Choosing K_POLL_TYPE_IGNORE disables the event.
- mode – Future. Use K_POLL_MODE_NOTIFY_ONLY.
- obj – Kernel object or poll signal.

int k_poll(struct *k_poll_event* *events, int num_events, *k_timeout_t* timeout)

Wait for one or many of multiple poll events to occur.

This routine allows a thread to wait concurrently for one or many of multiple poll events to have occurred. Such events can be a kernel object being available, like a semaphore, or a poll signal event.

When an event notifies that a kernel object is available, the kernel object is not "given" to the thread calling *k_poll()*: it merely signals the fact that the object was available when the *k_poll()* call was in effect. Also, all threads trying to acquire an object the regular way, i.e. by pending on the object, have precedence over the thread polling on the object. This means that the polling thread will never get the poll event on an object until the object becomes available and its pend queue is empty. For this reason, the *k_poll()* call is more effective when the objects being polled only have one thread, the polling thread, trying to acquire them.

When *k_poll()* returns 0, the caller should loop on all the events that were passed to *k_poll()* and check the state field for the values that were expected and take the associated actions.

Before being reused for another call to *k_poll()*, the user has to reset the state field to K_POLL_STATE_NOT_READY.

When called from user mode, a temporary memory allocation is required from the caller's resource pool.

Parameters

- events – An array of events to be polled for.
- num_events – The number of events in the array.
- timeout – Waiting period for an event to be ready, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- 0 – One or more events are ready.
- -EAGAIN – Waiting period timed out.

- **-EINTR** – Polling has been interrupted, e.g. with *k_queue_cancel_wait()*. All output events are still set and valid, cancelled event(s) will be set to K_POLL_STATE_CANCELLED. In other words, -EINTR status means that at least one of output events is K_POLL_STATE_CANCELLED.
- **-ENOMEM** – Thread resource pool insufficient memory (user mode only)
- **-EINVAL** – Bad parameters (user mode only)

void **k_poll_signal_init**(struct *k_poll_signal* *sig)

Initialize a poll signal object.

Ready a poll signal object to be signaled via *k_poll_signal_raise()*.

Parameters

- **sig** – A poll signal.

void **k_poll_signal_reset**(struct *k_poll_signal* *sig)

Reset a poll signal object's state to unsignaled.

Parameters

- **sig** – A poll signal object

void **k_poll_signal_check**(struct *k_poll_signal* *sig, unsigned int *signaled, int *result)

Fetch the signaled state and result value of a poll signal.

Parameters

- **sig** – A poll signal object
- **signaled** – An integer buffer which will be written nonzero if the object was signaled
- **result** – An integer destination buffer which will be written with the result value if the object was signaled, or an undefined value if it was not.

int **k_poll_signal_raise**(struct *k_poll_signal* *sig, int result)

Signal a poll signal object.

This routine makes ready a poll signal, which is basically a poll event of type K_POLL_TYPE_SIGNAL. If a thread was polling on that event, it will be made ready to run. A *result* value can be specified.

The poll signal contains a 'signaled' field that, when set by *k_poll_signal_raise()*, stays set until the user sets it back to 0 with *k_poll_signal_reset()*. It thus has to be reset by the user before being passed again to *k_poll()* or *k_poll()* will consider it being signaled, and will return immediately.

Note

The result is stored and the 'signaled' field is set even if this function returns an error indicating that an expiring poll was not notified. The next *k_poll()* will detect the missed raise.

Parameters

- **sig** – A poll signal.
- **result** – The value to store in the result field of the signal.

Return values

- **0** – The signal was delivered successfully.

- `-EAGAIN` – The polling thread's timeout is in the process of expiring.

struct `k_poll_signal`

#include <kernel.h>

Public Members

sys_dlist_t `poll_events`

PRIVATE - DO NOT TOUCH.

unsigned int `signaled`

1 if the event has been signaled, 0 otherwise.

Stays set to 1 until user resets it to 0.

int `result`

custom result value passed to *k_poll_signal_raise()* if needed

struct `k_poll_event`

#include <kernel.h> Poll Event.

Public Members

struct z_poller *`poller`

PRIVATE - DO NOT TOUCH.

uint32_t `tag`

optional user-specified tag, opaque, untouched by the API

uint32_t `type`

bitfield of event types (bitwise-ORed K_POLL_TYPE_xxx values)

uint32_t `state`

bitfield of event states (bitwise-ORed K_POLL_STATE_xxx values)

uint32_t `mode`

mode of operation, from enum k_poll_modes

uint32_t `unused`

unused bits in 32-bit word

`union k_poll_event`

per-type data

Semaphores

A *semaphore* is a kernel object that implements a traditional counting semaphore.

- *Concepts*
- *Implementation*
  - *Defining a Semaphore*
  - *Giving a Semaphore*
  - *Taking a Semaphore*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*
- *User Mode Semaphore API Reference*

Concepts Any number of semaphores can be defined (limited only by available RAM). Each semaphore is referenced by its memory address.

A semaphore has the following key properties:

- A **count** that indicates the number of times the semaphore can be taken. A count of zero indicates that the semaphore is unavailable.
- A **limit** that indicates the maximum value the semaphore's count can reach.

A semaphore must be initialized before it can be used. Its count must be set to a non-negative value that is less than or equal to its limit.

A semaphore may be **given** by a thread or an ISR. Giving the semaphore increments its count, unless the count is already equal to the limit.

A semaphore may be **taken** by a thread. Taking the semaphore decrements its count, unless the semaphore is unavailable (i.e. at zero). When a semaphore is unavailable a thread may choose to wait for it to be given. Any number of threads may wait on an unavailable semaphore simultaneously. When the semaphore is given, it is taken by the highest priority thread that has waited longest.

Note

You may initialize a "full" semaphore (count equal to limit) to limit the number of threads able to execute the critical section at the same time. You may also initialize an empty semaphore (count equal to 0, with a limit greater than 0) to create a gate through which no waiting thread may pass until the semaphore is incremented. All standard use cases of the common semaphore are supported.

Note

The kernel does allow an ISR to take a semaphore, however the ISR must not attempt to wait if the semaphore is unavailable.

Implementation

Defining a Semaphore A semaphore is defined using a variable of type k_sem. It must then be initialized by calling *k_sem_init()*.

The following code defines a semaphore, then configures it as a binary semaphore by setting its count to 0 and its limit to 1.

```
struct k_sem my_sem;

k_sem_init(&my_sem, 0, 1);
```

Alternatively, a semaphore can be defined and initialized at compile time by calling *K_SEM_DEFINE*.

The following code has the same effect as the code segment above.

```
K_SEM_DEFINE(my_sem, 0, 1);
```

Giving a Semaphore A semaphore is given by calling *k_sem_give()*.

The following code builds on the example above, and gives the semaphore to indicate that a unit of data is available for processing by a consumer thread.

```
void input_data_interrupt_handler(void *arg)
{
    /* notify thread that data is available */
    k_sem_give(&my_sem);

    ...
}
```

Taking a Semaphore A semaphore is taken by calling *k_sem_take()*.

The following code builds on the example above, and waits up to 50 milliseconds for the semaphore to be given. A warning is issued if the semaphore is not obtained in time.

```
void consumer_thread(void)
{
    ...

    if (k_sem_take(&my_sem, K_MSEC(50)) != 0) {
        printk("Input data not available!");
    } else {
        /* fetch available data */
        ...
    }
    ...
}
```

Suggested Uses Use a semaphore to control access to a set of resources by multiple threads.

Use a semaphore to synchronize processing between a producing and consuming threads or ISRs.

Configuration Options Related configuration options:

- None.

Related code samples

Basic Synchronization
Manipulate basic kernel synchronization primitives.

API Reference

group `semaphore_apis`

Defines

`K_SEM_MAX_LIMIT`
Maximum limit value allowed for a semaphore.

This is intended for use when a semaphore does not have an explicit maximum limit, and instead is just used for counting purposes.

`K_SEM_DEFINE`(name, initial_count, count_limit)
Statically define and initialize a semaphore.

The semaphore can be accessed outside the module where it is defined using:

```
extern struct k_sem <name>;
```

Parameters

- `name` – Name of the semaphore.
- `initial_count` – Initial semaphore count.
- `count_limit` – Maximum permitted semaphore count.

Functions

int `k_sem_init`(struct k_sem *sem, unsigned int initial_count, unsigned int limit)
Initialize a semaphore.

This routine initializes a semaphore object, prior to its first use.

See also

K_SEM_MAX_LIMIT

Parameters

- `sem` – Address of the semaphore.
- `initial_count` – Initial semaphore count.
- `limit` – Maximum permitted semaphore count.

Return values

- `0` – Semaphore created successfully
- `-EINVAL` – Invalid values

int **k_sem_take**(struct k_sem *sem, *k_timeout_t* timeout)

Take a semaphore.

This routine takes *sem*.

Function properties (list may not be complete)

isr-ok

> **Note**
>
> *timeout* must be set to K_NO_WAIT if called from ISR.

Parameters

- **sem** – Address of the semaphore.
- **timeout** – Waiting period to take the semaphore, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- **0** – Semaphore taken.
- **-EBUSY** – Returned without waiting.
- **-EAGAIN** – Waiting period timed out, or the semaphore was reset during the waiting period.

void **k_sem_give**(struct k_sem *sem)

Give a semaphore.

This routine gives *sem*, unless the semaphore is already at its maximum permitted count.

Function properties (list may not be complete)

isr-ok

Parameters

- **sem** – Address of the semaphore.

void **k_sem_reset**(struct k_sem *sem)

Resets a semaphore's count to zero.

This routine sets the count of *sem* to zero. Any outstanding semaphore takes will be aborted with -EAGAIN.

Parameters

- **sem** – Address of the semaphore.

unsigned int **k_sem_count_get**(struct k_sem *sem)

Get a semaphore's count.

This routine returns the current count of *sem*.

Parameters

- **sem** – Address of the semaphore.

Returns

Current semaphore count.

User Mode Semaphore API Reference The sys_sem exists in user memory working as counter semaphore for user mode thread when user mode enabled. When user mode isn't enabled, sys_sem behaves like k_sem.

group `user_semaphore_apis`

Defines

`SYS_SEM_DEFINE`(_name, _initial_count, _count_limit)

Statically define and initialize a sys_sem.

The semaphore can be accessed outside the module where it is defined using:

```
extern struct sys_sem <name>;
```

Route this to memory domains using K_APP_DMEM().

Parameters

- `_name` – Name of the semaphore.
- `_initial_count` – Initial semaphore count.
- `_count_limit` – Maximum permitted semaphore count.

Functions

int `sys_sem_init`(struct sys_sem *sem, unsigned int initial_count, unsigned int limit)

Initialize a semaphore.

This routine initializes a semaphore instance, prior to its first use.

Parameters

- `sem` – Address of the semaphore.
- `initial_count` – Initial semaphore count.
- `limit` – Maximum permitted semaphore count.

Return values

- `0` – Initial success.
- `-EINVAL` – Bad parameters, the value of limit should be located in (0, INT_MAX] and initial_count shouldn't be greater than limit.

int `sys_sem_give`(struct sys_sem *sem)

Give a semaphore.

This routine gives *sem*, unless the semaphore is already at its maximum permitted count.

Parameters

- `sem` – Address of the semaphore.

Return values

- `0` – Semaphore given.
- `-EINVAL` – Parameter address not recognized.
- `-EACCES` – Caller does not have enough access.
- `-EAGAIN` – Count reached Maximum permitted count and try again.

int **sys_sem_take**(struct sys_sem *sem, *k_timeout_t* timeout)

Take a sys_sem.

This routine takes *sem*.

Parameters

- **sem** – Address of the sys_sem.
- **timeout** – Waiting period to take the sys_sem, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- **0** – sys_sem taken.
- **-EINVAL** – Parameter address not recognized.
- **-ETIMEDOUT** – Waiting period timed out.
- **-EACCES** – Caller does not have enough access.

unsigned int **sys_sem_count_get**(struct sys_sem *sem)

Get sys_sem's value.

This routine returns the current value of *sem*.

Parameters

- **sem** – Address of the sys_sem.

Returns

Current value of sys_sem.

Mutexes

A *mutex* is a kernel object that implements a traditional reentrant mutex. A mutex allows multiple threads to safely share an associated hardware or software resource by ensuring mutually exclusive access to the resource.

- *Concepts*
  - *Reentrant Locking*
  - *Priority Inheritance*
- *Implementation*
  - *Defining a Mutex*
  - *Locking a Mutex*
  - *Unlocking a Mutex*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*
- *Futex API Reference*
- *User Mode Mutex API Reference*

Concepts Any number of mutexes can be defined (limited only by available RAM). Each mutex is referenced by its memory address.

A mutex has the following key properties:

- A **lock count** that indicates the number of times the mutex has been locked by the thread that has locked it. A count of zero indicates that the mutex is unlocked.
- An **owning thread** that identifies the thread that has locked the mutex, when it is locked.

A mutex must be initialized before it can be used. This sets its lock count to zero.

A thread that needs to use a shared resource must first gain exclusive rights to access it by **locking** the associated mutex. If the mutex is already locked by another thread, the requesting thread may choose to wait for the mutex to be unlocked.

After locking a mutex, the thread may safely use the associated resource for as long as needed; however, it is considered good practice to hold the lock for as short a time as possible to avoid negatively impacting other threads that want to use the resource. When the thread no longer needs the resource it must **unlock** the mutex to allow other threads to use the resource.

Any number of threads may wait on a locked mutex simultaneously. When the mutex becomes unlocked it is then locked by the highest-priority thread that has waited the longest.

Note

Mutex objects are *not* designed for use by ISRs.

Reentrant Locking A thread is permitted to lock a mutex it has already locked. This allows the thread to access the associated resource at a point in its execution when the mutex may or may not already be locked.

A mutex that is repeatedly locked by a thread must be unlocked an equal number of times before the mutex becomes fully unlocked so it can be claimed by another thread.

Priority Inheritance The thread that has locked a mutex is eligible for *priority inheritance*. This means the kernel will *temporarily* elevate the thread's priority if a higher priority thread begins waiting on the mutex. This allows the owning thread to complete its work and release the mutex more rapidly by executing at the same priority as the waiting thread. Once the mutex has been unlocked, the unlocking thread resets its priority to the level it had before locking that mutex.

Note

The `CONFIG_PRIORITY_CEILING` configuration option limits how high the kernel can raise a thread's priority due to priority inheritance. The default value of 0 permits unlimited elevation.

The owning thread's base priority is saved in the mutex when it obtains the lock. Each time a higher priority thread waits on a mutex, the kernel adjusts the owning thread's priority. When the owning thread releases the lock (or if the high priority waiting thread times out), the kernel restores the thread's base priority from the value saved in the mutex.

This works well for priority inheritance as long as only one locked mutex is involved. However, if multiple mutexes are involved, sub-optimal behavior will be observed if the mutexes are not unlocked in the reverse order to which the owning thread's priority was previously raised. Consequently it is recommended that a thread lock only a single mutex at a time when multiple mutexes are shared between threads of different priorities.

Implementation

Defining a Mutex A mutex is defined using a variable of type *k_mutex*. It must then be initialized by calling *k_mutex_init()*.

The following code defines and initializes a mutex.

```
struct k_mutex my_mutex;

k_mutex_init(&my_mutex);
```

Alternatively, a mutex can be defined and initialized at compile time by calling *K_MUTEX_DEFINE*.

The following code has the same effect as the code segment above.

```
K_MUTEX_DEFINE(my_mutex);
```

Locking a Mutex A mutex is locked by calling *k_mutex_lock()*.

The following code builds on the example above, and waits indefinitely for the mutex to become available if it is already locked by another thread.

```
k_mutex_lock(&my_mutex, K_FOREVER);
```

The following code waits up to 100 milliseconds for the mutex to become available, and gives a warning if the mutex does not become available.

```
if (k_mutex_lock(&my_mutex, K_MSEC(100)) == 0) {
    /* mutex successfully locked */
} else {
    printf("Cannot lock XYZ display\n");
}
```

Unlocking a Mutex A mutex is unlocked by calling *k_mutex_unlock()*.

The following code builds on the example above, and unlocks the mutex that was previously locked by the thread.

```
k_mutex_unlock(&my_mutex);
```

Suggested Uses Use a mutex to provide exclusive access to a resource, such as a physical device.

Configuration Options Related configuration options:

- CONFIG_PRIORITY_CEILING

API Reference

group **mutex_apis**

Defines

K_MUTEX_DEFINE(name)

Statically define and initialize a mutex.

The mutex can be accessed outside the module where it is defined using:

```
extern struct k_mutex <name>;
```

Parameters

- **name** – Name of the mutex.

Functions

int k_mutex_init(struct *k_mutex* *mutex)

Initialize a mutex.

This routine initializes a mutex object, prior to its first use.

Upon completion, the mutex is available and does not have an owner.

Parameters

- **mutex** – Address of the mutex.

Return values

0 – Mutex object created

int k_mutex_lock(struct *k_mutex* *mutex, *k_timeout_t* timeout)

Lock a mutex.

This routine locks *mutex*. If the mutex is locked by another thread, the calling thread waits until the mutex becomes available or until a timeout occurs.

A thread is permitted to lock a mutex it has already locked. The operation completes immediately and the lock count is increased by 1.

Mutexes may not be locked in ISRs.

Parameters

- **mutex** – Address of the mutex.
- **timeout** – Waiting period to lock the mutex, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- **0** – Mutex locked.
- **-EBUSY** – Returned without waiting.
- **-EAGAIN** – Waiting period timed out.

int k_mutex_unlock(struct *k_mutex* *mutex)

Unlock a mutex.

This routine unlocks *mutex*. The mutex must already be locked by the calling thread.

The mutex cannot be claimed by another thread until it has been unlocked by the calling thread as many times as it was previously locked by that thread.

Mutexes may not be unlocked in ISRs, as mutexes must only be manipulated in thread context due to ownership and priority inheritance semantics.

Parameters

- **mutex** – Address of the mutex.

Return values

- `0` – Mutex unlocked.
- `-EPERM` – The current thread does not own the mutex
- `-EINVAL` – The mutex is not locked

struct `k_mutex`

#include <*kernel.h*> Mutex Structure.

Public Members

_wait_q_t `wait_q`

Mutex wait queue.

struct *k_thread* *`owner`

Mutex owner.

uint32_t `lock_count`

Current lock count.

int `owner_orig_prio`

Original thread priority.

Futex API Reference k_futex is a lightweight mutual exclusion primitive designed to minimize kernel involvement. Uncontended operation relies only on atomic access to shared memory. k_futex are tracked as kernel objects and can live in user memory so that any access bypasses the kernel object permission management mechanism.

group `futex_apis`

Functions

int `k_futex_wait`(struct k_futex *futex, int expected, *k_timeout_t* timeout)

Pend the current thread on a futex.

Tests that the supplied futex contains the expected value, and if so, goes to sleep until some other thread calls *k_futex_wake()* on it.

Parameters

- `futex` – Address of the futex.
- `expected` – Expected value of the futex, if it is different the caller will not wait on it.
- `timeout` – Waiting period on the futex, or one of the special values K_NO_WAIT or K_FOREVER.

Return values

- `-EACCES` – Caller does not have read access to futex address.
- `-EAGAIN` – If the futex value did not match the expected parameter.
- `-EINVAL` – Futex parameter address not recognized by the kernel.

- `-ETIMEDOUT` – Thread woke up due to timeout and not a futex wakeup.
- `0` – if the caller went to sleep and was woken up. The caller should check the futex's value on wakeup to determine if it needs to block again.

int `k_futex_wake`(struct k_futex *futex, bool wake_all)

Wake one/all threads pending on a futex.

Wake up the highest priority thread pending on the supplied futex, or wakeup all the threads pending on the supplied futex, and the behavior depends on wake_all.

Parameters

- `futex` – Futex to wake up pending threads.
- `wake_all` – If true, wake up all pending threads; If false, wakeup the highest priority thread.

Return values

- `-EACCES` – Caller does not have access to the futex address.
- `-EINVAL` – Futex parameter address not recognized by the kernel.
- `Number` – of threads that were woken up.

User Mode Mutex API Reference sys_mutex behaves almost exactly like k_mutex, with the added advantage that a sys_mutex instance can reside in user memory. When user mode isn't enabled, sys_mutex behaves like k_mutex.

group `user_mutex_apis`

Defines

`SYS_MUTEX_DEFINE`(name)

Statically define and initialize a sys_mutex.

The mutex can be accessed outside the module where it is defined using:

```
extern struct sys_mutex <name>;
```

Route this to memory domains using K_APP_DMEM().

Parameters

- `name` – Name of the mutex.

Functions

static inline void `sys_mutex_init`(struct sys_mutex *mutex)

Initialize a mutex.

This routine initializes a mutex object, prior to its first use.

Upon completion, the mutex is available and does not have an owner.

This routine is only necessary to call when userspace is disabled and the mutex was not created with *SYS_MUTEX_DEFINE()*.

Parameters

- `mutex` – Address of the mutex.

static inline int **sys_mutex_lock**(struct sys_mutex *mutex, *k_timeout_t* timeout)

Lock a mutex.

This routine locks *mutex*. If the mutex is locked by another thread, the calling thread waits until the mutex becomes available or until a timeout occurs.

A thread is permitted to lock a mutex it has already locked. The operation completes immediately and the lock count is increased by 1.

Parameters

- **mutex** – Address of the mutex, which may reside in user memory
- **timeout** – Waiting period to lock the mutex, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- **0** – Mutex locked.
- **-EBUSY** – Returned without waiting.
- **-EAGAIN** – Waiting period timed out.
- **-EACCES** – Caller has no access to provided mutex address
- **-EINVAL** – Provided mutex not recognized by the kernel

static inline int **sys_mutex_unlock**(struct sys_mutex *mutex)

Unlock a mutex.

This routine unlocks *mutex*. The mutex must already be locked by the calling thread.

The mutex cannot be claimed by another thread until it has been unlocked by the calling thread as many times as it was previously locked by that thread.

Parameters

- **mutex** – Address of the mutex, which may reside in user memory

Return values

- **0** – Mutex unlocked
- **-EACCES** – Caller has no access to provided mutex address
- **-EINVAL** – Provided mutex not recognized by the kernel or mutex wasn't locked
- **-EPERM** – Caller does not own the mutex

Condition Variables

A *condition variable* is a synchronization primitive that enables threads to wait until a particular condition occurs.

- *Concepts*
- *Implementation*
  - *Defining a Condition Variable*
  - *Waiting on a Condition Variable*
  - *Signaling a Condition Variable*
- *Suggested Uses*

- *Configuration Options*
- *API Reference*

Concepts Any number of condition variables can be defined (limited only by available RAM). Each condition variable is referenced by its memory address.

To wait for a condition to become true, a thread can make use of a condition variable.

A condition variable is basically a queue of threads that threads can put themselves on when some state of execution (i.e., some condition) is not as desired (by waiting on the condition). The function *k_condvar_wait()* performs atomically the following steps;

1. Releases the last acquired mutex.
2. Puts the current thread in the condition variable queue.

Some other thread, when it changes said state, can then wake one (or more) of those waiting threads and thus allow them to continue by signaling on the condition using *k_condvar_signal()* or *k_condvar_broadcast()* then it:

1. Re-acquires the mutex previously released.
2. Returns from *k_condvar_wait()*.

A condition variable must be initialized before it can be used.

Implementation

Defining a Condition Variable A condition variable is defined using a variable of type `k_condvar`. It must then be initialized by calling *k_condvar_init()*.

The following code defines a condition variable:

```
struct k_condvar my_condvar;

k_condvar_init(&my_condvar);
```

Alternatively, a condition variable can be defined and initialized at compile time by calling *K_CONDVAR_DEFINE*.

The following code has the same effect as the code segment above.

```
K_CONDVAR_DEFINE(my_condvar);
```

Waiting on a Condition Variable A thread can wait on a condition by calling *k_condvar_wait()*.

The following code waits on the condition variable.

```
K_MUTEX_DEFINE(mutex);
K_CONDVAR_DEFINE(condvar)

int main(void)
{
    k_mutex_lock(&mutex, K_FOREVER);

    /* block this thread until another thread signals cond. While
     * blocked, the mutex is released, then re-acquired before this
```

(continues on next page)

(continued from previous page)

```
     * thread is woken up and the call returns.
     */
    k_condvar_wait(&condvar, &mutex, K_FOREVER);
    ...
    k_mutex_unlock(&mutex);
}
```

Signaling a Condition Variable A condition variable is signaled on by calling *k_condvar_signal()* for one thread or by calling *k_condvar_broadcast()* for multiple threads.

The following code builds on the example above.

```
void worker_thread(void)
{
    k_mutex_lock(&mutex, K_FOREVER);

    /*
     * Do some work and fulfill the condition
     */
    ...
    ...
    k_condvar_signal(&condvar);
    k_mutex_unlock(&mutex);
}
```

Suggested Uses Use condition variables with a mutex to signal changing states (conditions) from one thread to another thread. Condition variables are not the condition itself and they are not events. The condition is contained in the surrounding programming logic.

Mutexes alone are not designed for use as a notification/synchronization mechanism. They are meant to provide mutually exclusive access to a shared resource only.

Configuration Options Related configuration options:

- None.

Related code samples

Condition Variables
Manipulate condition variables in a multithreaded application.

API Reference

group condvar_apis

Defines

K_CONDVAR_DEFINE(name)

Statically define and initialize a condition variable.

The condition variable can be accessed outside the module where it is defined using:

```
extern struct k_condvar <name>;
```

Parameters

- **name** – Name of the condition variable.

Functions

int **k_condvar_init**(struct k_condvar *condvar)

Initialize a condition variable.

Parameters

- **condvar** – pointer to a k_condvar structure

Return values

0 – Condition variable created successfully

int **k_condvar_signal**(struct k_condvar *condvar)

Signals one thread that is pending on the condition variable.

Parameters

- **condvar** – pointer to a k_condvar structure

Return values

0 – On success

int **k_condvar_broadcast**(struct k_condvar *condvar)

Unblock all threads that are pending on the condition variable.

Parameters

- **condvar** – pointer to a k_condvar structure

Returns

An integer with number of woken threads on success

int **k_condvar_wait**(struct k_condvar *condvar, struct *k_mutex* *mutex, *k_timeout_t* timeout)

Waits on the condition variable releasing the mutex lock.

Atomically releases the currently owned mutex, blocks the current thread waiting on the condition variable specified by *condvar*, and finally acquires the mutex again.

The waiting thread unblocks only after another thread calls k_condvar_signal, or k_condvar_broadcast with the same condition variable.

Parameters

- **condvar** – pointer to a k_condvar structure
- **mutex** – Address of the mutex.
- **timeout** – Waiting period for the condition variable or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- **0** – On success
- **-EAGAIN** – Waiting period timed out.

Events

An *event object* is a kernel object that implements traditional events.

- *Concepts*
- *Implementation*
  - *Defining an Event Object*
  - *Setting Events*
  - *Posting Events*
  - *Waiting for Events*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

Concepts Any number of event objects can be defined (limited only by available RAM). Each event object is referenced by its memory address. One or more threads may wait on an event object until the desired set of events has been delivered to the event object. When new events are delivered to the event object, all threads whose wait conditions have been satisfied become ready simultaneously.

An event object has the following key properties:

- A 32-bit value that tracks which events have been delivered to it.

An event object must be initialized before it can be used.

Events may be **delivered** by a thread or an ISR. When delivering events, the events may either overwrite the existing set of events or add to them in a bitwise fashion. When overwriting the existing set of events, this is referred to as setting. When adding to them in a bitwise fashion, this is referred to as posting. Both posting and setting events have the potential to fulfill match conditions of multiple threads waiting on the event object. All threads whose match conditions have been met are made active at the same time.

Threads may wait on one or more events. They may either wait for all of the requested events, or for any of them. Furthermore, threads making a wait request have the option of resetting the current set of events tracked by the event object prior to waiting. Care must be taken with this option when multiple threads wait on the same event object.

Note

The kernel does allow an ISR to query an event object, however the ISR must not attempt to wait for the events.

Implementation

Defining an Event Object An event object is defined using a variable of type `k_event`. It must then be initialized by calling `k_event_init()`.

The following code defines an event object.

```
struct k_event my_event;

k_event_init(&my_event);
```

Alternatively, an event object can be defined and initialized at compile time by calling *K_EVENT_DEFINE*.

The following code has the same effect as the code segment above.

```
K_EVENT_DEFINE(my_event);
```

Setting Events Events in an event object are set by calling *k_event_set()*.

The following code builds on the example above, and sets the events tracked by the event object to 0x001.

```
void input_available_interrupt_handler(void *arg)
{
    /* notify threads that data is available */

    k_event_set(&my_event, 0x001);

    ...
}
```

Posting Events Events are posted to an event object by calling *k_event_post()*.

The following code builds on the example above, and posts a set of events to the event object.

```
void input_available_interrupt_handler(void *arg)
{
    ...

    /* notify threads that more data is available */

    k_event_post(&my_event, 0x120);

    ...
}
```

Waiting for Events Threads wait for events by calling *k_event_wait()*.

The following code builds on the example above, and waits up to 50 milliseconds for any of the specified events to be posted. A warning is issued if none of the events are posted in time.

```
void consumer_thread(void)
{
    uint32_t  events;

    events = k_event_wait(&my_event, 0xFFF, false, K_MSEC(50));
    if (events == 0) {
        printk("No input devices are available!");
    } else {
        /* Access the desired input device(s) */
        ...
    }
    ...
}
```

Alternatively, the consumer thread may desire to wait for all the events before continuing.

```
void consumer_thread(void)
{
    uint32_t  events;

    events = k_event_wait_all(&my_event, 0x121, false, K_MSEC(50));
    if (events == 0) {
        printk("At least one input device is not available!");
    } else {
        /* Access the desired input devices */
        ...
    }
    ...
}
```

Suggested Uses Use events to indicate that a set of conditions have occurred.

Use events to pass small amounts of data to multiple threads at once.

Configuration Options Related configuration options:

- CONFIG_EVENTS

API Reference

group event_apis

Defines

K_EVENT_DEFINE(name)

Statically define and initialize an event object.

The event can be accessed outside the module where it is defined using:

```
extern struct k_event <name>;
```

Parameters

- **name** – Name of the event object.

Functions

void k_event_init(struct *k_event* *event)

Initialize an event object.

This routine initializes an event object, prior to its first use.

Parameters

- **event** – Address of the event object.

uint32_t **k_event_post**(struct *k_event* *event, uint32_t events)

Post one or more events to an event object.

This routine posts one or more events to an event object. All tasks waiting on the event object *event* whose waiting conditions become met by this posting immediately unpend.

Posting differs from setting in that posted events are merged together with the current set of events tracked by the event object.

Parameters

- **event** – Address of the event object
- **events** – Set of events to post to *event*

Return values

Previous – value of the events in *event*

uint32_t **k_event_set**(struct *k_event* *event, uint32_t events)

Set the events in an event object.

This routine sets the events stored in event object to the specified value. All tasks waiting on the event object *event* whose waiting conditions become met by this immediately unpend.

Setting differs from posting in that set events replace the current set of events tracked by the event object.

Parameters

- **event** – Address of the event object
- **events** – Set of events to set in *event*

Return values

Previous – value of the events in *event*

uint32_t **k_event_set_masked**(struct *k_event* *event, uint32_t events, uint32_t events_mask)

Set or clear the events in an event object.

This routine sets the events stored in event object to the specified value. All tasks waiting on the event object *event* whose waiting conditions become met by this immediately unpend. Unlike *k_event_set*, this routine allows specific event bits to be set and cleared as determined by the mask.

Parameters

- **event** – Address of the event object
- **events** – Set of events to set/clear in *event*
- **events_mask** – Mask to be applied to *events*

Return values

Previous – value of the events in *events_mask*

uint32_t **k_event_clear**(struct *k_event* *event, uint32_t events)

Clear the events in an event object.

This routine clears (resets) the specified events stored in an event object.

Parameters

- **event** – Address of the event object
- **events** – Set of events to clear in *event*

Return values

`Previous` – value of the events in *event*

uint32_t `k_event_wait`(struct *k_event* *event, uint32_t events, bool reset, *k_timeout_t* timeout)

Wait for any of the specified events.

This routine waits on event object *event* until any of the specified events have been delivered to the event object, or the maximum wait time *timeout* has expired. A thread may wait on up to 32 distinctly numbered events that are expressed as bits in a single 32-bit word.

Note

The caller must be careful when resetting if there are multiple threads waiting for the event object *event*.

Parameters

- `event` – Address of the event object
- `events` – Set of desired events on which to wait
- `reset` – If true, clear the set of events tracked by the event object before waiting. If false, do not clear the events.
- `timeout` – Waiting period for the desired set of events or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- `set` – of matching events upon success
- `0` – if matching events were not received within the specified time

uint32_t `k_event_wait_all`(struct *k_event* *event, uint32_t events, bool reset, *k_timeout_t* timeout)

Wait for all of the specified events.

This routine waits on event object *event* until all of the specified events have been delivered to the event object, or the maximum wait time *timeout* has expired. A thread may wait on up to 32 distinctly numbered events that are expressed as bits in a single 32-bit word.

Note

The caller must be careful when resetting if there are multiple threads waiting for the event object *event*.

Parameters

- `event` – Address of the event object
- `events` – Set of desired events on which to wait
- `reset` – If true, clear the set of events tracked by the event object before waiting. If false, do not clear the events.
- `timeout` – Waiting period for the desired set of events or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- **set** – of matching events upon success
- **0** – if matching events were not received within the specified time

static inline uint32_t **k_event_test**(struct *k_event* *event, uint32_t events_mask)

Test the events currently tracked in the event object.

Parameters

- **event** – Address of the event object
- **events_mask** – Set of desired events to test

Return values

Current – value of events in *events_mask*

struct **k_event**

#include <*kernel.h*> Event Structure.

Symmetric Multiprocessing

On multiprocessor architectures, Zephyr supports the use of multiple physical CPUs running Zephyr application code. This support is "symmetric" in the sense that no specific CPU is treated specially by default. Any processor is capable of running any Zephyr thread, with access to all standard Zephyr APIs supported.

No special application code needs to be written to take advantage of this feature. If there are two Zephyr application threads runnable on a supported dual processor device, they will both run simultaneously.

SMP configuration is controlled under the CONFIG_SMP kconfig variable. This must be set to "y" to enable SMP features, otherwise a uniprocessor kernel will be built. In general the platform default will have enabled this anywhere it's supported. When enabled, the number of physical CPUs available is visible at build time as CONFIG_MP_MAX_NUM_CPUS. Likewise, the default for this will be the number of available CPUs on the platform and it is not expected that typical apps will change it. But it is legal and supported to set this to a smaller (but obviously not larger) number for special purposes (e.g. for testing, or to reserve a physical CPU for running non-Zephyr code).

Synchronization At the application level, core Zephyr IPC and synchronization primitives all behave identically under an SMP kernel. For example semaphores used to implement blocking mutual exclusion continue to be a proper application choice.

At the lowest level, however, Zephyr code has often used the *irq_lock()*/*irq_unlock()* primitives to implement fine grained critical sections using interrupt masking. These APIs continue to work via an emulation layer (see below), but the masking technique does not: the fact that your CPU will not be interrupted while you are in your critical section says nothing about whether a different CPU will be running simultaneously and be inspecting or modifying the same data!

Spinlocks SMP systems provide a more constrained *k_spin_lock()* primitive that not only masks interrupts locally, as done by *irq_lock()*, but also atomically validates that a shared lock variable has been modified before returning to the caller, "spinning" on the check if needed to wait for the other CPU to exit the lock. The default Zephyr implementation of *k_spin_lock()* and *k_spin_unlock()* is built on top of the pre-existing atomic_ layer (itself usually implemented using compiler intrinsics), though facilities exist for architectures to define their own for performance reasons.

One important difference between IRQ locks and spinlocks is that the earlier API was naturally recursive: the lock was global, so it was legal to acquire a nested lock inside of a critical section. Spinlocks are separable: you can have many locks for separate subsystems or data structures,

preventing CPUs from contending on a single global resource. But that means that spinlocks must not be used recursively. Code that holds a specific lock must not try to re-acquire it or it will deadlock (it is perfectly legal to nest **distinct** spinlocks, however). A validation layer is available to detect and report bugs like this.

When used on a uniprocessor system, the data component of the spinlock (the atomic lock variable) is unnecessary and elided. Except for the recursive semantics above, spinlocks in single-CPU contexts produce identical code to legacy IRQ locks. In fact the entirety of the Zephyr core kernel has now been ported to use spinlocks exclusively.

Legacy irq_lock() emulation For the benefit of applications written to the uniprocessor locking API, *irq_lock()* and *irq_unlock()* continue to work compatibly on SMP systems with identical semantics to their legacy versions. They are implemented as a single global spinlock, with a nesting count and the ability to be atomically reacquired on context switch into locked threads. The kernel will ensure that only one thread across all CPUs can hold the lock at any time, that it is released on context switch, and that it is re-acquired when necessary to restore the lock state when a thread is switched in. Other CPUs will spin waiting for the release to happen.

The overhead involved in this process has measurable performance impact, however. Unlike uniprocessor apps, SMP apps using *irq_lock()* are not simply invoking a very short (often ~1 instruction) interrupt masking operation. That, and the fact that the IRQ lock is global, means that code expecting to be run in an SMP context should be using the spinlock API wherever possible.

CPU Mask It is often desirable for real time applications to deliberately partition work across physical CPUs instead of relying solely on the kernel scheduler to decide on which threads to execute. Zephyr provides an API, controlled by the `CONFIG_SCHED_CPU_MASK` kconfig variable, which can associate a specific set of CPUs with each thread, indicating on which CPUs it can run.

By default, new threads can run on any CPU. Calling *k_thread_cpu_mask_disable()* with a particular CPU ID will prevent that thread from running on that CPU in the future. Likewise *k_thread_cpu_mask_enable()* will re-enable execution. There are also *k_thread_cpu_mask_clear()* and *k_thread_cpu_mask_enable_all()* APIs available for convenience. For obvious reasons, these APIs are illegal if called on a runnable thread. The thread must be blocked or suspended, otherwise an `-EINVAL` will be returned.

Note that when this feature is enabled, the scheduler algorithm involved in doing the per-CPU mask test requires that the list be traversed in full. The kernel does not keep a per-CPU run queue. That means that the performance benefits from the `CONFIG_SCHED_SCALABLE` and `CONFIG_SCHED_MULTIQ` scheduler backends cannot be realized. CPU mask processing is available only when `CONFIG_SCHED_DUMB` is the selected backend. This requirement is enforced in the configuration layer.

SMP Boot Process A Zephyr SMP kernel begins boot identically to a uniprocessor kernel. Auxiliary CPUs begin in a disabled state in the architecture layer. All standard kernel initialization, including device initialization, happens on a single CPU before other CPUs are brought online.

Just before entering the application `main()` function, the kernel calls `z_smp_init()` to launch the SMP initialization process. This enumerates over the configured CPUs, calling into the architecture layer using *arch_cpu_start()* for each one. This function is passed a memory region to use as a stack on the foreign CPU (in practice it uses the area that will become that CPU's interrupt stack), the address of a local `smp_init_top()` callback function to run on that CPU, and a pointer to a "start flag" address which will be used as an atomic signal.

The local SMP initialization (`smp_init_top()`) on each CPU is then invoked by the architecture layer. Note that interrupts are still masked at this point. This routine is responsible for calling `smp_timer_init()` to set up any needed stat in the timer driver. On many architectures the timer is a per-CPU device and needs to be configured specially on auxiliary CPUs. Then it waits

(spinning) for the atomic “start flag” to be released in the main thread, to guarantee that all SMP initialization is complete before any Zephyr application code runs, and finally calls `z_swap()` to transfer control to the appropriate runnable thread via the standard scheduler API.

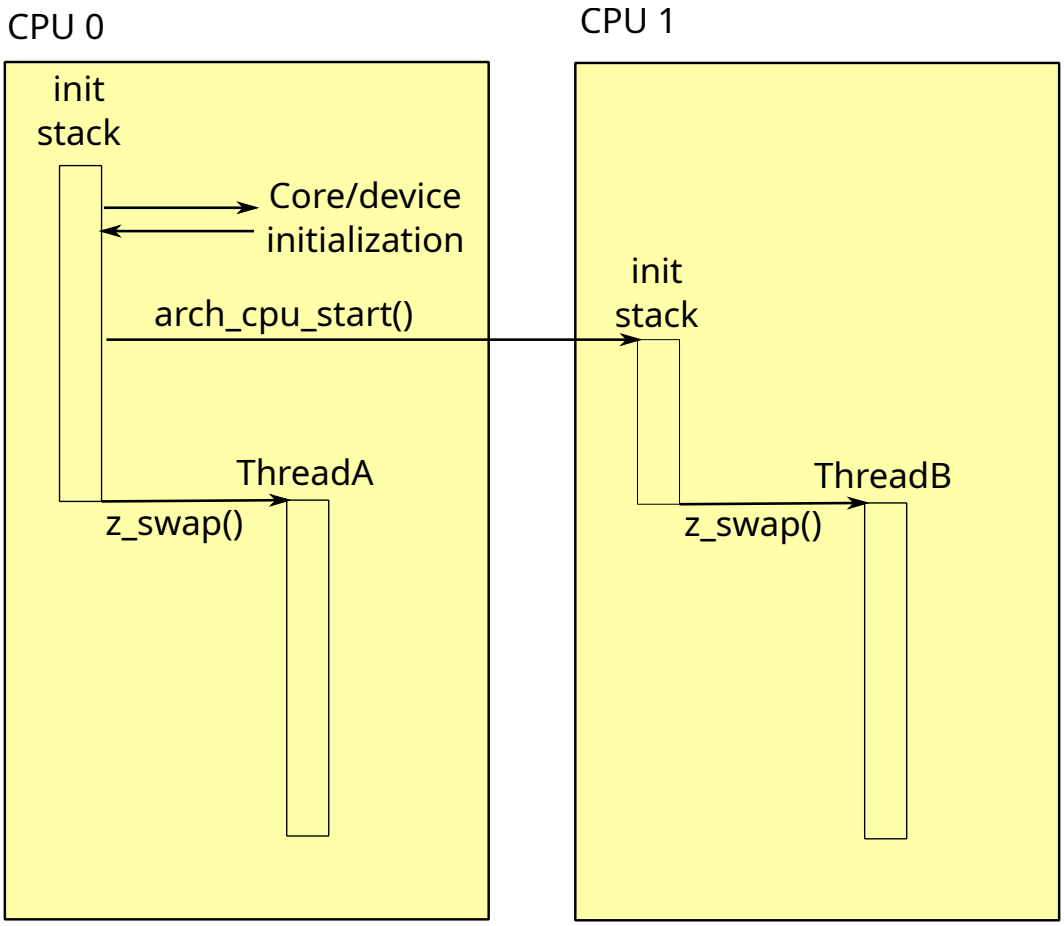


Fig. 1: Example SMP initialization process, showing a configuration with two CPUs and two app threads which begin operating simultaneously.

Interprocessor Interrupts When running in multiprocessor environments, it is occasionally the case that state modified on the local CPU needs to be synchronously handled on a different processor.

One example is the Zephyr *k_thread_abort()* API, which cannot return until the thread that had been aborted is no longer runnable. If it is currently running on another CPU, that becomes difficult to implement.

Another is low power idle. It is a firm requirement on many devices that system idle be implemented using a low-power mode with as many interrupts (including periodic timer interrupts) disabled or deferred as is possible. If a CPU is in such a state, and on another CPU a thread becomes runnable, the idle CPU has no way to “wake up” to handle the newly-runnable load.

So where possible, Zephyr SMP architectures should implement an interprocessor interrupt. The current framework is very simple: the architecture provides at least a *arch_sched_broadcast_ipi()* call, which when invoked will flag an interrupt on all CPUs (except the current one, though that is allowed behavior). If the architecture supports directed IPIs (see `CONFIG_ARCH_HAS_DIRECTED_IPIS`), then the architecture also provides a *arch_sched_directed_ipi()* call, which when invoked will flag an interrupt on the specified CPUs. When an interrupt is flagged on the CPUs, the `z_sched_ipi()` function implemented in the scheduler will get invoked on those CPUs. The expectation is that these APIs will evolve over

time to encompass more functionality (e.g. cross-CPU calls), and that the scheduler-specific calls here will be implemented in terms of a more general framework.

Note that not all SMP architectures will have a usable IPI mechanism (either missing, or just undocumented/unimplemented). In those cases Zephyr provides fallback behavior that is correct, but perhaps suboptimal.

Using this, *k_thread_abort()* becomes only slightly more complicated in SMP: for the case where a thread is actually running on another CPU (we can detect this atomically inside the scheduler), we broadcast an IPI and spin, waiting for the thread to either become "DEAD" or for it to re-enter the queue (in which case we terminate it the same way we would have in uniprocessor mode). Note that the "aborted" check happens on any interrupt exit, so there is no special handling needed in the IPI per se. This allows us to implement a reasonable fallback when IPI is not available: we can simply spin, waiting until the foreign CPU receives any interrupt, though this may be a much longer time!

Likewise idle wakeups are trivially implementable with an empty IPI handler. If a thread is added to an empty run queue (i.e. there may have been idle CPUs), we broadcast an IPI. A foreign CPU will then be able to see the new thread when exiting from the interrupt and will switch to it if available.

Without an IPI, however, a low power idle that requires an interrupt will not work to synchronously run new threads. The workaround in that case is more invasive: Zephyr will **not** enter the system idle handler and will instead spin in its idle loop, testing the scheduler state at high frequency (not spinning on it though, as that would involve severe lock contention) for new threads. The expectation is that power constrained SMP applications are always going to provide an IPI, and this code will only be used for testing purposes or on systems without power consumption requirements.

IPI Cascades The kernel can not control the order in which IPIs are processed by the CPUs in the system. In general, this is not an issue and a single set of IPIs is sufficient to trigger a reschedule on the N CPUs that results with them scheduling the highest N priority ready threads to execute. When CPU masking is used, there may be more than one valid set of threads (not to be confused with an optimal set of threads) that can be scheduled on the N CPUs and a single set of IPIs may be insufficient to result in any of these valid sets.

Note

When CPU masking is not in play, the optimal set of threads is the same as the valid set of threads. However when CPU masking is in play, there may be more than one valid set–one of which may be optimal.

To better illustrate the distinction, consider a 2-CPU system with ready threads T1 and T2 at priorities 1 and 2 respectively. Let T2 be pinned to CPU0 and T1 not be pinned. If CPU0 is executing T2 and CPU1 executing T1, then this set is is both valid and optimal. However, if CPU0 is executing T1 and CPU1 is idling, then this too would be valid though not optimal.

In those cases where a single set of IPIs is not sufficient to generate a valid set, the resulting set of executing threads are expected to be close to a valid set, and subsequent IPIs can generally be expected to correct the situation soon. However, for cases where neither the approximation nor the delay are acceptable, enabling `CONFIG_SCHED_IPI_CASCADE` will allow the kernel to generate cascading IPIs until the kernel has selected a valid set of ready threads for the CPUs.

There are three types of costs/penalties associated with the IPI cascades–and for these reasons they are disabled by default. The first is a cost incurred by the CPU producing the IPI when a new thread preempts the old thread as checks must be done to compare the old thread against the threads executing on the other CPUs. The second is a cost incurred by the CPUs receiving the IPIs as they must be processed. The third is the apparent sputtering of a thread as it "winks in" and then "winks out" due to cascades stemming from the aforementioned first cost.

SMP Kernel Internals In general, Zephyr kernel code is SMP-agnostic and, like application code, will work correctly regardless of the number of CPUs available. But in a few areas there are notable changes in structure or behavior.

Per-CPU data Many elements of the core kernel data need to be implemented for each CPU in SMP mode. For example, the `_current` thread pointer obviously needs to reflect what is running locally, there are many threads running concurrently. Likewise a kernel-provided interrupt stack needs to be created and assigned for each physical CPU, as does the interrupt nesting count used to detect ISR state.

These fields are now moved into a separate struct _cpu instance within the `_kernel` struct, which has a `cpus[]` array indexed by ID. Compatibility fields are provided for legacy uniprocessor code trying to access the fields of `cpus[0]` using the older syntax and assembly offsets.

Note that an important requirement on the architecture layer is that the pointer to this CPU struct be available rapidly when in kernel context. The expectation is that *arch_curr_cpu()* will be implemented using a CPU-provided register or addressing mode that can store this value across arbitrary context switches or interrupts and make it available to any kernel-mode code.

Similarly, where on a uniprocessor system Zephyr could simply create a global "idle thread" at the lowest priority, in SMP we may need one for each CPU. This makes the internal predicate test for "_is_idle()" in the scheduler, which is a hot path performance environment, more complicated than simply testing the thread pointer for equality with a known static variable. In SMP mode, idle threads are distinguished by a separate field in the thread struct.

Switch-based context switching The traditional Zephyr context switch primitive has been `z_swap()`. Unfortunately, this function takes no argument specifying a thread to switch to. The expectation has always been that the scheduler has already made its preemption decision when its state was last modified and cached the resulting "next thread" pointer in a location where architecture context switch primitives can find it via a simple struct offset. That technique will not work in SMP, because the other CPU may have modified scheduler state since the current CPU last exited the scheduler (for example: it might already be running that cached thread!).

Instead, the SMP "switch to" decision needs to be made synchronously with the swap call, and as we don't want per-architecture assembly code to be handling scheduler internal state, Zephyr requires a somewhat lower-level context switch primitives for SMP systems: *arch_switch()* is always called with interrupts masked, and takes exactly two arguments. The first is an opaque (architecture defined) handle to the context to which it should switch, and the second is a pointer to such a handle into which it should store the handle resulting from the thread that is being switched out. The kernel then implements a portable `z_swap()` implementation on top of this primitive which includes the relevant scheduler logic in a location where the architecture doesn't need to understand it.

Similarly, on interrupt exit, switch-based architectures are expected to call `z_get_next_switch_handle()` to retrieve the next thread to run from the scheduler. The argument to `z_get_next_switch_handle()` is either the interrupted thread's "handle" reflecting the same opaque type used by *arch_switch()*, or NULL if that thread cannot be released to the scheduler just yet. The choice between a handle value or NULL depends on the way CPU interrupt mode is implemented.

Architectures with a large CPU register file would typically preserve only the caller-saved registers on the current thread's stack when interrupted in order to minimize interrupt latency, and preserve the callee-saved registers only when *arch_switch()* is called to minimize context switching latency. Such architectures must use NULL as the argument to `z_get_next_switch_handle()` to determine if there is a new thread to schedule, and follow through with their own *arch_switch()* or derivative if so, or directly leave interrupt mode otherwise. In the former case it is up to that switch code to store the handle resulting from the thread that is being switched out in that thread's "switch_handle" field after its context has fully been saved.

Architectures whose entry in interrupt mode already preserves the entire thread state may pass that thread's handle directly to `z_get_next_switch_handle()` and be done in one step.

Note that while SMP requires `CONFIG_USE_SWITCH`, the reverse is not true. A uniprocessor architecture built with `CONFIG_SMP` set to No might still decide to implement its context switching using *arch_switch()*.

API Reference

group **spinlock_apis**

Spinlock APIs.

Defines

`K_SPINLOCK_BREAK`

Leaves a code block guarded with *K_SPINLOCK* after releasing the lock.

See *K_SPINLOCK* for details.

`K_SPINLOCK`(lck)

Guards a code block with the given spinlock, automatically acquiring the lock before executing the code block.

The lock will be released either when reaching the end of the code block or when leaving the block with *K_SPINLOCK_BREAK*.

Example usage:

```
K_SPINLOCK(&mylock) {

  ...execute statements with the lock held...

  if (some_condition) {
    ...release the lock and leave the guarded section prematurely:
    K_SPINLOCK_BREAK;
  }

  ...execute statements with the lock held...

}
```

Behind the scenes this pattern expands to a for-loop whose body is executed exactly once:

```
for (k_spinlock_key_t key = k_spin_lock(&mylock); ...; k_spin_unlock(&mylock,␣
↪key)) {
    ...
}
```

Note

In user mode the spinlock must be placed in memory accessible to the application, see K_APP_DMEM and K_APP_BMEM macros for details.

Warning

The code block must execute to its end or be left by calling *K_SPINLOCK_BREAK*. Otherwise, e.g. if exiting the block with a break, goto or return statement, the spinlock will not be released on exit.

Parameters

- `lck` – Spinlock used to guard the enclosed code block.

Typedefs

typedef struct z_spinlock_key **k_spinlock_key_t**

Spinlock key type.

This type defines a "key" value used by a spinlock implementation to store the system interrupt state at the time of a call to *k_spin_lock()*. It is expected to be passed to a matching *k_spin_unlock()*.

This type is opaque and should not be inspected by application code.

Functions

ALWAYS_INLINE static *k_spinlock_key_t* **k_spin_lock**(struct *k_spinlock* *l)

Lock a spinlock.

This routine locks the specified spinlock, returning a key handle representing interrupt state needed at unlock time. Upon returning, the calling thread is guaranteed not to be suspended or interrupted on its current CPU until it calls *k_spin_unlock()*. The implementation guarantees mutual exclusion: exactly one thread on one CPU will return from *k_spin_lock()* at a time. Other CPUs trying to acquire a lock already held by another CPU will enter an implementation-defined busy loop ("spinning") until the lock is released.

Separate spin locks may be nested. It is legal to lock an (unlocked) spin lock while holding a different lock. Spin locks are not recursive, however: an attempt to acquire a spin lock that the CPU already holds will deadlock.

In circumstances where only one CPU exists, the behavior of *k_spin_lock()* remains as specified above, though obviously no spinning will take place. Implementations may be free to optimize in uniprocessor contexts such that the locking reduces to an interrupt mask operation.

Parameters

- `l` – A pointer to the spinlock to lock

Returns

A key value that must be passed to *k_spin_unlock()* when the lock is released.

ALWAYS_INLINE static int **k_spin_trylock**(struct *k_spinlock* *l, *k_spinlock_key_t* *k)

Attempt to lock a spinlock.

This routine makes one attempt to lock `l`. If it is successful, then it will store the key into `k`.

See also

k_spin_lock

See also

k_spin_unlock

Parameters

- `l` – **[in]** A pointer to the spinlock to lock
- `k` – **[out]** A pointer to the spinlock key

Return values

- `0` – on success
- `-EBUSY` – if another thread holds the lock

ALWAYS_INLINE static void **k_spin_unlock**(struct *k_spinlock* *l, *k_spinlock_key_t* key)

Unlock a spin lock.

This releases a lock acquired by *k_spin_lock()*. After this function is called, any CPU will be able to acquire the lock. If other CPUs are currently spinning inside *k_spin_lock()* waiting for this lock, exactly one of them will return synchronously with the lock held.

Spin locks must be properly nested. A call to *k_spin_unlock()* must be made on the lock object most recently locked using *k_spin_lock()*, using the key value that it returned. Attempts to unlock mis-nested locks, or to unlock locks that are not held, or to passing a key parameter other than the one returned from *k_spin_lock()*, are illegal. When CONFIG_SPIN_VALIDATE is set, some of these errors can be detected by the framework.

Parameters

- `l` – A pointer to the spinlock to release
- `key` – The value returned from *k_spin_lock()* when this lock was acquired

struct **k_spinlock**

#include <*spinlock.h*> Kernel Spin Lock.

This struct defines a spin lock record on which CPUs can wait with *k_spin_lock()*. Any number of spinlocks may be defined in application code.

3.1.2 Data Passing

These pages cover kernel objects which can be used to pass data between threads and ISRs.

The following table summarizes their high-level features.

Object	Bidirectional?	Data structure	Data item size	Data Alignment	ISRs can receive?	ISRs can send?	Overrun handling
FIFO	No	Queue	Arbitrary [1]	4 B [2]	Yes [3]	Yes	N/A
LIFO	No	Queue	Arbitrary [1]	4 B [2]	Yes [3]	Yes	N/A
Stack	No	Array	Word	Word	Yes [3]	Yes	Undefined behavior
Message queue	No	Ring buffer	Arbitrary [6]	Power of two	Yes [3]	Yes	Pend thread or return -errno
Mailbox	Yes	Queue	Arbitrary [1]	Arbitrary	No	No	N/A
Pipe	No	Ring buffer [4]	Arbitrary	Arbitrary	Yes [5]	Yes [5]	Pend thread or return -errno

[1] Callers allocate space for queue overhead in the data elements themselves.

[2] Objects added with k_fifo_alloc_put() and k_lifo_alloc_put() do not have alignment constraints, but use temporary memory from the calling thread's resource pool.

[3] ISRs can receive only when passing K_NO_WAIT as the timeout argument.

[4] Optional.

[5] ISRS can send and/or receive only when passing K_NO_WAIT as the timeout argument.

[6] Data item size must be a multiple of the data alignment.

Queues

A Queue in Zephyr is a kernel object that implements a traditional queue, allowing threads and ISRs to add and remove data items of any size. The queue is similar to a FIFO and serves as the underlying implementation for both *k_fifo* and *k_lifo*. For more information on usage see *k_fifo*.

Configuration Options Related configuration options:

- None

API Reference

group `queue_apis`

Defines

`K_QUEUE_DEFINE`(name)

Statically define and initialize a queue.

The queue can be accessed outside the module where it is defined using:

```
extern struct k_queue <name>;
```

Parameters

- **name** – Name of the queue.

Functions

void **k_queue_init**(struct k_queue *queue)

Initialize a queue.

This routine initializes a queue object, prior to its first use.

Parameters

- **queue** – Address of the queue.

void **k_queue_cancel_wait**(struct k_queue *queue)

Cancel waiting on a queue.

This routine causes first thread pending on *queue*, if any, to return from *k_queue_get()* call with NULL value (as if timeout expired). If the queue is being waited on by *k_poll()*, it will return with -EINTR and K_POLL_STATE_CANCELLED state (and per above, subsequent *k_queue_get()* will return NULL).

Function properties (list may not be complete)

isr-ok

Parameters

- **queue** – Address of the queue.

void **k_queue_append**(struct k_queue *queue, void *data)

Append an element to the end of a queue.

This routine appends a data item to *queue*. A queue data item must be aligned on a word boundary, and the first word of the item is reserved for the kernel's use.

Function properties (list may not be complete)

isr-ok

Parameters

- **queue** – Address of the queue.
- **data** – Address of the data item.

int32_t **k_queue_alloc_append**(struct k_queue *queue, void *data)

Append an element to a queue.

This routine appends a data item to *queue*. There is an implicit memory allocation to create an additional temporary bookkeeping data structure from the calling thread's resource pool, which is automatically freed when the item is removed. The data itself is not copied.

Function properties (list may not be complete)

isr-ok

Parameters

- `queue` – Address of the queue.
- `data` – Address of the data item.

Return values

- `0` – on success
- `-ENOMEM` – if there isn't sufficient RAM in the caller's resource pool

void `k_queue_prepend`(struct k_queue *queue, void *data)

Prepend an element to a queue.

This routine prepends a data item to *queue*. A queue data item must be aligned on a word boundary, and the first word of the item is reserved for the kernel's use.

Function properties (list may not be complete)

isr-ok

Parameters

- `queue` – Address of the queue.
- `data` – Address of the data item.

int32_t `k_queue_alloc_prepend`(struct k_queue *queue, void *data)

Prepend an element to a queue.

This routine prepends a data item to *queue*. There is an implicit memory allocation to create an additional temporary bookkeeping data structure from the calling thread's resource pool, which is automatically freed when the item is removed. The data itself is not copied.

Function properties (list may not be complete)

isr-ok

Parameters

- `queue` – Address of the queue.
- `data` – Address of the data item.

Return values

- `0` – on success
- `-ENOMEM` – if there isn't sufficient RAM in the caller's resource pool

void `k_queue_insert`(struct k_queue *queue, void *prev, void *data)

Inserts an element to a queue.

This routine inserts a data item to *queue* after previous item. A queue data item must be aligned on a word boundary, and the first word of the item is reserved for the kernel's use.

Function properties (list may not be complete)

isr-ok

Parameters

- `queue` – Address of the queue.
- `prev` – Address of the previous data item.
- `data` – Address of the data item.

int `k_queue_append_list`(struct k_queue *queue, void *head, void *tail)

Atomically append a list of elements to a queue.

This routine adds a list of data items to *queue* in one operation. The data items must be in a singly-linked list, with the first word in each data item pointing to the next data item; the list must be NULL-terminated.

Function properties (list may not be complete)

isr-ok

Parameters

- `queue` – Address of the queue.
- `head` – Pointer to first node in singly-linked list.
- `tail` – Pointer to last node in singly-linked list.

Return values

- `0` – on success
- `-EINVAL` – on invalid supplied data

int `k_queue_merge_slist`(struct k_queue *queue, *sys_slist_t* *list)

Atomically add a list of elements to a queue.

This routine adds a list of data items to *queue* in one operation. The data items must be in a singly-linked list implemented using a sys_slist_t object. Upon completion, the original list is empty.

Function properties (list may not be complete)

isr-ok

Parameters

- `queue` – Address of the queue.
- `list` – Pointer to sys_slist_t object.

Return values

- `0` – on success
- `-EINVAL` – on invalid data

void *`k_queue_get`(struct k_queue *queue, *k_timeout_t* timeout)

Get an element from a queue.

This routine removes first data item from *queue*. The first word of the data item is reserved for the kernel's use.

Function properties (list may not be complete)

isr-ok

Note

timeout must be set to K_NO_WAIT if called from ISR.

Parameters

- **queue** – Address of the queue.
- **timeout** – Waiting period to obtain a data item, or one of the special values K_NO_WAIT and K_FOREVER.

Returns

Address of the data item if successful; NULL if returned without waiting, or waiting period timed out.

bool **k_queue_remove**(struct k_queue *queue, void *data)

Remove an element from a queue.

This routine removes data item from *queue*. The first word of the data item is reserved for the kernel's use. Removing elements from k_queue rely on sys_slist_find_and_remove which is not a constant time operation.

Function properties (list may not be complete)

isr-ok

Note

timeout must be set to K_NO_WAIT if called from ISR.

Parameters

- **queue** – Address of the queue.
- **data** – Address of the data item.

Returns

true if data item was removed

bool **k_queue_unique_append**(struct k_queue *queue, void *data)

Append an element to a queue only if it's not present already.

This routine appends data item to *queue*. The first word of the data item is reserved for the kernel's use. Appending elements to k_queue relies on sys_slist_is_node_in_list which is not a constant time operation.

Function properties (list may not be complete)

isr-ok

Parameters

- **queue** – Address of the queue.
- **data** – Address of the data item.

Returns

true if data item was added, false if not

int k_queue_is_empty(struct k_queue *queue)

Query a queue to see if it has data available.

Note that the data might be already gone by the time this function returns if other threads are also trying to read from the queue.

Function properties (list may not be complete)

isr-ok

Parameters

- queue – Address of the queue.

Returns

Non-zero if the queue is empty.

Returns

0 if data is available.

void *k_queue_peek_head(struct k_queue *queue)

Peek element at the head of queue.

Return element from the head of queue without removing it.

Parameters

- queue – Address of the queue.

Returns

Head element, or NULL if queue is empty.

void *k_queue_peek_tail(struct k_queue *queue)

Peek element at the tail of queue.

Return element from the tail of queue without removing it.

Parameters

- queue – Address of the queue.

Returns

Tail element, or NULL if queue is empty.

FIFOs

A *FIFO* is a kernel object that implements a traditional first in, first out (FIFO) queue, allowing threads and ISRs to add and remove data items of any size.

- *Concepts*
- *Implementation*
  - *Defining a FIFO*
  - *Writing to a FIFO*
  - *Reading from a FIFO*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

Concepts Any number of FIFOs can be defined (limited only by available RAM). Each FIFO is referenced by its memory address.

A FIFO has the following key properties:

- A **queue** of data items that have been added but not yet removed. The queue is implemented as a simple linked list.

A FIFO must be initialized before it can be used. This sets its queue to empty.

FIFO data items must be aligned on a word boundary, as the kernel reserves the first word of an item for use as a pointer to the next data item in the queue. Consequently, a data item that holds N bytes of application data requires N+4 (or N+8) bytes of memory. There are no alignment or reserved space requirements for data items if they are added with *k_fifo_alloc_put()*, instead additional memory is temporarily allocated from the calling thread's resource pool.

Note

FIFO data items are restricted to single active instance across all FIFO data queues. Any attempt to re-add a FIFO data item to a queue before it has been removed from the queue to which it was previously added will result in undefined behavior.

A data item may be **added** to a FIFO by a thread or an ISR. The item is given directly to a waiting thread, if one exists; otherwise the item is added to the FIFO's queue. There is no limit to the number of items that may be queued.

A data item may be **removed** from a FIFO by a thread. If the FIFO's queue is empty a thread may choose to wait for a data item to be given. Any number of threads may wait on an empty FIFO simultaneously. When a data item is added, it is given to the highest priority thread that has waited longest.

Note

The kernel does allow an ISR to remove an item from a FIFO, however the ISR must not attempt to wait if the FIFO is empty.

If desired, **multiple data items** can be added to a FIFO in a single operation if they are chained together into a singly-linked list. This capability can be useful if multiple writers are adding sets of related data items to the FIFO, as it ensures the data items in each set are not interleaved with other data items. Adding multiple data items to a FIFO is also more efficient than adding them one at a time, and can be used to guarantee that anyone who removes the first data item in a set will be able to remove the remaining data items without waiting.

Implementation

Defining a FIFO A FIFO is defined using a variable of type `k_fifo`. It must then be initialized by calling *k_fifo_init()*.

The following code defines and initializes an empty FIFO.

```
struct k_fifo my_fifo;

k_fifo_init(&my_fifo);
```

Alternatively, an empty FIFO can be defined and initialized at compile time by calling *K_FIFO_DEFINE*.

The following code has the same effect as the code segment above.

```
K_FIFO_DEFINE(my_fifo);
```

Writing to a FIFO A data item is added to a FIFO by calling *k_fifo_put()*.

The following code builds on the example above, and uses the FIFO to send data to one or more consumer threads.

```
struct data_item_t {
    void *fifo_reserved;   /* 1st word reserved for use by FIFO */
    ...
};

struct data_item_t tx_data;

void producer_thread(int unused1, int unused2, int unused3)
{
    while (1) {
        /* create data item to send */
        tx_data = ...

        /* send data to consumers */
        k_fifo_put(&my_fifo, &tx_data);

        ...
    }
}
```

Additionally, a singly-linked list of data items can be added to a FIFO by calling *k_fifo_put_list()* or *k_fifo_put_slist()*.

Finally, a data item can be added to a FIFO with *k_fifo_alloc_put()*. With this API, there is no need to reserve space for the kernel's use in the data item, instead additional memory will be allocated from the calling thread's resource pool until the item is read.

Reading from a FIFO A data item is removed from a FIFO by calling *k_fifo_get()*.

The following code builds on the example above, and uses the FIFO to obtain data items from a producer thread, which are then processed in some manner.

```
void consumer_thread(int unused1, int unused2, int unused3)
{
    struct data_item_t  *rx_data;

    while (1) {
        rx_data = k_fifo_get(&my_fifo, K_FOREVER);

        /* process FIFO data item */
        ...
    }
}
```

Suggested Uses Use a FIFO to asynchronously transfer data items of arbitrary size in a "first in, first out" manner.

Configuration Options Related configuration options:

- None

API Reference

group `fifo_apis`

Defines

`k_fifo_init`(fifo)

Initialize a FIFO queue.

This routine initializes a FIFO queue, prior to its first use.

Parameters

- `fifo` – Address of the FIFO queue.

`k_fifo_cancel_wait`(fifo)

Cancel waiting on a FIFO queue.

This routine causes first thread pending on *fifo*, if any, to return from *k_fifo_get()* call with NULL value (as if timeout expired).

Function properties (list may not be complete)

isr-ok

Parameters

- `fifo` – Address of the FIFO queue.

`k_fifo_put`(fifo, data)

Add an element to a FIFO queue.

This routine adds a data item to *fifo*. A FIFO data item must be aligned on a word boundary, and the first word of the item is reserved for the kernel's use.

Function properties (list may not be complete)

isr-ok

Parameters

- `fifo` – Address of the FIFO.
- `data` – Address of the data item.

`k_fifo_alloc_put`(fifo, data)

Add an element to a FIFO queue.

This routine adds a data item to *fifo*. There is an implicit memory allocation to create an additional temporary bookkeeping data structure from the calling thread's resource pool, which is automatically freed when the item is removed. The data itself is not copied.

Function properties (list may not be complete)

isr-ok

Parameters

- `fifo` – Address of the FIFO.
- `data` – Address of the data item.

Return values

- `0` – on success
- `-ENOMEM` – if there isn't sufficient RAM in the caller's resource pool

`k_fifo_put_list`(fifo, head, tail)

Atomically add a list of elements to a FIFO.

This routine adds a list of data items to *fifo* in one operation. The data items must be in a singly-linked list, with the first word of each data item pointing to the next data item; the list must be NULL-terminated.

Function properties (list may not be complete)

isr-ok

Parameters

- `fifo` – Address of the FIFO queue.
- `head` – Pointer to first node in singly-linked list.
- `tail` – Pointer to last node in singly-linked list.

`k_fifo_put_slist`(fifo, list)

Atomically add a list of elements to a FIFO queue.

This routine adds a list of data items to *fifo* in one operation. The data items must be in a singly-linked list implemented using a sys_slist_t object. Upon completion, the sys_slist_t object is invalid and must be re-initialized via *sys_slist_init()*.

Function properties (list may not be complete)

isr-ok

Parameters

- `fifo` – Address of the FIFO queue.
- `list` – Pointer to sys_slist_t object.

`k_fifo_get`(fifo, timeout)

Get an element from a FIFO queue.

This routine removes a data item from *fifo* in a "first in, first out" manner. The first word of the data item is reserved for the kernel's use.

Function properties (list may not be complete)

isr-ok

Note

timeout must be set to K_NO_WAIT if called from ISR.

Parameters

- `fifo` – Address of the FIFO queue.
- `timeout` – Waiting period to obtain a data item, or one of the special values K_NO_WAIT and K_FOREVER.

Returns
Address of the data item if successful; NULL if returned without waiting, or waiting period timed out.

`k_fifo_is_empty`(fifo)

Query a FIFO queue to see if it has data available.

Note that the data might be already gone by the time this function returns if other threads is also trying to read from the FIFO.

Function properties (list may not be complete)
isr-ok

Parameters

- `fifo` – Address of the FIFO queue.

Returns
Non-zero if the FIFO queue is empty.

Returns
0 if data is available.

`k_fifo_peek_head`(fifo)

Peek element at the head of a FIFO queue.

Return element from the head of FIFO queue without removing it. A usecase for this is if elements of the FIFO object are themselves containers. Then on each iteration of processing, a head container will be peeked, and some data processed out of it, and only if the container is empty, it will be completely remove from the FIFO queue.

Parameters

- `fifo` – Address of the FIFO queue.

Returns
Head element, or NULL if the FIFO queue is empty.

`k_fifo_peek_tail`(fifo)

Peek element at the tail of FIFO queue.

Return element from the tail of FIFO queue (without removing it). A usecase for this is if elements of the FIFO queue are themselves containers. Then it may be useful to add more data to the last container in a FIFO queue.

Parameters

- `fifo` – Address of the FIFO queue.

Returns
Tail element, or NULL if a FIFO queue is empty.

`K_FIFO_DEFINE`(name)

Statically define and initialize a FIFO queue.

The FIFO queue can be accessed outside the module where it is defined using:

```
extern struct k_fifo <name>;
```

Parameters

- `name` – Name of the FIFO queue.

LIFOs

A *LIFO* is a kernel object that implements a traditional last in, first out (LIFO) queue, allowing threads and ISRs to add and remove data items of any size.

- *Concepts*
- *Implementation*
  - *Defining a LIFO*
  - *Writing to a LIFO*
  - *Reading from a LIFO*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

Concepts Any number of LIFOs can be defined (limited only by available RAM). Each LIFO is referenced by its memory address.

A LIFO has the following key properties:

- A **queue** of data items that have been added but not yet removed. The queue is implemented as a simple linked list.

A LIFO must be initialized before it can be used. This sets its queue to empty.

LIFO data items must be aligned on a word boundary, as the kernel reserves the first word of an item for use as a pointer to the next data item in the queue. Consequently, a data item that holds N bytes of application data requires N+4 (or N+8) bytes of memory. There are no alignment or reserved space requirements for data items if they are added with `k_lifo_alloc_put()`, instead additional memory is temporarily allocated from the calling thread's resource pool.

Note

LIFO data items are restricted to single active instance across all LIFO data queues. Any attempt to re-add a LIFO data item to a queue before it has been removed from the queue to which it was previously added will result in undefined behavior.

A data item may be **added** to a LIFO by a thread or an ISR. The item is given directly to a waiting thread, if one exists; otherwise the item is added to the LIFO's queue. There is no limit to the number of items that may be queued.

A data item may be **removed** from a LIFO by a thread. If the LIFO's queue is empty a thread may choose to wait for a data item to be given. Any number of threads may wait on an empty LIFO simultaneously. When a data item is added, it is given to the highest priority thread that has waited longest.

Note

The kernel does allow an ISR to remove an item from a LIFO, however the ISR must not attempt to wait if the LIFO is empty.

Implementation

Defining a LIFO A LIFO is defined using a variable of type `k_lifo`. It must then be initialized by calling *k_lifo_init()*.

The following defines and initializes an empty LIFO.

```
struct k_lifo my_lifo;

k_lifo_init(&my_lifo);
```

Alternatively, an empty LIFO can be defined and initialized at compile time by calling *K_LIFO_DEFINE*.

The following code has the same effect as the code segment above.

```
K_LIFO_DEFINE(my_lifo);
```

Writing to a LIFO A data item is added to a LIFO by calling *k_lifo_put()*.

The following code builds on the example above, and uses the LIFO to send data to one or more consumer threads.

```
struct data_item_t {
    void *LIFO_reserved;   /* 1st word reserved for use by LIFO */
    ...
};

struct data_item_t tx data;

void producer_thread(int unused1, int unused2, int unused3)
{
    while (1) {
        /* create data item to send */
        tx_data = ...

        /* send data to consumers */
        k_lifo_put(&my_lifo, &tx_data);

        ...
    }
}
```

A data item can be added to a LIFO with *k_lifo_alloc_put()*. With this API, there is no need to reserve space for the kernel's use in the data item, instead additional memory will be allocated from the calling thread's resource pool until the item is read.

Reading from a LIFO A data item is removed from a LIFO by calling *k_lifo_get()*.

The following code builds on the example above, and uses the LIFO to obtain data items from a producer thread, which are then processed in some manner.

```
void consumer_thread(int unused1, int unused2, int unused3)
{
    struct data_item_t  *rx_data;

    while (1) {
        rx_data = k_lifo_get(&my_lifo, K_FOREVER);

        /* process LIFO data item */
```

(continues on next page)

(continued from previous page)

```
        ...
    }
}
```

Suggested Uses Use a LIFO to asynchronously transfer data items of arbitrary size in a "last in, first out" manner.

Configuration Options Related configuration options:

- None.

API Reference

group `lifo_apis`

Defines

`k_lifo_init`(lifo)

Initialize a LIFO queue.

This routine initializes a LIFO queue object, prior to its first use.

Parameters

- `lifo` – Address of the LIFO queue.

`k_lifo_put`(lifo, data)

Add an element to a LIFO queue.

This routine adds a data item to *lifo*. A LIFO queue data item must be aligned on a word boundary, and the first word of the item is reserved for the kernel's use.

Function properties (list may not be complete)

isr-ok

Parameters

- `lifo` – Address of the LIFO queue.
- `data` – Address of the data item.

`k_lifo_alloc_put`(lifo, data)

Add an element to a LIFO queue.

This routine adds a data item to *lifo*. There is an implicit memory allocation to create an additional temporary bookkeeping data structure from the calling thread's resource pool, which is automatically freed when the item is removed. The data itself is not copied.

Function properties (list may not be complete)

isr-ok

Parameters

- `lifo` – Address of the LIFO.

- `data` – Address of the data item.

Return values

- `0` – on success
- `-ENOMEM` – if there isn't sufficient RAM in the caller's resource pool

`k_lifo_get`(lifo, timeout)

Get an element from a LIFO queue.

This routine removes a data item from *LIFO* in a "last in, first out" manner. The first word of the data item is reserved for the kernel's use.

Function properties (list may not be complete)

isr-ok

> **Note**
>
> *timeout* must be set to K_NO_WAIT if called from ISR.

Parameters

- `lifo` – Address of the LIFO queue.
- `timeout` – Waiting period to obtain a data item, or one of the special values K_NO_WAIT and K_FOREVER.

Returns

Address of the data item if successful; NULL if returned without waiting, or waiting period timed out.

`K_LIFO_DEFINE`(name)

Statically define and initialize a LIFO queue.

The LIFO queue can be accessed outside the module where it is defined using:

```
extern struct k_lifo <name>;
```

Parameters

- `name` – Name of the fifo.

Stacks

A *stack* is a kernel object that implements a traditional last in, first out (LIFO) queue, allowing threads and ISRs to add and remove a limited number of integer data values.

- *Concepts*
- *Implementation*
  - *Defining a Stack*
  - *Pushing to a Stack*
  - *Popping from a Stack*
- *Suggested Uses*

- *Configuration Options*
- *API Reference*

Concepts Any number of stacks can be defined (limited only by available RAM). Each stack is referenced by its memory address.

A stack has the following key properties:

- A **queue** of integer data values that have been added but not yet removed. The queue is implemented using an array of stack_data_t values and must be aligned on a native word boundary. The stack_data_t type corresponds to the native word size i.e. 32 bits or 64 bits depending on the CPU architecture and compilation mode.
- A **maximum quantity** of data values that can be queued in the array.

A stack must be initialized before it can be used. This sets its queue to empty.

A data value can be **added** to a stack by a thread or an ISR. The value is given directly to a waiting thread, if one exists; otherwise the value is added to the LIFO's queue.

Note

If `CONFIG_NO_RUNTIME_CHECKS` is enabled, the kernel will *not* detect and prevent attempts to add a data value to a stack that has already reached its maximum quantity of queued values. Adding a data value to a stack that is already full will result in array overflow, and lead to unpredictable behavior.

A data value may be **removed** from a stack by a thread. If the stack's queue is empty a thread may choose to wait for it to be given. Any number of threads may wait on an empty stack simultaneously. When a data item is added, it is given to the highest priority thread that has waited longest.

Note

The kernel does allow an ISR to remove an item from a stack, however the ISR must not attempt to wait if the stack is empty.

Implementation

Defining a Stack A stack is defined using a variable of type `k_stack`. It must then be initialized by calling *`k_stack_init()`* or *`k_stack_alloc_init()`*. In the latter case, a buffer is not provided and it is instead allocated from the calling thread's resource pool.

The following code defines and initializes an empty stack capable of holding up to ten word-sized data values.

```
#define MAX_ITEMS 10

stack_data_t my_stack_array[MAX_ITEMS];
struct k_stack my_stack;

k_stack_init(&my_stack, my_stack_array, MAX_ITEMS);
```

Alternatively, a stack can be defined and initialized at compile time by calling *K_STACK_DEFINE*.

The following code has the same effect as the code segment above. Observe that the macro defines both the stack and its array of data values.

```
K_STACK_DEFINE(my_stack, MAX_ITEMS);
```

Pushing to a Stack A data item is added to a stack by calling *k_stack_push()*.

The following code builds on the example above, and shows how a thread can create a pool of data structures by saving their memory addresses in a stack.

```
/* define array of data structures */
struct my_buffer_type {
    int field1;
    ...
    };
struct my_buffer_type my_buffers[MAX_ITEMS];

/* save address of each data structure in a stack */
for (int i = 0; i < MAX_ITEMS; i++) {
    k_stack_push(&my_stack, (stack_data_t)&my_buffers[i]);
}
```

Popping from a Stack A data item is taken from a stack by calling *k_stack_pop()*.

The following code builds on the example above, and shows how a thread can dynamically allocate an unused data structure. When the data structure is no longer required, the thread must push its address back on the stack to allow the data structure to be reused.

```
struct my_buffer_type *new_buffer;

k_stack_pop(&buffer_stack, (stack_data_t *)&new_buffer, K_FOREVER);
new_buffer->field1 = ...
```

Suggested Uses Use a stack to store and retrieve integer data values in a "last in, first out" manner, when the maximum number of stored items is known.

Configuration Options Related configuration options:

- None.

API Reference

group `stack_apis`

Defines

`K_STACK_DEFINE`(name, stack_num_entries)

Statically define and initialize a stack.

The stack can be accessed outside the module where it is defined using:

```
extern struct k_stack <name>;
```

Parameters

- `name` – Name of the stack.
- `stack_num_entries` – Maximum number of values that can be stacked.

Functions

void `k_stack_init`(struct k_stack *stack, stack_data_t *buffer, uint32_t num_entries)

Initialize a stack.

This routine initializes a stack object, prior to its first use.

Parameters

- `stack` – Address of the stack.
- `buffer` – Address of array used to hold stacked values.
- `num_entries` – Maximum number of values that can be stacked.

int32_t `k_stack_alloc_init`(struct k_stack *stack, uint32_t num_entries)

Initialize a stack.

This routine initializes a stack object, prior to its first use. Internal buffers will be allocated from the calling thread's resource pool. This memory will be released if *k_stack_cleanup()* is called, or userspace is enabled and the stack object loses all references to it.

Parameters

- `stack` – Address of the stack.
- `num_entries` – Maximum number of values that can be stacked.

Returns

-ENOMEM if memory couldn't be allocated

int `k_stack_cleanup`(struct k_stack *stack)

Release a stack's allocated buffer.

If a stack object was given a dynamically allocated buffer via *k_stack_alloc_init()*, this will free it. This function does nothing if the buffer wasn't dynamically allocated.

Parameters

- `stack` – Address of the stack.

Return values

- `0` – on success
- `-EAGAIN` – when object is still in use

int `k_stack_push`(struct k_stack *stack, stack_data_t data)

Push an element onto a stack.

This routine adds a stack_data_t value *data* to *stack*.

Function properties (list may not be complete)

isr-ok

Parameters

- `stack` – Address of the stack.
- `data` – Value to push onto the stack.

Return values

- `0` – on success
- `-ENOMEM` – if stack is full

int `k_stack_pop`(struct k_stack *stack, stack_data_t *data, *k_timeout_t* timeout)

Pop an element from a stack.

This routine removes a stack_data_t value from *stack* in a "last in,

first out" manner and stores the value in *data*.

Function properties (list may not be complete)

isr-ok

> **Note**
>
> *timeout* must be set to K_NO_WAIT if called from ISR.

Parameters

- `stack` – Address of the stack.
- `data` – Address of area to hold the value popped from the stack.
- `timeout` – Waiting period to obtain a value, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- `0` – Element popped from stack.
- `-EBUSY` – Returned without waiting.
- `-EAGAIN` – Waiting period timed out.

Message Queues

A *message queue* is a kernel object that implements a simple message queue, allowing threads and ISRs to asynchronously send and receive fixed-size data items.

- *Concepts*
- *Implementation*
  - *Defining a Message Queue*
  - *Writing to a Message Queue*
  - *Reading from a Message Queue*
  - *Peeking into a Message Queue*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

Concepts Any number of message queues can be defined (limited only by available RAM). Each message queue is referenced by its memory address.

A message queue has the following key properties:

- A **ring buffer** of data items that have been sent but not yet received.
- A **data item size**, measured in bytes.
- A **maximum quantity** of data items that can be queued in the ring buffer.

A message queue must be initialized before it can be used. This sets its ring buffer to empty.

A data item can be **sent** to a message queue by a thread or an ISR. The data item pointed at by the sending thread is copied to a waiting thread, if one exists; otherwise the item is copied to the message queue's ring buffer, if space is available. In either case, the size of the data area being sent *must* equal the message queue's data item size.

If a thread attempts to send a data item when the ring buffer is full, the sending thread may choose to wait for space to become available. Any number of sending threads may wait simultaneously when the ring buffer is full; when space becomes available it is given to the highest priority sending thread that has waited the longest.

A data item can be **received** from a message queue by a thread. The data item is copied to the area specified by the receiving thread; the size of the receiving area *must* equal the message queue's data item size.

If a thread attempts to receive a data item when the ring buffer is empty, the receiving thread may choose to wait for a data item to be sent. Any number of receiving threads may wait simultaneously when the ring buffer is empty; when a data item becomes available it is given to the highest priority receiving thread that has waited the longest.

A thread can also **peek** at the message on the head of a message queue without removing it from the queue. The data item is copied to the area specified by the receiving thread; the size of the receiving area *must* equal the message queue's data item size.

Note

The kernel does allow an ISR to receive an item from a message queue, however the ISR must not attempt to wait if the message queue is empty.

Note

Alignment of the message queue's ring buffer is not necessary. The underlying implementation uses `memcpy()` (which is alignment-agnostic) and does not expose any internal pointers.

Implementation

Defining a Message Queue A message queue is defined using a variable of type *k_msgq*. It must then be initialized by calling *k_msgq_init()*.

The following code defines and initializes an empty message queue that is capable of holding 10 items, each of which is 12 bytes long.

```
struct data_item_type {
    uint32_t field1;
    uint32_t field2;
    uint32_t field3;
};
```

(continues on next page)

(continued from previous page)

```
char my_msgq_buffer[10 * sizeof(struct data_item_type)];
struct k_msgq my_msgq;

k_msgq_init(&my_msgq, my_msgq_buffer, sizeof(struct data_item_type), 10);
```

Alternatively, a message queue can be defined and initialized at compile time by calling *K_MSGQ_DEFINE*.

The following code has the same effect as the code segment above. Observe that the macro defines both the message queue and its buffer.

```
K_MSGQ_DEFINE(my_msgq, sizeof(struct data_item_type), 10, 1);
```

Writing to a Message Queue A data item is added to a message queue by calling *k_msgq_put()*.

The following code builds on the example above, and uses the message queue to pass data items from a producing thread to one or more consuming threads. If the message queue fills up because the consumers can't keep up, the producing thread throws away all existing data so the newer data can be saved.

```
void producer_thread(void)
{
    struct data_item_type data;

    while (1) {
        /* create data item to send (e.g. measurement, timestamp, ...) */
        data = ...

        /* send data to consumers */
        while (k_msgq_put(&my_msgq, &data, K_NO_WAIT) != 0) {
            /* message queue is full: purge old data & try again */
            k_msgq_purge(&my_msgq);
        }

        /* data item was successfully added to message queue */
    }
}
```

Reading from a Message Queue A data item is taken from a message queue by calling *k_msgq_get()*.

The following code builds on the example above, and uses the message queue to process data items generated by one or more producing threads. Note that the return value of *k_msgq_get()* should be tested as -ENOMSG can be returned due to *k_msgq_purge()*.

```
void consumer_thread(void)
{
    struct data_item_type data;

    while (1) {
        /* get a data item */
        k_msgq_get(&my_msgq, &data, K_FOREVER);

        /* process data item */
        ...
    }
}
```

Peeking into a Message Queue A data item is read from a message queue by calling *k_msgq_peek()*.

The following code peeks into the message queue to read the data item at the head of the queue that is generated by one or more producing threads.

```
void consumer_thread(void)
{
    struct data_item_type data;

    while (1) {
        /* read a data item by peeking into the queue */
        k_msgq_peek(&my_msgq, &data);

        /* process data item */
        ...
    }
}
```

Suggested Uses Use a message queue to transfer small data items between threads in an asynchronous manner.

Note

A message queue can be used to transfer large data items, if desired. However, this can increase interrupt latency as interrupts are locked while a data item is written or read. The time to write or read a data item increases linearly with its size since the item is copied in its entirety to or from the buffer in memory. For this reason, it is usually preferable to transfer large data items by exchanging a pointer to the data item, rather than the data item itself.

A synchronous transfer can be achieved by using the kernel's mailbox object type.

Configuration Options Related configuration options:

- None.

API Reference

group `msgq_apis`

Defines

`K_MSGQ_FLAG_ALLOC`

`K_MSGQ_DEFINE`(q_name, q_msg_size, q_max_msgs, q_align)

Statically define and initialize a message queue.

The message queue's ring buffer contains space for *q_max_msgs* messages, each of which is *q_msg_size* bytes long. Alignment of the message queue's ring buffer is not necessary, setting *q_align* to 1 is sufficient.

The message queue can be accessed outside the module where it is defined using:

```
extern struct k_msgq <name>;
```

Parameters

- `q_name` – Name of the message queue.
- `q_msg_size` – Message size (in bytes).
- `q_max_msgs` – Maximum number of messages that can be queued.
- `q_align` – Alignment of the message queue's ring buffer (power of 2).

Functions

void **k_msgq_init**(struct *k_msgq* *msgq, char *buffer, size_t msg_size, uint32_t max_msgs)

Initialize a message queue.

This routine initializes a message queue object, prior to its first use.

The message queue's ring buffer must contain space for *max_msgs* messages, each of which is *msg_size* bytes long. Alignment of the message queue's ring buffer is not necessary.

Parameters

- `msgq` – Address of the message queue.
- `buffer` – Pointer to ring buffer that holds queued messages.
- `msg_size` – Message size (in bytes).
- `max_msgs` – Maximum number of messages that can be queued.

int **k_msgq_alloc_init**(struct *k_msgq* *msgq, size_t msg_size, uint32_t max_msgs)

Initialize a message queue.

This routine initializes a message queue object, prior to its first use, allocating its internal ring buffer from the calling thread's resource pool.

Memory allocated for the ring buffer can be released by calling *k_msgq_cleanup()*, or if userspace is enabled and the msgq object loses all of its references.

Parameters

- `msgq` – Address of the message queue.
- `msg_size` – Message size (in bytes).
- `max_msgs` – Maximum number of messages that can be queued.

Returns

0 on success, -ENOMEM if there was insufficient memory in the thread's resource pool, or -EINVAL if the size parameters cause an integer overflow.

int **k_msgq_cleanup**(struct *k_msgq* *msgq)

Release allocated buffer for a queue.

Releases memory allocated for the ring buffer.

Parameters

- `msgq` – message queue to cleanup

Return values

- `0` – on success
- `-EBUSY` – Queue not empty

int k_msgq_put(struct *k_msgq* *msgq, const void *data, *k_timeout_t* timeout)

Send a message to a message queue.

This routine sends a message to message queue *q*.

Function properties (list may not be complete)

isr-ok

> **Note**
>
> The message content is copied from *data* into *msgq* and the *data* pointer is not retained, so the message content will not be modified by this function.

Parameters

- `msgq` – Address of the message queue.
- `data` – Pointer to the message.
- `timeout` – Waiting period to add the message, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- `0` – Message sent.
- `-ENOMSG` – Returned without waiting or queue purged.
- `-EAGAIN` – Waiting period timed out.

int k_msgq_get(struct *k_msgq* *msgq, void *data, *k_timeout_t* timeout)

Receive a message from a message queue.

This routine receives a message from message queue *q* in a “first in,

first out” manner.

Function properties (list may not be complete)

isr-ok

> **Note**
>
> *timeout* must be set to K_NO_WAIT if called from ISR.

Parameters

- `msgq` – Address of the message queue.
- `data` – Address of area to hold the received message.
- `timeout` – Waiting period to receive the message, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- `0` – Message received.
- `-ENOMSG` – Returned without waiting.
- `-EAGAIN` – Waiting period timed out.

int k_msgq_peek(struct *k_msgq* *msgq, void *data)

Peek/read a message from a message queue.

This routine reads a message from message queue *q* in a "first in,

first out" manner and leaves the message in the queue.

Function properties (list may not be complete)

isr-ok

Parameters

- msgq – Address of the message queue.
- data – Address of area to hold the message read from the queue.

Return values

- 0 – Message read.
- -ENOMSG – Returned when the queue has no message.

int k_msgq_peek_at(struct *k_msgq* *msgq, void *data, uint32_t idx)

Peek/read a message from a message queue at the specified index.

This routine reads a message from message queue at the specified index and leaves the message in the queue. k_msgq_peek_at(msgq, data, 0) is equivalent to k_msgq_peek(msgq, data)

Function properties (list may not be complete)

isr-ok

Parameters

- msgq – Address of the message queue.
- data – Address of area to hold the message read from the queue.
- idx – Message queue index at which to peek

Return values

- 0 – Message read.
- -ENOMSG – Returned when the queue has no message at index.

void k_msgq_purge(struct *k_msgq* *msgq)

Purge a message queue.

This routine discards all unreceived messages in a message queue's ring buffer. Any threads that are blocked waiting to send a message to the message queue are unblocked and see an -ENOMSG error code.

Parameters

- msgq – Address of the message queue.

uint32_t k_msgq_num_free_get(struct *k_msgq* *msgq)

Get the amount of free space in a message queue.

This routine returns the number of unused entries in a message queue's ring buffer.

Parameters

- msgq – Address of the message queue.

Returns
Number of unused ring buffer entries.

void **k_msgq_get_attrs**(struct *k_msgq* *msgq, struct *k_msgq_attrs* *attrs)
Get basic attributes of a message queue.

This routine fetches basic attributes of message queue into attr argument.

Parameters

- **msgq** – Address of the message queue.
- **attrs** – pointer to message queue attribute structure.

uint32_t **k_msgq_num_used_get**(struct *k_msgq* *msgq)
Get the number of messages in a message queue.

This routine returns the number of messages in a message queue's ring buffer.

Parameters

- **msgq** – Address of the message queue.

Returns
Number of messages.

struct **k_msgq**
#include <*kernel.h*> Message Queue Structure.

Public Members

_wait_q_t **wait_q**
Message queue wait queue.

struct *k_spinlock* **lock**
Lock.

size_t **msg_size**
Message size.

uint32_t **max_msgs**
Maximal number of messages.

char ***buffer_start**
Start of message buffer.

char ***buffer_end**
End of message buffer.

char ***read_ptr**
Read pointer.

char ***write_ptr**
Write pointer.

uint32_t **used_msgs**
Number of used messages.

uint8_t **flags**
Message queue.

struct **k_msgq_attrs**
#include <*kernel.h*> Message Queue Attributes.

Public Members

size_t **msg_size**
Message Size.

uint32_t **max_msgs**
Maximal number of messages.

uint32_t **used_msgs**
Used messages.

Mailboxes

A *mailbox* is a kernel object that provides enhanced message queue capabilities that go beyond the capabilities of a message queue object. A mailbox allows threads to send and receive messages of any size synchronously or asynchronously.

- *Concepts*
  - *Message Format*
  - *Message Lifecycle*
  - *Thread Compatibility*
  - *Message Flow Control*
- *Implementation*
  - *Defining a Mailbox*
  - *Message Descriptors*
  - *Sending a Message*
  - *Receiving a Message*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

Concepts Any number of mailboxes can be defined (limited only by available RAM). Each mailbox is referenced by its memory address.

A mailbox has the following key properties:

- A **send queue** of messages that have been sent but not yet received.
- A **receive queue** of threads that are waiting to receive a message.

A mailbox must be initialized before it can be used. This sets both of its queues to empty.

A mailbox allows threads, but not ISRs, to exchange messages. A thread that sends a message is known as the **sending thread**, while a thread that receives the message is known as the **receiving thread**. Each message may be received by only one thread (i.e. point-to-multipoint and broadcast messaging is not supported).

Messages exchanged using a mailbox are handled non-anonymously, allowing both threads participating in an exchange to know (and even specify) the identity of the other thread.

Message Format A **message descriptor** is a data structure that specifies where a message's data is located, and how the message is to be handled by the mailbox. Both the sending thread and the receiving thread supply a message descriptor when accessing a mailbox. The mailbox uses the message descriptors to perform a message exchange between compatible sending and receiving threads. The mailbox also updates certain message descriptor fields during the exchange, allowing both threads to know what has occurred.

A mailbox message contains zero or more bytes of **message data**. The size and format of the message data is application-defined, and can vary from one message to the next.

A **message buffer** is an area of memory provided by the thread that sends or receives the message data. An array or structure variable can often be used for this purpose.

A message that has neither form of message data is called an **empty message**.

> **Note**
>
> A message whose message buffer exists, but contains zero bytes of actual data, is *not* an empty message.

Message Lifecycle The life cycle of a message is straightforward. A message is created when it is given to a mailbox by the sending thread. The message is then owned by the mailbox until it is given to a receiving thread. The receiving thread may retrieve the message data when it receives the message from the mailbox, or it may perform data retrieval during a second, subsequent mailbox operation. Only when data retrieval has occurred is the message deleted by the mailbox.

Thread Compatibility A sending thread can specify the address of the thread to which the message is sent, or send it to any thread by specifying `K_ANY`. Likewise, a receiving thread can specify the address of the thread from which it wishes to receive a message, or it can receive a message from any thread by specifying `K_ANY`. A message is exchanged only when the requirements of both the sending thread and receiving thread are satisfied; such threads are said to be **compatible**.

For example, if thread A sends a message to thread B (and only thread B) it will be received by thread B if thread B tries to receive a message from thread A or if thread B tries to receive from any thread. The exchange will not occur if thread B tries to receive a message from thread C. The message can never be received by thread C, even if it tries to receive a message from thread A (or from any thread).

Message Flow Control Mailbox messages can be exchanged **synchronously** or **asynchronously**. In a synchronous exchange, the sending thread blocks until the message has been fully processed by the receiving thread. In an asynchronous exchange, the sending thread does not wait until the message has been received by another thread before continuing; this allows the sending thread to do other work (such as gather data that will be used in the next message) *before* the message is given to a receiving thread and fully processed. The technique used for a given message exchange is determined by the sending thread.

The synchronous exchange technique provides an implicit form of flow control, preventing a sending thread from generating messages faster than they can be consumed by receiving threads. The asynchronous exchange technique provides an explicit form of flow control, which allows a sending thread to determine if a previously sent message still exists before sending a subsequent message.

Implementation

Defining a Mailbox A mailbox is defined using a variable of type *k_mbox*. It must then be initialized by calling *k_mbox_init()*.

The following code defines and initializes an empty mailbox.

```
struct k_mbox my_mailbox;

k_mbox_init(&my_mailbox);
```

Alternatively, a mailbox can be defined and initialized at compile time by calling *K_MBOX_DEFINE*.

The following code has the same effect as the code segment above.

```
K_MBOX_DEFINE(my_mailbox);
```

Message Descriptors A message descriptor is a structure of type *k_mbox_msg*. Only the fields listed below should be used; any other fields are for internal mailbox use only.

info
: A 32-bit value that is exchanged by the message sender and receiver, and whose meaning is defined by the application. This exchange is bi-directional, allowing the sender to pass a value to the receiver during any message exchange, and allowing the receiver to pass a value to the sender during a synchronous message exchange.

size
: The message data size, in bytes. Set it to zero when sending an empty message, or when sending a message buffer with no actual data. When receiving a message, set it to the maximum amount of data desired, or to zero if the message data is not wanted. The mailbox updates this field with the actual number of data bytes exchanged once the message is received.

tx_data
: A pointer to the sending thread's message buffer. Set it to `NULL` when sending an empty message. Leave this field uninitialized when receiving a message.

tx_target_thread
: The address of the desired receiving thread. Set it to `K_ANY` to allow any thread to receive the message. Leave this field uninitialized when receiving a message. The mailbox updates this field with the actual receiver's address once the message is received.

rx_source_thread
: The address of the desired sending thread. Set it to `K_ANY` to receive a message sent by any thread. Leave this field uninitialized when sending a message. The mailbox updates this field with the actual sender's address when the message is put into the mailbox.

Sending a Message A thread sends a message by first creating its message data, if any.

Next, the sending thread creates a message descriptor that characterizes the message to be sent, as described in the previous section.

Finally, the sending thread calls a mailbox send API to initiate the message exchange. The message is immediately given to a compatible receiving thread, if one is currently waiting. Otherwise, the message is added to the mailbox's send queue.

Any number of messages may exist simultaneously on a send queue. The messages in the send queue are sorted according to the priority of the sending thread. Messages of equal priority are sorted so that the oldest message can be received first.

For a synchronous send operation, the operation normally completes when a receiving thread has both received the message and retrieved the message data. If the message is not received before the waiting period specified by the sending thread is reached, the message is removed from the mailbox's send queue and the send operation fails. When a send operation completes successfully the sending thread can examine the message descriptor to determine which thread received the message, how much data was exchanged, and the application-defined info value supplied by the receiving thread.

Note

A synchronous send operation may block the sending thread indefinitely, even when the thread specifies a maximum waiting period. The waiting period only limits how long the mailbox waits before the message is received by another thread. Once a message is received there is *no* limit to the time the receiving thread may take to retrieve the message data and unblock the sending thread.

For an asynchronous send operation, the operation always completes immediately. This allows the sending thread to continue processing regardless of whether the message is given to a receiving thread immediately or added to the send queue. The sending thread may optionally specify a semaphore that the mailbox gives when the message is deleted by the mailbox, for example, when the message has been received and its data retrieved by a receiving thread. The use of a semaphore allows the sending thread to easily implement a flow control mechanism that ensures that the mailbox holds no more than an application-specified number of messages from a sending thread (or set of sending threads) at any point in time.

Note

A thread that sends a message asynchronously has no way to determine which thread received the message, how much data was exchanged, or the application-defined info value supplied by the receiving thread.

Sending an Empty Message This code uses a mailbox to synchronously pass 4 byte random values to any consuming thread that wants one. The message "info" field is large enough to carry the information being exchanged, so the data portion of the message isn't used.

```
void producer_thread(void)
{
    struct k_mbox_msg send_msg;

    while (1) {

        /* generate random value to send */
        uint32_t random_value = sys_rand32_get();
```

(continues on next page)

(continued from previous page)

```
        /* prepare to send empty message */
        send_msg.info = random_value;
        send_msg.size = 0;
        send_msg.tx_data = NULL;
        send_msg.tx_target_thread = K_ANY;

        /* send message and wait until a consumer receives it */
        k_mbox_put(&my_mailbox, &send_msg, K_FOREVER);
    }
}
```

Sending Data Using a Message Buffer This code uses a mailbox to synchronously pass variable-sized requests from a producing thread to any consuming thread that wants it. The message "info" field is used to exchange information about the maximum size message buffer that each thread can handle.

```
void producer_thread(void)
{
    char buffer[100];
    int buffer_bytes_used;

    struct k_mbox_msg send_msg;

    while (1) {

        /* generate data to send */
        ...
        buffer_bytes_used = ... ;
        memcpy(buffer, source, buffer_bytes_used);

        /* prepare to send message */
        send_msg.info = buffer_bytes_used;
        send_msg.size = buffer_bytes_used;
        send_msg.tx_data = buffer;
        send_msg.tx_target_thread = K_ANY;

        /* send message and wait until a consumer receives it */
        k_mbox_put(&my_mailbox, &send_msg, K_FOREVER);

        /* info, size, and tx_target_thread fields have been updated */

        /* verify that message data was fully received */
        if (send_msg.size < buffer_bytes_used) {
            printf("some message data dropped during transfer!");
            printf("receiver only had room for %d bytes", send_msg.info);
        }
    }
}
```

Receiving a Message A thread receives a message by first creating a message descriptor that characterizes the message it wants to receive. It then calls one of the mailbox receive APIs. The mailbox searches its send queue and takes the message from the first compatible thread it finds. If no compatible thread exists, the receiving thread may choose to wait for one. If no compatible thread appears before the waiting period specified by the receiving thread is reached, the receive operation fails. Once a receive operation completes successfully the receiving thread can examine the message descriptor to determine which thread sent the message, how much data was exchanged, and the application-defined info value supplied by the sending thread.

Any number of receiving threads may wait simultaneously on a mailboxes' receive queue. The threads are sorted according to their priority; threads of equal priority are sorted so that the one that started waiting first can receive a message first.

Note

Receiving threads do not always receive messages in a first in, first out (FIFO) order, due to the thread compatibility constraints specified by the message descriptors. For example, if thread A waits to receive a message only from thread X and then thread B waits to receive a message from thread Y, an incoming message from thread Y to any thread will be given to thread B and thread A will continue to wait.

The receiving thread controls both the quantity of data it retrieves from an incoming message and where the data ends up. The thread may choose to take all of the data in the message, to take only the initial part of the data, or to take no data at all. Similarly, the thread may choose to have the data copied into a message buffer of its choice.

The following sections outline various approaches a receiving thread may use when retrieving message data.

Retrieving Data at Receive Time The most straightforward way for a thread to retrieve message data is to specify a message buffer when the message is received. The thread indicates both the location of the message buffer (which must not be NULL) and its size.

The mailbox copies the message's data to the message buffer as part of the receive operation. If the message buffer is not big enough to contain all of the message's data, any uncopied data is lost. If the message is not big enough to fill all of the buffer with data, the unused portion of the message buffer is left unchanged. In all cases the mailbox updates the receiving thread's message descriptor to indicate how many data bytes were copied (if any).

The immediate data retrieval technique is best suited for small messages where the maximum size of a message is known in advance.

The following code uses a mailbox to process variable-sized requests from any producing thread, using the immediate data retrieval technique. The message "info" field is used to exchange information about the maximum size message buffer that each thread can handle.

```
void consumer_thread(void)
{
    struct k_mbox_msg recv_msg;
    char buffer[100];

    int i;
    int total;

    while (1) {
        /* prepare to receive message */
        recv_msg.info = 100;
        recv_msg.size = 100;
        recv_msg.rx_source_thread = K_ANY;

        /* get a data item, waiting as long as needed */
        k_mbox_get(&my_mailbox, &recv_msg, buffer, K_FOREVER);

        /* info, size, and rx_source_thread fields have been updated */

        /* verify that message data was fully received */
        if (recv_msg.info != recv_msg.size) {
            printf("some message data dropped during transfer!");
            printf("sender tried to send %d bytes", recv_msg.info);
```

(continues on next page)

(continued from previous page)

```
        }

        /* compute sum of all message bytes (from 0 to 100 of them) */
        total = 0;
        for (i = 0; i < recv_msg.size; i++) {
            total += buffer[i];
        }
    }
}
```

Retrieving Data Later Using a Message Buffer A receiving thread may choose to defer message data retrieval at the time the message is received, so that it can retrieve the data into a message buffer at a later time. The thread does this by specifying a message buffer location of NULL and a size indicating the maximum amount of data it is willing to retrieve later.

The mailbox does not copy any message data as part of the receive operation. However, the mailbox still updates the receiving thread's message descriptor to indicate how many data bytes are available for retrieval.

The receiving thread must then respond as follows:

- If the message descriptor size is zero, then either the sender's message contained no data or the receiving thread did not want to receive any data. The receiving thread does not need to take any further action, since the mailbox has already completed data retrieval and deleted the message.
- If the message descriptor size is non-zero and the receiving thread still wants to retrieve the data, the thread must call *k_mbox_data_get()* and supply a message buffer large enough to hold the data. The mailbox copies the data into the message buffer and deletes the message.
- If the message descriptor size is non-zero and the receiving thread does *not* want to retrieve the data, the thread must call *k_mbox_data_get()* and specify a message buffer of NULL. The mailbox deletes the message without copying the data.

The subsequent data retrieval technique is suitable for applications where immediate retrieval of message data is undesirable. For example, it can be used when memory limitations make it impractical for the receiving thread to always supply a message buffer capable of holding the largest possible incoming message.

The following code uses a mailbox's deferred data retrieval mechanism to get message data from a producing thread only if the message meets certain criteria, thereby eliminating unneeded data copying. The message "info" field supplied by the sender is used to classify the message.

```
void consumer_thread(void)
{
    struct k_mbox_msg recv_msg;
    char buffer[10000];

    while (1) {
        /* prepare to receive message */
        recv_msg.size = 10000;
        recv_msg.rx_source_thread = K_ANY;

        /* get message, but not its data */
        k_mbox_get(&my_mailbox, &recv_msg, NULL, K_FOREVER);

        /* get message data for only certain types of messages */
        if (is_message_type_ok(recv_msg.info)) {
            /* retrieve message data and delete the message */
            k_mbox_data_get(&recv_msg, buffer);
```

(continues on next page)

(continued from previous page)

```
            /* process data in "buffer" */
            ...
        } else {
            /* ignore message data and delete the message */
            k_mbox_data_get(&recv_msg, NULL);
        }
    }
}
```

Suggested Uses Use a mailbox to transfer data items between threads whenever the capabilities of a message queue are insufficient.

Configuration Options Related configuration options:

- CONFIG_NUM_MBOX_ASYNC_MSGS

API Reference

group mailbox_apis

Defines

K_MBOX_DEFINE(name)

Statically define and initialize a mailbox.

The mailbox is to be accessed outside the module where it is defined using:

```
extern struct k_mbox <name>;
```

Parameters

- **name** – Name of the mailbox.

Functions

void k_mbox_init(struct *k_mbox* *mbox)

Initialize a mailbox.

This routine initializes a mailbox object, prior to its first use.

Parameters

- **mbox** – Address of the mailbox.

int k_mbox_put(struct *k_mbox* *mbox, struct *k_mbox_msg* *tx_msg, *k_timeout_t* timeout)

Send a mailbox message in a synchronous manner.

This routine sends a message to *mbox* and waits for a receiver to both receive and process it. The message data may be in a buffer or non-existent (i.e. an empty message).

Parameters

- **mbox** – Address of the mailbox.
- **tx_msg** – Address of the transmit message descriptor.

- **timeout** – Waiting period for the message to be received, or one of the special values K_NO_WAIT and K_FOREVER. Once the message has been received, this routine waits as long as necessary for the message to be completely processed.

Return values

- **0** – Message sent.
- **-ENOMSG** – Returned without waiting.
- **-EAGAIN** – Waiting period timed out.

void **k_mbox_async_put**(struct *k_mbox* *mbox, struct *k_mbox_msg* *tx_msg, struct k_sem *sem)

Send a mailbox message in an asynchronous manner.

This routine sends a message to *mbox* without waiting for a receiver to process it. The message data may be in a buffer or non-existent (i.e. an empty message). Optionally, the semaphore *sem* will be given when the message has been both received and completely processed by the receiver.

Parameters

- **mbox** – Address of the mailbox.
- **tx_msg** – Address of the transmit message descriptor.
- **sem** – Address of a semaphore, or NULL if none is needed.

int **k_mbox_get**(struct *k_mbox* *mbox, struct *k_mbox_msg* *rx_msg, void *buffer, *k_timeout_t* timeout)

Receive a mailbox message.

This routine receives a message from *mbox*, then optionally retrieves its data and disposes of the message.

Parameters

- **mbox** – Address of the mailbox.
- **rx_msg** – Address of the receive message descriptor.
- **buffer** – Address of the buffer to receive data, or NULL to defer data retrieval and message disposal until later.
- **timeout** – Waiting period for a message to be received, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- **0** – Message received.
- **-ENOMSG** – Returned without waiting.
- **-EAGAIN** – Waiting period timed out.

void **k_mbox_data_get**(struct *k_mbox_msg* *rx_msg, void *buffer)

Retrieve mailbox message data into a buffer.

This routine completes the processing of a received message by retrieving its data into a buffer, then disposing of the message.

Alternatively, this routine can be used to dispose of a received message without retrieving its data.

Parameters

- **rx_msg** – Address of the receive message descriptor.

- **buffer** – Address of the buffer to receive data, or NULL to discard the data.

struct **k_mbox_msg**

#include <*kernel.h*> Mailbox Message Structure.

Public Members

size_t **size**

size of message (in bytes)

uint32_t **info**

application-defined information value

void ***tx_data**

sender's message data buffer

k_tid_t **rx_source_thread**

source thread id

k_tid_t **tx_target_thread**

target thread id

struct **k_mbox**

#include <*kernel.h*> Mailbox Structure.

Public Members

_wait_q_t **tx_msg_queue**

Transmit messages queue.

_wait_q_t **rx_msg_queue**

Receive message queue.

Pipes

A *pipe* is a kernel object that allows a thread to send a byte stream to another thread. Pipes can be used to synchronously transfer chunks of data in whole or in part.

- *Concepts*
- *Implementation*
  - *Writing to a Pipe*
  - *Reading from a Pipe*
  - *Flushing a Pipe's Buffer*

  - *Flushing a Pipe*
- *Suggested uses*
- *Configuration Options*
- *API Reference*

Concepts The pipe can be configured with a ring buffer which holds data that has been sent but not yet received; alternatively, the pipe may have no ring buffer.

Any number of pipes can be defined (limited only by available RAM). Each pipe is referenced by its memory address.

A pipe has the following key property:

- A **size** that indicates the size of the pipe's ring buffer. Note that a size of zero defines a pipe with no ring buffer.

A pipe must be initialized before it can be used. The pipe is initially empty.

Data is synchronously **sent** either in whole or in part to a pipe by a thread. If the specified minimum number of bytes can not be immediately satisfied, then the operation will either fail immediately or attempt to send as many bytes as possible and then pend in the hope that the send can be completed later. Accepted data is either copied to the pipe's ring buffer or directly to the waiting reader(s).

Data is synchronously **received** from a pipe by a thread. If the specified minimum number of bytes can not be immediately satisfied, then the operation will either fail immediately or attempt to receive as many bytes as possible and then pend in the hope that the receive can be completed later. Accepted data is either copied from the pipe's ring buffer or directly from the waiting sender(s).

Data may also be **flushed** from a pipe by a thread. Flushing can be performed either on the entire pipe or on only its ring buffer. Flushing the entire pipe is equivalent to reading all the information in the ring buffer **and** waiting to be written into a giant temporary buffer which is then discarded. Flushing the ring buffer is equivalent to reading **only** the data in the ring buffer into a temporary buffer which is then discarded. Flushing the ring buffer does not guarantee that the ring buffer will stay empty; flushing it may allow a pended writer to fill the ring buffer.

Note

Flushing does not in practice allocate or use additional buffers.

Note

The kernel does allow for an ISR to flush a pipe from an ISR. It also allows it to send/receive data to/from one provided it does not attempt to wait for space/data.

Implementation A pipe is defined using a variable of type `k_pipe` and an optional character buffer of type `unsigned char`. It must then be initialized by calling `k_pipe_init()`.

The following code defines and initializes an empty pipe that has a ring buffer capable of holding 100 bytes and is aligned to a 4-byte boundary.

```
unsigned char __aligned(4) my_ring_buffer[100];
struct k_pipe my_pipe;

k_pipe_init(&my_pipe, my_ring_buffer, sizeof(my_ring_buffer));
```

Alternatively, a pipe can be defined and initialized at compile time by calling *K_PIPE_DEFINE*.

The following code has the same effect as the code segment above. Observe that macro defines both the pipe and its ring buffer.

```
K_PIPE_DEFINE(my_pipe, 100, 4);
```

Writing to a Pipe Data is added to a pipe by calling *k_pipe_put()*.

The following code builds on the example above, and uses the pipe to pass data from a producing thread to one or more consuming threads. If the pipe's ring buffer fills up because the consumers can't keep up, the producing thread waits for a specified amount of time.

```
struct message_header {
    ...
};

void producer_thread(void)
{
    unsigned char *data;
    size_t total_size;
    size_t bytes_written;
    int    rc;
    ...

    while (1) {
        /* Craft message to send in the pipe */
        data = ...;
        total_size = ...;

        /* send data to the consumers */
        rc = k_pipe_put(&my_pipe, data, total_size, &bytes_written,
                        sizeof(struct message_header), K_NO_WAIT);

        if (rc < 0) {
            /* Incomplete message header sent */
            ...
        } else if (bytes_written < total_size) {
            /* Some of the data was sent */
            ...
        } else {
            /* All data sent */
            ...
        }
    }
}
```

Reading from a Pipe Data is read from the pipe by calling *k_pipe_get()*.

The following code builds on the example above, and uses the pipe to process data items generated by one or more producing threads.

```
void consumer_thread(void)
{
    unsigned char buffer[120];
```

(continues on next page)

(continued from previous page)

```
    size_t   bytes_read;
    struct message_header  *header = (struct message_header *)buffer;

    while (1) {
        rc = k_pipe_get(&my_pipe, buffer, sizeof(buffer), &bytes_read,
                        sizeof(*header), K_MSEC(100));

        if ((rc < 0) || (bytes_read < sizeof (*header))) {
            /* Incomplete message header received */
            ...
        } else if (header->num_data_bytes + sizeof(*header) > bytes_read) {
            /* Only some data was received */
            ...
        } else {
            /* All data was received */
            ...
        }
    }
}
```

Use a pipe to send streams of data between threads.

Note

A pipe can be used to transfer long streams of data if desired. However it is often preferable to send pointers to large data items to avoid copying the data.

Flushing a Pipe's Buffer Data is flushed from the pipe's ring buffer by calling *k_pipe_buffer_flush()*.

The following code builds on the examples above, and flushes the pipe's buffer.

```
void monitor_thread(void)
{
    while (1) {
        ...
        /* Pipe buffer contains stale data. Flush it. */
        k_pipe_buffer_flush(&my_pipe);
        ...
    }
}
```

Flushing a Pipe All data in the pipe is flushed by calling *k_pipe_flush()*.

The following code builds on the examples above, and flushes all the data in the pipe.

```
void monitor_thread(void)
{
    while (1) {
        ...
        /* Critical error detected. Flush the entire pipe to reset it. */
        k_pipe_flush(&my_pipe);
        ...
    }
}
```

Suggested uses Use a pipe to send streams of data between threads.

> **Note**
>
> A pipe can be used to transfer long streams of data if desired. However it is often preferable to send pointers to large data items to avoid copying the data. Copying large data items will negatively impact interrupt latency as a spinlock is held while copying that data.

Configuration Options Related configuration options:

- `CONFIG_PIPES`

API Reference

group `pipe_apis`

Defines

`K_PIPE_DEFINE`(name, pipe_buffer_size, pipe_align)

Statically define and initialize a pipe.

The pipe can be accessed outside the module where it is defined using:

```
extern struct k_pipe <name>;
```

Parameters

- **`name`** – Name of the pipe.
- **`pipe_buffer_size`** – Size of the pipe's ring buffer (in bytes), or zero if no ring buffer is used.
- **`pipe_align`** – Alignment of the pipe's ring buffer (power of 2).

Functions

void `k_pipe_init`(struct *k_pipe* *pipe, unsigned char *buffer, size_t size)

Initialize a pipe.

This routine initializes a pipe object, prior to its first use.

Parameters

- **`pipe`** – Address of the pipe.
- **`buffer`** – Address of the pipe's ring buffer, or NULL if no ring buffer is used.
- **`size`** – Size of the pipe's ring buffer (in bytes), or zero if no ring buffer is used.

int `k_pipe_cleanup`(struct *k_pipe* *pipe)

Release a pipe's allocated buffer.

If a pipe object was given a dynamically allocated buffer via *k_pipe_alloc_init()*, this will free it. This function does nothing if the buffer wasn't dynamically allocated.

Parameters

- **pipe** – Address of the pipe.

Return values

- **0** – on success
- **-EAGAIN** – nothing to cleanup

int **k_pipe_alloc_init**(struct *k_pipe* *pipe, size_t size)

Initialize a pipe and allocate a buffer for it.

Storage for the buffer region will be allocated from the calling thread's resource pool. This memory will be released if *k_pipe_cleanup()* is called, or userspace is enabled and the pipe object loses all references to it.

This function should only be called on uninitialized pipe objects.

Parameters

- **pipe** – Address of the pipe.
- **size** – Size of the pipe's ring buffer (in bytes), or zero if no ring buffer is used.

Return values

- **0** – on success
- **-ENOMEM** – if memory couldn't be allocated

int **k_pipe_put**(struct *k_pipe* *pipe, const void *data, size_t bytes_to_write, size_t *bytes_written, size_t min_xfer, *k_timeout_t* timeout)

Write data to a pipe.

This routine writes up to *bytes_to_write* bytes of data to *pipe*.

Parameters

- **pipe** – Address of the pipe.
- **data** – Address of data to write.
- **bytes_to_write** – Size of data (in bytes).
- **bytes_written** – Address of area to hold the number of bytes written.
- **min_xfer** – Minimum number of bytes to write.
- **timeout** – Waiting period to wait for the data to be written, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- **0** – At least *min_xfer* bytes of data were written.
- **-EIO** – Returned without waiting; zero data bytes were written.
- **-EAGAIN** – Waiting period timed out; between zero and *min_xfer* minus one data bytes were written.

int **k_pipe_get**(struct *k_pipe* *pipe, void *data, size_t bytes_to_read, size_t *bytes_read, size_t min_xfer, *k_timeout_t* timeout)

Read data from a pipe.

This routine reads up to *bytes_to_read* bytes of data from *pipe*.

Parameters

- **pipe** – Address of the pipe.
- **data** – Address to place the data read from pipe.
- **bytes_to_read** – Maximum number of data bytes to read.

- **bytes_read** – Address of area to hold the number of bytes read.
- **min_xfer** – Minimum number of data bytes to read.
- **timeout** – Waiting period to wait for the data to be read, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- **0** – At least *min_xfer* bytes of data were read.
- **-EINVAL** – invalid parameters supplied
- **-EIO** – Returned without waiting; zero data bytes were read.
- **-EAGAIN** – Waiting period timed out; between zero and *min_xfer* minus one data bytes were read.

size_t **k_pipe_read_avail**(struct *k_pipe* *pipe)

Query the number of bytes that may be read from *pipe*.

Parameters

- **pipe** – Address of the pipe.

Return values

a – number n such that 0 <= n <= *k_pipe::size*; the result is zero for unbuffered pipes.

size_t **k_pipe_write_avail**(struct *k_pipe* *pipe)

Query the number of bytes that may be written to *pipe*.

Parameters

- **pipe** – Address of the pipe.

Return values

a – number n such that 0 <= n <= *k_pipe::size*; the result is zero for unbuffered pipes.

void **k_pipe_flush**(struct *k_pipe* *pipe)

Flush the pipe of write data.

This routine flushes the pipe. Flushing the pipe is equivalent to reading both all the data in the pipe's buffer and all the data waiting to go into that pipe into a large temporary buffer and discarding the buffer. Any writers that were previously pended become unpended.

Parameters

- **pipe** – Address of the pipe.

void **k_pipe_buffer_flush**(struct *k_pipe* *pipe)

Flush the pipe's internal buffer.

This routine flushes the pipe's internal buffer. This is equivalent to reading up to N bytes from the pipe (where N is the size of the pipe's buffer) into a temporary buffer and then discarding that buffer. If there were writers previously pending, then some may unpend as they try to fill up the pipe's emptied buffer.

Parameters

- **pipe** – Address of the pipe.

struct **k_pipe**

#include <kernel.h> Pipe Structure.

Public Members

unsigned char *`buffer`
: Pipe buffer: may be NULL.

size_t `size`
: Buffer size.

size_t `bytes_used`
: Number of bytes used in buffer.

size_t `read_index`
: Where in buffer to read from.

size_t `write_index`
: Where in buffer to write.

struct *k_spinlock* `lock`
: Synchronization lock.

_wait_q_t `readers`
: Reader wait queue.

_wait_q_t `writers`
: Writer wait queue.

uint8_t `flags`
: Wait queue.

  Flags

3.1.3 Memory Management

See *Memory Management*.

3.1.4 Timing

These pages cover timing related services.

Kernel Timing

Zephyr provides a robust and scalable timing framework to enable reporting and tracking of timed events from hardware timing sources of arbitrary precision.

Time Units Kernel time is tracked in several units which are used for different purposes.

Real time values, typically specified in milliseconds or microseconds, are the default presentation of time to application code. They have the advantages of being universally portable and pervasively understood, though they may not match the precision of the underlying hardware perfectly.

The kernel presents a "cycle" count via the *k_cycle_get_32()* and *k_cycle_get_64()* APIs. The intent is that this counter represents the fastest cycle counter that the operating system is able to present to the user (for example, a CPU cycle counter) and that the read operation is very fast. The expectation is that very sensitive application code might use this in a polling manner to achieve maximal precision. The frequency of this counter is required to be steady over time, and is available from `sys_clock_hw_cycles_per_sec()` (which on almost all platforms is a runtime constant that evaluates to CONFIG_SYS_CLOCK_HW_CYCLES_PER_SEC).

For asynchronous timekeeping, the kernel defines a "ticks" concept. A "tick" is the internal count in which the kernel does all its internal uptime and timeout bookkeeping. Interrupts are expected to be delivered on tick boundaries to the extent practical, and no fractional ticks are tracked. The choice of tick rate is configurable via `CONFIG_SYS_CLOCK_TICKS_PER_SEC`. Defaults on most hardware platforms (ones that support setting arbitrary interrupt timeouts) are expected to be in the range of 10 kHz, with software emulation platforms and legacy drivers using a more traditional 100 Hz value.

Conversion Zephyr provides an extensively enumerated conversion library with rounding control for all time units. Any unit of "ms" (milliseconds), "us" (microseconds), "tick", or "cyc" can be converted to any other. Control of rounding is provided, and each conversion is available in "floor" (round down to nearest output unit), "ceil" (round up) and "near" (round to nearest). Finally the output precision can be specified as either 32 or 64 bits.

For example: `k_ms_to_ticks_ceil32()` will convert a millisecond input value to the next higher number of ticks, returning a result truncated to 32 bits of precision; and `k_cyc_to_us_floor64()` will convert a measured cycle count to an elapsed number of microseconds in a full 64 bits of precision. See the reference documentation for the full enumeration of conversion routines.

On most platforms, where the various counter rates are integral multiples of each other and where the output fits within a single word, these conversions expand to a 2-4 operation sequence, requiring full precision only where actually required and requested.

Uptime The kernel tracks a system uptime count on behalf of the application. This is available at all times via *k_uptime_get()*, which provides an uptime value in milliseconds since system boot. This is expected to be the utility used by most portable application code.

The internal tracking, however, is as a 64 bit integer count of ticks. Apps with precise timing requirements (that are willing to do their own conversions to portable real time units) may access this with *k_uptime_ticks()*.

Timeouts The Zephyr kernel provides many APIs with a "timeout" parameter. Conceptually, this indicates the time at which an event will occur. For example:

- Kernel blocking operations like *k_sem_take()* or *k_queue_get()* may provide a timeout after which the routine will return with an error code if no data is available.
- Kernel `k_timer` objects must specify delays for their duration and period.
- The kernel *k_work_delayable* API provides a timeout parameter indicating when a work queue item will be added to the system queue.

All these values are specified using a *k_timeout_t* value. This is an opaque struct type that must be initialized using one of a family of kernel timeout macros. The most common, *K_MSEC*, defines a time in milliseconds after the current time.

What is meant by "current time" for relative timeouts depends on the context:

- When scheduling a relative timeout from within a timeout callback (e.g. from within the expiry function passed to *k_timer_init()* or the work handler passed to *k_work_init_delayable()*), "current time" is the exact time at which the currently firing timeout was originally scheduled even if the "real time" will already have advanced. This is to ensure that timers scheduled from within another timer's callback will always be calculated with a precise offset to the firing timer. It is thereby possible to fire at regular intervals without introducing systematic clock drift over time.
- When scheduling a timeout from application context, "current time" means the value returned by *k_uptime_ticks()* at the time at which the kernel receives the timeout value.

Other options for timeout initialization follow the unit conventions described above: *K_NSEC()*, *K_USEC*, *K_TICKS* and *K_CYC()* specify timeout values that will expire after specified numbers of nanoseconds, microseconds, ticks and cycles, respectively.

Precision of *k_timeout_t* values is configurable, with the default being 32 bits. Large uptime counts in non-tick units will experience complicated rollover semantics, so it is expected that timing-sensitive applications with long uptimes will be configured to use a 64 bit timeout type.

Finally, it is possible to specify timeouts as absolute times since system boot. A timeout initialized with `K_TIMEOUT_ABS_MS` indicates a timeout that will expire after the system uptime reaches the specified value. There are likewise nanosecond, microsecond, cycles and ticks variants of this API.

Timing Internals

Timeout Queue All Zephyr *k_timeout_t* events specified using the API above are managed in a single, global queue of events. Each event is stored in a double-linked list, with an attendant delta count in ticks from the previous event. The action to take on an event is specified as a callback function pointer provided by the subsystem requesting the event, along with a `_timeout` tracking struct that is expected to be embedded within subsystem-defined data structures (for example: a `wait_q` struct, or a `k_tid_t` thread struct).

Note that all variant units passed via a *k_timeout_t* are converted to ticks once on insertion into the list. There no multiple-conversion steps internal to the kernel, so precision is guaranteed at the tick level no matter how many events exist or how long a timeout might be.

Note that the list structure means that the CPU work involved in managing large numbers of timeouts is quadratic in the number of active timeouts. The API design of the timeout queue was intended to permit a more scalable backend data structure, but no such implementation exists currently.

Timer Drivers Kernel timing at the tick level is driven by a timer driver with a comparatively simple API.

- The driver is expected to be able to "announce" new ticks to the kernel via the *sys_clock_announce()* call, which passes an integer number of ticks that have elapsed since the last announce call (or system boot). These calls can occur at any time, but the driver is expected to attempt to ensure (to the extent practical given interrupt latency interactions) that they occur near tick boundaries (i.e. not "halfway through" a tick), and most importantly that they be correct over time and subject to minimal skew vs. other counters and real world time.
- The driver is expected to provide a *sys_clock_set_timeout()* call to the kernel which indicates how many ticks may elapse before the kernel must receive an announce call to trigger registered timeouts. It is legal to announce new ticks before that moment (though they must be correct) but delay after that will cause events to be missed. Note that the timeout value passed here is in a delta from current time, but that does not absolve the driver of

the requirement to provide ticks at a steady rate over time. Naive implementations of this function are subject to bugs where the fractional tick gets "reset" incorrectly and causes clock skew.

- The driver is expected to provide a *sys_clock_elapsed()* call which provides a current indication of how many ticks have elapsed (as compared to a real world clock) since the last call to *sys_clock_announce()*, which the kernel needs to test newly arriving timeouts for expiration.

Note that a natural implementation of this API results in a "tickless" kernel, which receives and processes timer interrupts only for registered events, relying on programmable hardware counters to provide irregular interrupts. But a traditional, "ticked" or "dumb" counter driver can be trivially implemented also:

- The driver can receive interrupts at a regular rate corresponding to the OS tick rate, calling *sys_clock_announce()* with an argument of one each time.
- The driver can ignore calls to *sys_clock_set_timeout()*, as every tick will be announced regardless of timeout status.
- The driver can return zero for every call to *sys_clock_elapsed()* as no more than one tick can be detected as having elapsed (because otherwise an interrupt would have been received).

SMP Details In general, the timer API described above does not change when run in a multiprocessor context. The kernel will internally synchronize all access appropriately, and ensure that all critical sections are small and minimal. But some notes are important to detail:

- Zephyr is agnostic about which CPU services timer interrupts. It is not illegal (though probably undesirable in some circumstances) to have every timer interrupt handled on a single processor. Existing SMP architectures implement symmetric timer drivers.
- The *sys_clock_announce()* call is expected to be globally synchronized at the driver level. The kernel does not do any per-CPU tracking, and expects that if two timer interrupts fire near simultaneously, that only one will provide the current tick count to the timing subsystem. The other may legally provide a tick count of zero if no ticks have elapsed. It should not "skip" the announce call because of timeslicing requirements (see below).
- Some SMP hardware uses a single, global timer device, others use a per-CPU counter. The complexity here (for example: ensuring counter synchronization between CPUs) is expected to be managed by the driver, not the kernel.
- The next timeout value passed back to the driver via *sys_clock_set_timeout()* is done identically for every CPU. So by default, every CPU will see simultaneous timer interrupts for every event, even though by definition only one of them should see a non-zero ticks argument to *sys_clock_announce()*. This is probably a correct default for timing sensitive applications (because it minimizes the chance that an errant ISR or interrupt lock will delay a timeout), but may be a performance problem in some cases. The current design expects that any such optimization is the responsibility of the timer driver.

Time Slicing An auxiliary job of the timing subsystem is to provide tick counters to the scheduler that allow implementation of time slicing of threads. A thread time-slice cannot be a timeout value, as it does not reflect a global expiration but instead a per-CPU value that needs to be tracked independently on each CPU in an SMP context.

Because there may be no other hardware available to drive timeslicing, Zephyr multiplexes the existing timer driver. This means that the value passed to *sys_clock_set_timeout()* may be clamped to a smaller value than the current next timeout when a time sliced thread is currently scheduled.

Subsystems that keep millisecond APIs In general, code like this will port just like applications code will. Millisecond values from the user may be treated any way the subsystem likes, and then converted into kernel timeouts using *K_MSEC()* at the point where they are presented to the kernel.

Obviously this comes at the cost of not being able to use new features, like the higher precision timeout constructors or absolute timeouts. But for many subsystems with simple needs, this may be acceptable.

One complexity is *K_FOREVER*. Subsystems that might have been able to accept this value to their millisecond API in the past no longer can, because it is no longer an integral type. Such code will need to use a different, integer-valued token to represent "forever". *K_NO_WAIT* has the same typesafety concern too, of course, but as it is (and has always been) simply a numerical zero, it has a natural porting path.

Subsystems using k_timeout_t Ideally, code that takes a "timeout" parameter specifying a time to wait should be using the kernel native abstraction where possible. But *k_timeout_t* is opaque, and needs to be converted before it can be inspected by an application.

Some conversions are simple. Code that needs to test for *K_FOREVER* can simply use the *K_TIMEOUT_EQ()* macro to test the opaque struct for equality and take special action.

The more complicated case is when the subsystem needs to take a timeout and loop, waiting for it to finish while doing some processing that may require multiple blocking operations on underlying kernel code. For example, consider this design:

```
void my_wait_for_event(struct my_subsys *obj, int32_t timeout_in_ms)
{
    while (true) {
        uint32_t start = k_uptime_get_32();

        if (is_event_complete(obj)) {
            return;
        }

        /* Wait for notification of state change */
        k_sem_take(obj->sem, timeout_in_ms);

        /* Subtract elapsed time */
        timeout_in_ms -= (k_uptime_get_32() - start);
    }
}
```

This code requires that the timeout value be inspected, which is no longer possible. For situations like this, the new API provides the internal *sys_timepoint_calc()* and *sys_timepoint_timeout()* routines that converts an arbitrary timeout to and from a timepoint value based on an uptime tick at which it will expire. So such a loop might look like:

```
void my_wait_for_event(struct my_subsys *obj, k_timeout_t timeout)
{
    /* Compute the end time from the timeout */
    k_timepoint_t end = sys_timepoint_calc(timeout);

    do {
        if (is_event_complete(obj)) {
            return;
        }

        /* Update timeout with remaining time */
        timeout = sys_timepoint_timeout(end);
```

(continues on next page)

(continued from previous page)

```
        /* Wait for notification of state change */
        k_sem_take(obj->sem, timeout);
    } while (!K_TIMEOUT_EQ(timeout, K_NO_WAIT));
}
```

Note that *sys_timepoint_calc()* accepts special values *K_FOREVER* and *K_NO_WAIT*, and works identically for absolute timeouts as well as conventional ones. Conversely, *sys_timepoint_timeout()* may return *K_FOREVER* or *K_NO_WAIT* if those were used to create the timepoint, the later also being returned if the timepoint is now in the past. For simple cases, *sys_timepoint_expired()* can be used as well.

But some care is still required for subsystems that use those. Note that delta timeouts need to be interpreted relative to a "current time", and obviously that time is the time of the call to *sys_timepoint_calc()*. But the user expects that the time is the time they passed the timeout to you. Care must be taken to call this function just once, as synchronously as possible to the timeout creation in user code. It should not be used on a "stored" timeout value, and should never be called iteratively in a loop.

API Reference

group `clock_apis`

System Clock APIs.

Defines

`K_NO_WAIT`

Generate null timeout delay.

This macro generates a timeout delay that instructs a kernel API not to wait if the requested operation cannot be performed immediately.

Returns
Timeout delay value.

`K_NSEC`(t)

Generate timeout delay from nanoseconds.

This macro generates a timeout delay that instructs a kernel API to wait up to *t* nanoseconds to perform the requested operation. Note that timer precision is limited to the tick rate, not the requested value.

Parameters

- **t** – Duration in nanoseconds.

Returns
Timeout delay value.

`K_USEC`(t)

Generate timeout delay from microseconds.

This macro generates a timeout delay that instructs a kernel API to wait up to *t* microseconds to perform the requested operation. Note that timer precision is limited to the tick rate, not the requested value.

Parameters

- **t** – Duration in microseconds.

Returns
Timeout delay value.

K_CYC(t)

Generate timeout delay from cycles.

This macro generates a timeout delay that instructs a kernel API to wait up to *t* cycles to perform the requested operation.

Parameters

- **t** – Duration in cycles.

Returns
Timeout delay value.

K_TICKS(t)

Generate timeout delay from system ticks.

This macro generates a timeout delay that instructs a kernel API to wait up to *t* ticks to perform the requested operation.

Parameters

- **t** – Duration in system ticks.

Returns
Timeout delay value.

K_MSEC(ms)

Generate timeout delay from milliseconds.

This macro generates a timeout delay that instructs a kernel API to wait up to *ms* milliseconds to perform the requested operation.

Parameters

- **ms** – Duration in milliseconds.

Returns
Timeout delay value.

K_SECONDS(s)

Generate timeout delay from seconds.

This macro generates a timeout delay that instructs a kernel API to wait up to *s* seconds to perform the requested operation.

Parameters

- **s** – Duration in seconds.

Returns
Timeout delay value.

K_MINUTES(m)

Generate timeout delay from minutes.

This macro generates a timeout delay that instructs a kernel API to wait up to *m* minutes to perform the requested operation.

Parameters

- **m** – Duration in minutes.

Returns
Timeout delay value.

K_HOURS(h)

Generate timeout delay from hours.

This macro generates a timeout delay that instructs a kernel API to wait up to *h* hours to perform the requested operation.

Parameters

- h – Duration in hours.

Returns
Timeout delay value.

K_FOREVER

Generate infinite timeout delay.

This macro generates a timeout delay that instructs a kernel API to wait as long as necessary to perform the requested operation.

Returns
Timeout delay value.

K_TICKS_FOREVER

K_TIMEOUT_EQ(a, b)

Compare timeouts for equality.

The *k_timeout_t* object is an opaque struct that should not be inspected by application code. This macro exists so that users can test timeout objects for equality with known constants (e.g. K_NO_WAIT and K_FOREVER) when implementing their own APIs in terms of Zephyr timeout constants.

Returns
True if the timeout objects are identical

NSEC_PER_USEC

number of nanoseconds per micorsecond

NSEC_PER_MSEC

number of nanoseconds per millisecond

USEC_PER_MSEC

number of microseconds per millisecond

MSEC_PER_SEC

number of milliseconds per second

SEC_PER_MIN

number of seconds per minute

MIN_PER_HOUR

number of minutes per hour

HOUR_PER_DAY

number of hours per day

USEC_PER_SEC

number of microseconds per second

NSEC_PER_SEC

number of nanoseconds per second

SYS_CLOCK_HW_CYCLES_TO_NS_AVG(X, NCYCLES)

SYS_CLOCK_HW_CYCLES_TO_NS_AVG converts CPU clock cycles to nanoseconds and calculates the average cycle time.

Typedefs

typedef uint32_t **k_ticks_t**

Tick precision used in timeout APIs.

This type defines the word size of the timeout values used in *k_timeout_t* objects, and thus defines an upper bound on maximum timeout length (or equivalently minimum tick duration). Note that this does not affect the size of the system uptime counter, which is always a 64 bit count of ticks.

Functions

void **sys_clock_set_timeout**(int32_t ticks, bool idle)

Set system clock timeout.

Informs the system clock driver that the next needed call to *sys_clock_announce()* will not be until the specified number of ticks from the current time have elapsed. Note that spurious calls to *sys_clock_announce()* are allowed (i.e. it's legal to announce every tick and implement this function as a noop), the requirement is that one tick announcement should occur within one tick BEFORE the specified expiration (that is, passing ticks==1 means "announce

the next tick", this convention was chosen to match legacy usage). Similarly a ticks value of zero (or even negative) is legal and treated identically: it simply indicates the kernel would like the next tick announcement as soon as possible.

Note that ticks can also be passed the special value K_TICKS_FOREVER, indicating that no future timer interrupts are expected or required and that the system is permitted to enter an indefinite sleep even if this could cause rollover of the internal counter (i.e. the system uptime counter is allowed to be wrong

Note also that it is conventional for the kernel to pass INT_MAX for ticks if it wants to preserve the uptime tick count but doesn't have a specific event to await. The intent here is that the driver will schedule any needed timeout as far into the future as possible. For the specific case of INT_MAX, the next call to *sys_clock_announce()* may occur at any point in the future, not just at INT_MAX ticks. But the correspondence between the announced ticks and real-world time must be correct.

A final note about SMP: note that the call to *sys_clock_set_timeout()* is made on any CPU, and reflects the next timeout desired globally. The resulting calls(s) to *sys_clock_announce()* must be properly serialized by the driver such that a given tick is announced exactly once across the system. The kernel does not (cannot, really) attempt to serialize things by "assigning" timeouts to specific CPUs.

Parameters

- **ticks** – Timeout in tick units

- **idle** – Hint to the driver that the system is about to enter the idle state immediately after setting the timeout

void **sys_clock_idle_exit**(void)

Timer idle exit notification.

This notifies the timer driver that the system is exiting the idle and allows it to do whatever bookkeeping is needed to restore timer operation and compute elapsed ticks.

> **Note**
>
> Legacy timer drivers also use this opportunity to call back into *sys_clock_announce()* to notify the kernel of expired ticks. This is allowed for compatibility, but not recommended. The kernel will figure that out on its own.

void **sys_clock_announce**(int32_t ticks)

Announce time progress to the kernel.

Informs the kernel that the specified number of ticks have elapsed since the last call to *sys_clock_announce()* (or system startup for the first call). The timer driver is expected to delivery these announcements as close as practical (subject to hardware and latency limitations) to tick boundaries.

Parameters

- **ticks** – Elapsed time, in ticks

uint32_t **sys_clock_elapsed**(void)

Ticks elapsed since last *sys_clock_announce()* call.

Queries the clock driver for the current time elapsed since the last call to *sys_clock_announce()* was made. The kernel will call this with appropriate locking, the driver needs only provide an instantaneous answer.

void **sys_clock_disable**(void)

Disable system timer.

> **Note**
>
> Not all system timer drivers has the capability of being disabled. The config CONFIG_SYSTEM_TIMER_HAS_DISABLE_SUPPORT can be used to check if the system timer has the capability of being disabled.

uint32_t **sys_clock_cycle_get_32**(void)

Hardware cycle counter.

Timer drivers are generally responsible for the system cycle counter as well as the tick announcements. This function is generally called out of the architecture layer (

> **See also**
>
> arch_k_cycle_get_32()) to implement the cycle counter, though the user-facing API is owned by the architecture, not the driver. The rate must match CONFIG_SYS_CLOCK_HW_CYCLES_PER_SEC.

Note

If the counter clock is large enough for this to wrap its full range within a few seconds (i.e. CONFIG_SYS_CLOCK_HW_CYCLES_PER_SEC is greater than 50Mhz) then it is recommended to also implement *sys_clock_cycle_get_64()*.

Returns
The current cycle time. This should count up monotonically through the full 32 bit space, wrapping at 0xffffffff. Hardware with fewer bits of precision in the timer is expected to synthesize a 32 bit count.

uint64_t **sys_clock_cycle_get_64**(void)

64 bit hardware cycle counter

As for *sys_clock_cycle_get_32()*, but with a 64 bit return value. Not all hardware has 64 bit counters. This function need be implemented only if CONFIG_TIMER_HAS_64BIT_CYCLE_COUNTER is set.

Note

If the counter clock is large enough for *sys_clock_cycle_get_32()* to wrap its full range within a few seconds (i.e. CONFIG_SYS_CLOCK_HW_CYCLES_PER_SEC is greater than 50Mhz) then it is recommended to implement this API.

Returns
The current cycle time. This should count up monotonically through the full 64 bit space, wrapping at 2^64-1. Hardware with fewer bits of precision in the timer is generally not expected to implement this API.

int64_t **k_uptime_ticks**(void)

Get system uptime, in system ticks.

This routine returns the elapsed time since the system booted, in ticks (c.f. `CONFIG_SYS_CLOCK_TICKS_PER_SEC`), which is the fundamental unit of resolution of kernel timekeeping.

Returns
Current uptime in ticks.

static inline int64_t **k_uptime_get**(void)

Get system uptime.

This routine returns the elapsed time since the system booted, in milliseconds.

Note

While this function returns time in milliseconds, it does not mean it has millisecond resolution. The actual resolution depends on `CONFIG_SYS_CLOCK_TICKS_PER_SEC` config option.

Returns
Current uptime in milliseconds.

static inline uint32_t **k_uptime_get_32**(void)

Get system uptime (32-bit version).

This routine returns the lower 32 bits of the system uptime in milliseconds.

Because correct conversion requires full precision of the system clock there is no benefit to using this over *k_uptime_get()* unless you know the application will never run long enough for the system clock to approach 2^32 ticks. Calls to this function may involve interrupt blocking and 64-bit math.

Note

While this function returns time in milliseconds, it does not mean it has millisecond resolution. The actual resolution depends on `CONFIG_SYS_CLOCK_TICKS_PER_SEC` config option

Returns
The low 32 bits of the current uptime, in milliseconds.

static inline uint32_t **k_uptime_seconds**(void)

Get system uptime in seconds.

This routine returns the elapsed time since the system booted, in seconds.

Returns
Current uptime in seconds.

static inline int64_t **k_uptime_delta**(int64_t *reftime)

Get elapsed time.

This routine computes the elapsed time between the current system uptime and an earlier reference time, in milliseconds.

Parameters

- **reftime** – Pointer to a reference time, which is updated to the current uptime upon return.

Returns
Elapsed time.

static inline uint32_t **k_cycle_get_32**(void)

Read the hardware clock.

This routine returns the current time, as measured by the system's hardware clock.

Returns
Current hardware clock up-counter (in cycles).

static inline uint64_t **k_cycle_get_64**(void)

Read the 64-bit hardware clock.

This routine returns the current time in 64-bits, as measured by the system's hardware clock, if available.

See also

CONFIG_TIMER_HAS_64BIT_CYCLE_COUNTER

Returns
Current hardware clock up-counter (in cycles).

uint32_t **sys_clock_tick_get_32**(void)

Return the lower part of the current system tick count.

Returns
the current system tick count

int64_t **sys_clock_tick_get**(void)

Return the current system tick count.

Returns
the current system tick count

k_timepoint_t **sys_timepoint_calc**(*k_timeout_t* timeout)

Calculate a timepoint value.

Returns a timepoint corresponding to the expiration (relative to an unlocked "now"!) of a timeout object. When used correctly, this should be called once, synchronously with the user passing a new timeout value. It should not be used iteratively to adjust a timeout (see *sys_timepoint_timeout()* for that purpose).

See also

sys_timepoint_timeout()

See also

sys_timepoint_expired()

Parameters

- **timeout** – Timeout value relative to current time (may also be K_FOREVER or K_NO_WAIT).

Return values
Timepoint – value corresponding to given timeout

k_timeout_t **sys_timepoint_timeout**(*k_timepoint_t* timepoint)

Remaining time to given timepoint.

Returns the timeout interval between current time and provided timepoint. If the timepoint is now in the past or if it was created with K_NO_WAIT then K_NO_WAIT is returned. If it was created with K_FOREVER then K_FOREVER is returned.

See also

sys_timepoint_calc()

Parameters

- **timepoint** – Timepoint for which a timeout value is wanted.

Return values
`Corresponding` – timeout value.

static inline uint64_t **sys_clock_timeout_end_calc**(*k_timeout_t* timeout)

Provided for backward compatibility.

This is deprecated. Consider *sys_timepoint_calc()* instead.

See also

sys_timepoint_calc()

static inline int **sys_timepoint_cmp**(*k_timepoint_t* a, *k_timepoint_t* b)

Compare two timepoint values.

This function is used to compare two timepoint values.

Parameters

- `a` – Timepoint to compare
- `b` – Timepoint to compare against.

Returns
zero if both timepoints are the same. Negative value if timepoint *a* is before timepoint *b*, positive otherwise.

static inline bool **sys_timepoint_expired**(*k_timepoint_t* timepoint)

Indicates if timepoint is expired.

See also

sys_timepoint_calc()

Parameters

- `timepoint` – Timepoint to evaluate

Return values
`true` – if the timepoint is in the past, false otherwise

struct **k_timeout_t**

#include <*sys_clock.h*> Kernel timeout type.

Timeout arguments presented to kernel APIs are stored in this opaque type, which is capable of representing times in various formats and units. It should be constructed from application data using one of the macros defined for this purpose (e.g. *K_MSEC()*, K_TIMEOUT_ABS_TICKS(), etc...), or be one of the two constants K_NO_WAIT or K_FOREVER. Applications should not inspect the internal data once constructed. Timeout values may be compared for equality with the *K_TIMEOUT_EQ()* macro.

struct **k_timepoint_t**

#include <*sys_clock.h*> Kernel timepoint type.

Absolute timepoints are stored in this opaque type. It is best not to inspect its content directly.

See also

sys_timepoint_calc()

See also

sys_timepoint_timeout()

See also

sys_timepoint_expired()

Timers

A *timer* is a kernel object that measures the passage of time using the kernel's system clock. When a timer's specified time limit is reached it can perform an application-defined action, or it can simply record the expiration and wait for the application to read its status.

- *Concepts*
- *Implementation*
  - *Defining a Timer*
  - *Using a Timer Expiry Function*
  - *Reading Timer Status*
  - *Using Timer Status Synchronization*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

Concepts Any number of timers can be defined (limited only by available RAM). Each timer is referenced by its memory address.

A timer has the following key properties:

- A **duration** specifying the time interval before the timer expires for the first time. This is a `k_timeout_t` value that may be initialized via different units.
- A **period** specifying the time interval between all timer expirations after the first one, also a `k_timeout_t`. It must be non-negative. A period of `K_NO_WAIT` (i.e. zero) or `K_FOREVER` means that the timer is a one-shot timer that stops after a single expiration. (For example then, if a timer is started with a duration of 200 and a period of 75, it will first expire after 200 ms and then every 75 ms after that.)
- An **expiry function** that is executed each time the timer expires. The function is executed by the system clock interrupt handler. If no expiry function is required a `NULL` function can be specified.

- A **stop function** that is executed if the timer is stopped prematurely while running. The function is executed by the thread that stops the timer. If no stop function is required a `NULL` function can be specified.
- A **status** value that indicates how many times the timer has expired since the status value was last read.

A timer must be initialized before it can be used. This specifies its expiry function and stop function values, sets the timer's status to zero, and puts the timer into the **stopped** state.

A timer is **started** by specifying a duration and a period. The timer's status is reset to zero, and then the timer enters the **running** state and begins counting down towards expiry.

Note that the timer's duration and period parameters specify **minimum** delays that will elapse. Because of internal system timer precision (and potentially runtime interactions like interrupt delay) it is possible that more time may have passed as measured by reads from the relevant system time APIs. But at least this much time is guaranteed to have elapsed.

When a running timer expires its status is incremented and the timer executes its expiry function, if one exists; If a thread is waiting on the timer, it is unblocked. If the timer's period is zero the timer enters the stopped state; otherwise, the timer restarts with a new duration equal to its period.

A running timer can be stopped in mid-countdown, if desired. The timer's status is left unchanged, then the timer enters the stopped state and executes its stop function, if one exists. If a thread is waiting on the timer, it is unblocked. Attempting to stop a non-running timer is permitted, but has no effect on the timer since it is already stopped.

A running timer can be restarted in mid-countdown, if desired. The timer's status is reset to zero, then the timer begins counting down using the new duration and period values specified by the caller. If a thread is waiting on the timer, it continues waiting.

A timer's status can be read directly at any time to determine how many times the timer has expired since its status was last read. Reading a timer's status resets its value to zero. The amount of time remaining before the timer expires can also be read; a value of zero indicates that the timer is stopped.

A thread may read a timer's status indirectly by **synchronizing** with the timer. This blocks the thread until the timer's status is non-zero (indicating that it has expired at least once) or the timer is stopped; if the timer status is already non-zero or the timer is already stopped the thread continues without waiting. The synchronization operation returns the timer's status and resets it to zero.

> **Note**
>
> Only a single user should examine the status of any given timer, since reading the status (directly or indirectly) changes its value. Similarly, only a single thread at a time should synchronize with a given timer. ISRs are not permitted to synchronize with timers, since ISRs are not allowed to block.

Implementation

Defining a Timer A timer is defined using a variable of type `k_timer`. It must then be initialized by calling *`k_timer_init()`*.

The following code defines and initializes a timer.

```
struct k_timer my_timer;
extern void my_expiry_function(struct k_timer *timer_id);
```

(continues on next page)

(continued from previous page)

```
k_timer_init(&my_timer, my_expiry_function, NULL);
```

Alternatively, a timer can be defined and initialized at compile time by calling *K_TIMER_DEFINE*.

The following code has the same effect as the code segment above.

```
K_TIMER_DEFINE(my_timer, my_expiry_function, NULL);
```

Using a Timer Expiry Function The following code uses a timer to perform a non-trivial action on a periodic basis. Since the required work cannot be done at the interrupt level, the timer's expiry function submits a work item to the *system workqueue*, whose thread performs the work.

```
void my_work_handler(struct k_work *work)
{
    /* do the processing that needs to be done periodically */
    ...
}

K_WORK_DEFINE(my_work, my_work_handler);

void my_timer_handler(struct k_timer *dummy)
{
    k_work_submit(&my_work);
}

K_TIMER_DEFINE(my_timer, my_timer_handler, NULL);

...

/* start a periodic timer that expires once every second */
k_timer_start(&my_timer, K_SECONDS(1), K_SECONDS(1));
```

Reading Timer Status The following code reads a timer's status directly to determine if the timer has expired or not.

```
K_TIMER_DEFINE(my_status_timer, NULL, NULL);

...

/* start a one-shot timer that expires after 200 ms */
k_timer_start(&my_status_timer, K_MSEC(200), K_NO_WAIT);

/* do work */
...

/* check timer status */
if (k_timer_status_get(&my_status_timer) > 0) {
    /* timer has expired */
} else if (k_timer_remaining_get(&my_status_timer) == 0) {
    /* timer was stopped (by someone else) before expiring */
} else {
    /* timer is still running */
}
```

Using Timer Status Synchronization The following code performs timer status synchronization to allow a thread to do useful work while ensuring that a pair of protocol operations are separated by the specified time interval.

```
K_TIMER_DEFINE(my_sync_timer, NULL, NULL);

...

/* do first protocol operation */
...

/* start a one-shot timer that expires after 500 ms */
k_timer_start(&my_sync_timer, K_MSEC(500), K_NO_WAIT);

/* do other work */
...

/* ensure timer has expired (waiting for expiry, if necessary) */
k_timer_status_sync(&my_sync_timer);

/* do second protocol operation */
...
```

Note

If the thread had no other work to do it could simply sleep between the two protocol operations, without using a timer.

Suggested Uses Use a timer to initiate an asynchronous operation after a specified amount of time.

Use a timer to determine whether or not a specified amount of time has elapsed. In particular, timers should be used when higher precision and/or unit control is required than that afforded by the simpler *k_sleep()* and *k_usleep()* calls.

Use a timer to perform other work while carrying out operations involving time limits.

Note

If a thread needs to measure the time required to perform an operation it can read the *system clock or the hardware clock* directly, rather than using a timer.

Configuration Options Related configuration options:

- None

Related code samples

KSCAN
Use the KSCAN API to read key presses and releases on a keyboard matrix.

API Reference

group `timer_apis`

Defines

K_TIMER_DEFINE(name, expiry_fn, stop_fn)

Statically define and initialize a timer.

The timer can be accessed outside the module where it is defined using:

```
extern struct k_timer <name>;
```

Parameters

- **name** – Name of the timer variable.
- **expiry_fn** – Function to invoke each time the timer expires.
- **stop_fn** – Function to invoke if the timer is stopped while running.

Typedefs

typedef void (*k_timer_expiry_t)(struct k_timer *timer)

Timer expiry function type.

A timer's expiry function is executed by the system clock interrupt handler each time the timer expires. The expiry function is optional, and is only invoked if the timer has been initialized with one.

Param timer
Address of timer.

typedef void (*k_timer_stop_t)(struct k_timer *timer)

Timer stop function type.

A timer's stop function is executed if the timer is stopped prematurely. The function runs in the context of call that stops the timer. As *k_timer_stop()* can be invoked from an ISR, the stop function must be callable from interrupt context (isr-ok).

The stop function is optional, and is only invoked if the timer has been initialized with one.

Param timer
Address of timer.

Functions

void k_timer_init(struct k_timer *timer, *k_timer_expiry_t* expiry_fn, *k_timer_stop_t* stop_fn)

Initialize a timer.

This routine initializes a timer, prior to its first use.

Parameters

- **timer** – Address of timer.
- **expiry_fn** – Function to invoke each time the timer expires.
- **stop_fn** – Function to invoke if the timer is stopped while running.

void **k_timer_start**(struct k_timer *timer, *k_timeout_t* duration, *k_timeout_t* period)

Start a timer.

This routine starts a timer, and resets its status to zero. The timer begins counting down using the specified duration and period values.

Attempting to start a timer that is already running is permitted. The timer's status is reset to zero and the timer begins counting down using the new duration and period values.

Parameters

- **timer** – Address of timer.
- **duration** – Initial timer duration.
- **period** – Timer period.

void **k_timer_stop**(struct k_timer *timer)

Stop a timer.

This routine stops a running timer prematurely. The timer's stop function, if one exists, is invoked by the caller.

Attempting to stop a timer that is not running is permitted, but has no effect on the timer.

Function properties (list may not be complete)

isr-ok

Note

The stop handler has to be callable from ISRs if *k_timer_stop* is to be called from ISRs.

Parameters

- **timer** – Address of timer.

uint32_t **k_timer_status_get**(struct k_timer *timer)

Read timer status.

This routine reads the timer's status, which indicates the number of times it has expired since its status was last read.

Calling this routine resets the timer's status to zero.

Parameters

- **timer** – Address of timer.

Returns

Timer status.

uint32_t **k_timer_status_sync**(struct k_timer *timer)

Synchronize thread to timer expiration.

This routine blocks the calling thread until the timer's status is non-zero (indicating that it has expired at least once since it was last examined) or the timer is stopped. If the timer status is already non-zero, or the timer is already stopped, the caller continues without waiting.

Calling this routine resets the timer's status to zero.

This routine must not be used by interrupt handlers, since they are not allowed to block.

Parameters

- **timer** – Address of timer.

Returns
Timer status.

k_ticks_t **k_timer_expires_ticks**(const struct k_timer *timer)

Get next expiration time of a timer, in system ticks.

This routine returns the future system uptime reached at the next time of expiration of the timer, in units of system ticks. If the timer is not running, current system time is returned.

Parameters

- **timer** – The timer object

Returns
Uptime of expiration, in ticks

k_ticks_t **k_timer_remaining_ticks**(const struct k_timer *timer)

Get time remaining before a timer next expires, in system ticks.

This routine computes the time remaining before a running timer next expires, in units of system ticks. If the timer is not running, it returns zero.

Parameters

- **timer** – The timer object

Returns
Remaining time until expiration, in ticks

static inline uint32_t **k_timer_remaining_get**(struct k_timer *timer)

Get time remaining before a timer next expires.

This routine computes the (approximate) time remaining before a running timer next expires. If the timer is not running, it returns zero.

Parameters

- **timer** – Address of timer.

Returns
Remaining time (in milliseconds).

void **k_timer_user_data_set**(struct k_timer *timer, void *user_data)

Associate user-specific data with a timer.

This routine records the *user_data* with the *timer*, to be retrieved later.

It can be used e.g. in a timer handler shared across multiple subsystems to retrieve data specific to the subsystem this timer is associated with.

Parameters

- **timer** – Address of timer.
- **user_data** – User data to associate with the timer.

void ***k_timer_user_data_get**(const struct k_timer *timer)

Retrieve the user-specific data from a timer.

Parameters

- **timer** – Address of timer.

Returns
The user data.

3.1.5 Other

These pages cover other kernel services.

Atomic Services

An *atomic variable* is one that can be read and modified by threads and ISRs in an uninterruptible manner. It is a 32-bit variable on 32-bit machines and a 64-bit variable on 64-bit machines.

- *Concepts*
- *Implementation*
  - *Defining an Atomic Variable*
  - *Manipulating an Atomic Variable*
  - *Manipulating an Array of Atomic Variables*
  - *Memory Ordering*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

Concepts Any number of atomic variables can be defined (limited only by available RAM).

Using the kernel's atomic APIs to manipulate an atomic variable guarantees that the desired operation occurs correctly, even if higher priority contexts also manipulate the same variable.

The kernel also supports the atomic manipulation of a single bit in an array of atomic variables.

Implementation

Defining an Atomic Variable An atomic variable is defined using a variable of type `atomic_t`.

By default an atomic variable is initialized to zero. However, it can be given a different value using *ATOMIC_INIT*:

```
atomic_t flags = ATOMIC_INIT(0xFF);
```

Manipulating an Atomic Variable An atomic variable is manipulated using the APIs listed at the end of this section.

The following code shows how an atomic variable can be used to keep track of the number of times a function has been invoked. Since the count is incremented atomically, there is no risk that it will become corrupted in mid-increment if a thread calling the function is interrupted if by a higher priority context that also calls the routine.

```
atomic_t call_count;

int call_counting_routine(void)
{
    /* increment invocation counter */
    atomic_inc(&call_count);

    /* do rest of routine's processing */
    ...
}
```

Manipulating an Array of Atomic Variables An array of 32-bit atomic variables can be defined in the conventional manner. However, you can also define an N-bit array of atomic variables using *ATOMIC_DEFINE*.

A single bit in array of atomic variables can be manipulated using the APIs listed at the end of this section that end with `_bit()`.

The following code shows how a set of 200 flag bits can be implemented using an array of atomic variables.

```
#define NUM_FLAG_BITS 200

ATOMIC_DEFINE(flag_bits, NUM_FLAG_BITS);

/* set specified flag bit & return its previous value */
int set_flag_bit(int bit_position)
{
    return (int)atomic_set_bit(flag_bits, bit_position);
}
```

Memory Ordering For consistency and correctness, all Zephyr atomic APIs are expected to include a full memory barrier (in the sense of e.g. "serializing" instructions on x86, "DMB" on ARM, or a "sequentially consistent" operation as defined by the C++ memory model) where needed by hardware to guarantee a reliable picture across contexts. Any architecture-specific implementations are responsible for ensuring this behavior.

Suggested Uses Use an atomic variable to implement critical section processing that only requires the manipulation of a single 32-bit value.

Use multiple atomic variables to implement critical section processing on a set of flag bits in a bit array longer than 32 bits.

Note

Using atomic variables is typically far more efficient than using other techniques to implement critical sections such as using a mutex or locking interrupts.

Configuration Options Related configuration options:

- `CONFIG_ATOMIC_OPERATIONS_BUILTIN`
- `CONFIG_ATOMIC_OPERATIONS_ARCH`
- `CONFIG_ATOMIC_OPERATIONS_C`

Important

All atomic services APIs can be used by both threads and ISRs.

API Reference

group `atomic_apis`

Defines

`ATOMIC_INIT`(i)

Initialize an atomic variable.

This macro can be used to initialize an atomic variable. For example,

```
atomic_t my_var = ATOMIC_INIT(75);
```

Parameters

- `i` – Value to assign to atomic variable.

`ATOMIC_PTR_INIT`(p)

Initialize an atomic pointer variable.

This macro can be used to initialize an atomic pointer variable. For example,

```
atomic_ptr_t my_ptr = ATOMIC_PTR_INIT(&data);
```

Parameters

- `p` – Pointer value to assign to atomic pointer variable.

`ATOMIC_BITMAP_SIZE`(num_bits)

This macro computes the number of atomic variables necessary to represent a bitmap with *num_bits*.

Parameters

- `num_bits` – Number of bits.

`ATOMIC_DEFINE`(name, num_bits)

Define an array of atomic variables.

This macro defines an array of atomic variables containing at least *num_bits* bits.

Note

If used from file scope, the bits of the array are initialized to zero; if used from within a function, the bits are left uninitialized.

Parameters

- `name` – Name of array of atomic variables.
- `num_bits` – Number of bits needed.

Functions

static inline bool **atomic_test_bit**(const atomic_t *target, int bit)

Atomically test a bit.

This routine tests whether bit number *bit* of *target* is set or not. The target may be a single atomic variable or an array of them.

> **Note**
>
> As for all atomic APIs, includes a full/sequentially-consistent memory barrier (where applicable).

Parameters

- **target** – Address of atomic variable or array.
- **bit** – Bit number (starting from 0).

Returns

true if the bit was set, false if it wasn't.

static inline bool **atomic_test_and_clear_bit**(atomic_t *target, int bit)

Atomically test and clear a bit.

Atomically clear bit number *bit* of *target* and return its old value. The target may be a single atomic variable or an array of them.

> **Note**
>
> As for all atomic APIs, includes a full/sequentially-consistent memory barrier (where applicable).

Parameters

- **target** – Address of atomic variable or array.
- **bit** – Bit number (starting from 0).

Returns

false if the bit was already cleared, true if it wasn't.

static inline bool **atomic_test_and_set_bit**(atomic_t *target, int bit)

Atomically set a bit.

Atomically set bit number *bit* of *target* and return its old value. The target may be a single atomic variable or an array of them.

> **Note**
>
> As for all atomic APIs, includes a full/sequentially-consistent memory barrier (where applicable).

Parameters

- **target** – Address of atomic variable or array.
- **bit** – Bit number (starting from 0).

Returns
true if the bit was already set, false if it wasn't.

static inline void **atomic_clear_bit**(atomic_t *target, int bit)

Atomically clear a bit.

Atomically clear bit number *bit* of *target*. The target may be a single atomic variable or an array of them.

> **Note**
> As for all atomic APIs, includes a full/sequentially-consistent memory barrier (where applicable).

Parameters

- **target** – Address of atomic variable or array.
- **bit** – Bit number (starting from 0).

static inline void **atomic_set_bit**(atomic_t *target, int bit)

Atomically set a bit.

Atomically set bit number *bit* of *target*. The target may be a single atomic variable or an array of them.

> **Note**
> As for all atomic APIs, includes a full/sequentially-consistent memory barrier (where applicable).

Parameters

- **target** – Address of atomic variable or array.
- **bit** – Bit number (starting from 0).

static inline void **atomic_set_bit_to**(atomic_t *target, int bit, bool val)

Atomically set a bit to a given value.

Atomically set bit number *bit* of *target* to value *val*. The target may be a single atomic variable or an array of them.

> **Note**
> As for all atomic APIs, includes a full/sequentially-consistent memory barrier (where applicable).

Parameters

- **target** – Address of atomic variable or array.
- **bit** – Bit number (starting from 0).
- **val** – true for 1, false for 0.

bool **atomic_cas**(atomic_t *target, atomic_val_t old_value, atomic_val_t new_value)

Atomic compare-and-set.

This routine performs an atomic compare-and-set on *target*. If the current value of *target* equals *old_value*, *target* is set to *new_value*. If the current value of *target* does not equal *old_value*, *target* is left unchanged.

> **Note**
>
> As for all atomic APIs, includes a full/sequentially-consistent memory barrier (where applicable).

Parameters

- **target** – Address of atomic variable.
- **old_value** – Original value to compare against.
- **new_value** – New value to store.

Returns

true if *new_value* is written, false otherwise.

bool **atomic_ptr_cas**(atomic_ptr_t *target, atomic_ptr_val_t old_value, atomic_ptr_val_t new_value)

Atomic compare-and-set with pointer values.

This routine performs an atomic compare-and-set on *target*. If the current value of *target* equals *old_value*, *target* is set to *new_value*. If the current value of *target* does not equal *old_value*, *target* is left unchanged.

> **Note**
>
> As for all atomic APIs, includes a full/sequentially-consistent memory barrier (where applicable).

Parameters

- **target** – Address of atomic variable.
- **old_value** – Original value to compare against.
- **new_value** – New value to store.

Returns

true if *new_value* is written, false otherwise.

atomic_val_t **atomic_add**(atomic_t *target, atomic_val_t value)

Atomic addition.

This routine performs an atomic addition on *target*.

> **Note**
>
> As for all atomic APIs, includes a full/sequentially-consistent memory barrier (where applicable).

Parameters

- **target** – Address of atomic variable.

- **value** – Value to add.

Returns
Previous value of *target*.

atomic_val_t **atomic_sub**(atomic_t *target, atomic_val_t value)

Atomic subtraction.

This routine performs an atomic subtraction on *target*.

> **Note**
>
> As for all atomic APIs, includes a full/sequentially-consistent memory barrier (where applicable).

Parameters

- **target** – Address of atomic variable.
- **value** – Value to subtract.

Returns
Previous value of *target*.

atomic_val_t **atomic_inc**(atomic_t *target)

Atomic increment.

This routine performs an atomic increment by 1 on *target*.

> **Note**
>
> As for all atomic APIs, includes a full/sequentially-consistent memory barrier (where applicable).

Parameters

- **target** – Address of atomic variable.

Returns
Previous value of *target*.

atomic_val_t **atomic_dec**(atomic_t *target)

Atomic decrement.

This routine performs an atomic decrement by 1 on *target*.

> **Note**
>
> As for all atomic APIs, includes a full/sequentially-consistent memory barrier (where applicable).

Parameters

- **target** – Address of atomic variable.

Returns
Previous value of *target*.

atomic_val_t **atomic_get**(const atomic_t *target)

Atomic get.

This routine performs an atomic read on *target*.

> **Note**
>
> As for all atomic APIs, includes a full/sequentially-consistent memory barrier (where applicable).

Parameters

- **target** – Address of atomic variable.

Returns

Value of *target*.

atomic_ptr_val_t **atomic_ptr_get**(const atomic_ptr_t *target)

Atomic get a pointer value.

This routine performs an atomic read on *target*.

> **Note**
>
> As for all atomic APIs, includes a full/sequentially-consistent memory barrier (where applicable).

Parameters

- **target** – Address of pointer variable.

Returns

Value of *target*.

atomic_val_t **atomic_set**(atomic_t *target, atomic_val_t value)

Atomic get-and-set.

This routine atomically sets *target* to *value* and returns the previous value of *target*.

> **Note**
>
> As for all atomic APIs, includes a full/sequentially-consistent memory barrier (where applicable).

Parameters

- **target** – Address of atomic variable.
- **value** – Value to write to *target*.

Returns

Previous value of *target*.

atomic_ptr_val_t **atomic_ptr_set**(atomic_ptr_t *target, atomic_ptr_val_t value)

Atomic get-and-set for pointer values.

This routine atomically sets *target* to *value* and returns the previous value of *target*.

> **Note**
>
> As for all atomic APIs, includes a full/sequentially-consistent memory barrier (where applicable).

Parameters

- `target` – Address of atomic variable.
- `value` – Value to write to *target*.

Returns
Previous value of *target*.

atomic_val_t **atomic_clear**(atomic_t *target)

Atomic clear.

This routine atomically sets *target* to zero and returns its previous value. (Hence, it is equivalent to atomic_set(target, 0).)

> **Note**
>
> As for all atomic APIs, includes a full/sequentially-consistent memory barrier (where applicable).

Parameters

- `target` – Address of atomic variable.

Returns
Previous value of *target*.

atomic_ptr_val_t **atomic_ptr_clear**(atomic_ptr_t *target)

Atomic clear of a pointer value.

This routine atomically sets *target* to zero and returns its previous value. (Hence, it is equivalent to atomic_set(target, 0).)

> **Note**
>
> As for all atomic APIs, includes a full/sequentially-consistent memory barrier (where applicable).

Parameters

- `target` – Address of atomic variable.

Returns
Previous value of *target*.

atomic_val_t **atomic_or**(atomic_t *target, atomic_val_t value)

Atomic bitwise inclusive OR.

This routine atomically sets *target* to the bitwise inclusive OR of *target* and *value*.

> **Note**
>
> As for all atomic APIs, includes a full/sequentially-consistent memory barrier (where applicable).

Parameters

- `target` – Address of atomic variable.
- `value` – Value to OR.

Returns
Previous value of *target*.

atomic_val_t **atomic_xor**(atomic_t *target, atomic_val_t value)

Atomic bitwise exclusive OR (XOR).

This routine atomically sets *target* to the bitwise exclusive OR (XOR) of *target* and *value*.

> **Note**
>
> As for all atomic APIs, includes a full/sequentially-consistent memory barrier (where applicable).

Parameters

- `target` – Address of atomic variable.
- `value` – Value to XOR

Returns
Previous value of *target*.

atomic_val_t **atomic_and**(atomic_t *target, atomic_val_t value)

Atomic bitwise AND.

This routine atomically sets *target* to the bitwise AND of *target* and *value*.

> **Note**
>
> As for all atomic APIs, includes a full/sequentially-consistent memory barrier (where applicable).

Parameters

- `target` – Address of atomic variable.
- `value` – Value to AND.

Returns
Previous value of *target*.

atomic_val_t **atomic_nand**(atomic_t *target, atomic_val_t value)

Atomic bitwise NAND.

This routine atomically sets *target* to the bitwise NAND of *target* and *value*. (This operation is equivalent to target = ~(target & value).)

Note

As for all atomic APIs, includes a full/sequentially-consistent memory barrier (where applicable).

Parameters

- `target` – Address of atomic variable.
- `value` – Value to NAND.

Returns
Previous value of *target*.

Floating Point Services

The kernel allows threads to use floating point registers on board configurations that support these registers.

Note

Floating point services are currently available only for boards based on ARM Cortex-M SoCs supporting the Floating Point Extension, the Intel x86 architecture, the SPARC architecture and ARCv2 SoCs supporting the Floating Point Extension. The services provided are architecture specific.

The kernel does not support the use of floating point registers by ISRs.

- *Concepts*
  - *No FP registers mode*
  - *Unshared FP registers mode*
  - *Shared FP registers mode*
- *Implementation*
  - *Performing Floating Point Arithmetic*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

Concepts The kernel can be configured to provide only the floating point services required by an application. Three modes of operation are supported, which are described below. In addition, the kernel's support for the SSE registers can be included or omitted, as desired.

No FP registers mode This mode is used when the application has no threads that use floating point registers. It is the kernel's default floating point services mode.

If a thread uses any floating point register, the kernel generates a fatal error condition and aborts the thread.

Unshared FP registers mode This mode is used when the application has only a single thread that uses floating point registers.

On x86 platforms, the kernel initializes the floating point registers so they can be used by any thread (initialization in skipped on ARM Cortex-M platforms and ARCv2 platforms). The floating point registers are left unchanged whenever a context switch occurs.

> **Note**
>
> The behavior is undefined, if two or more threads attempt to use the floating point registers, as the kernel does not attempt to detect (or prevent) multiple threads from using these registers.

Shared FP registers mode This mode is used when the application has two or more threads that use floating point registers. Depending upon the underlying CPU architecture, the kernel supports one or more of the following thread sub-classes:

- non-user: A thread that cannot use any floating point registers
- FPU user: A thread that can use the standard floating point registers
- SSE user: A thread that can use both the standard floating point registers and SSE registers

The kernel initializes and enables access to the floating point registers, so they can be used by any thread, then saves and restores these registers during context switches to ensure the computations performed by each FPU user or SSE user are not impacted by the computations performed by the other users.

> **Note**
>
> The Shared FP registers mode is the default Floating Point Services mode in ARM Cortex-M.

ARM Cortex-M architecture (with the Floating Point Extension) On the ARM Cortex-M architecture with the Floating Point Extension, the kernel treats *all* threads as FPU users when shared FP registers mode is enabled. This means that any thread is allowed to access the floating point registers. The ARM kernel automatically detects that a given thread is using the floating point registers the first time the thread accesses them.

Pretag a thread that intends to use the FP registers by using one of the techniques listed below.

- A statically-created ARM thread can be pretagged by passing the `K_FP_REGS` option to `K_THREAD_DEFINE`.
- A dynamically-created ARM thread can be pretagged by passing the `K_FP_REGS` option to `k_thread_create()`.

Pretagging a thread with the `K_FP_REGS` option instructs the MPU-based stack protection mechanism to properly configure the size of the thread's guard region to always guarantee stack overflow detection, and enable lazy stacking for the given thread upon thread creation.

During thread context switching the ARM kernel saves the *callee-saved* floating point registers, if the switched-out thread has been using them. Additionally, the *caller-saved* floating point registers are saved on the thread's stack. If the switched-in thread has been using the floating point registers, the kernel restores the *callee-saved* FP registers of the switched-in thread and the *caller-saved* FP context is restored from the thread's stack. Thus, the kernel does not save or restore the FP context of threads that are not using the FP registers.

Each thread that intends to use the floating point registers must provide an extra 72 bytes of stack space where the callee-saved FP context can be saved.

Lazy Stacking is currently enabled in Zephyr applications on ARM Cortex-M architecture, minimizing interrupt latency, when the floating point context is active.

When the MPU-based stack protection mechanism is not enabled, lazy stacking is always active in the Zephyr application. When the MPU-based stack protection is enabled, the following rules apply with respect to lazy stacking:

- Lazy stacking is activated by default on threads that are pretagged with *K_FP_REGS*
- Lazy stacking is activated dynamically on threads that are not pretagged with *K_FP_REGS*, as soon as the kernel detects that they are using the floating point registers.

If an ARM thread does not require use of the floating point registers any more, it can call *k_float_disable()*. This instructs the kernel not to save or restore its FP context during thread context switching.

Note

The Shared FP registers mode is the default Floating Point Services mode on ARM64. The compiler is free to optimize code using FP/SIMD registers, and library functions such as memcpy are known to make use of them.

ARM64 architecture On the ARM64 (Aarch64) architecture the kernel treats each thread as a FPU user on a case-by-case basis. A "lazy save" algorithm is used during context switching which updates the floating point registers only when it is absolutely necessary. For example, the registers are *not* saved when switching from an FPU user to a non-user thread, and then back to the original FPU user.

FPU register usage by ISRs is supported although not recommended. When an ISR uses floating point or SIMD registers, then the access is trapped, the current FPU user context is saved in the thread object and the ISR is resumed with interrupts disabled so to prevent another IRQ from interrupting the ISR and potentially requesting FPU usage. Because ISR don't have a persistent register context, there are no provision for saving an ISR's FPU context either, hence the IRQ disabling.

Each thread object becomes 512 bytes larger when Shared FP registers mode is enabled.

ARCv2 architecture On the ARCv2 architecture, the kernel treats each thread as a non-user or FPU user and the thread must be tagged by one of the following techniques.

- A statically-created ARC thread can be tagged by passing the *K_FP_REGS* option to *K_THREAD_DEFINE*.
- A dynamically-created ARC thread can be tagged by passing the *K_FP_REGS* to *k_thread_create()*.

If an ARC thread does not require use of the floating point registers any more, it can call *k_float_disable()*. This instructs the kernel not to save or restore its FP context during thread context switching.

During thread context switching the ARC kernel saves the *callee-saved* floating point registers, if the switched-out thread has been using them. Additionally, the *caller-saved* floating point registers are saved on the thread's stack. If the switched-in thread has been using the floating point registers, the kernel restores the *callee-saved* FP registers of the switched-in thread and the *caller-saved* FP context is restored from the thread's stack. Thus, the kernel does not save or restore the FP context of threads that are not using the FP registers. An extra 16 bytes (single floating point hardware) or 32 bytes (double floating point hardware) of stack space is required to load and store floating point registers.

RISC-V architecture On the RISC-V architecture the kernel treats each thread as an FPU user on a case-by-case basis with the FPU access allocated on demand. A "lazy save" algorithm is used during context switching which updates the floating point registers only when it is absolutely necessary. For example, the FPU registers are *not* saved when switching from an FPU user to a non-user thread (or an FPU user that doesn't touch the FPU during its scheduling slot), and then back to the original FPU user.

FPU register usage by ISRs is supported although not recommended. When an ISR uses floating point or SIMD registers, then the access is trapped, the current FPU user context is saved in the thread object and the ISR is resumed with interrupts disabled so to prevent another IRQ from interrupting the ISR and potentially requesting FPU usage. Because ISR don't have a persistent register context, there are no provision for saving an ISR's FPU context either, hence the IRQ disabling.

As an optimization, the FPU context is preemptively restored upon scheduling back an "active FPU user" thread that had its FPU context saved away due to FPU usage by another thread. Active FPU users are so designated when they make the FPU state "dirty" during their most recent scheduling slot before being scheduled out. So if a thread doesn't modify the FPU state within its scheduling slot and another thread claims the FPU for itself afterwards then that first thread will be subjected to the on-demand regime and won't have its FPU context restored until it attempts to access it again. But if that thread does modify the FPU before being scheduled out then it is likely to continue using it when scheduled back in and preemptively restoring its FPU context saves on the exception trap overhead that would occur otherwise.

Each thread object becomes 136 bytes (single-precision floating point hardware) or 264 bytes (double-precision floating point hardware) larger when Shared FP registers mode is enabled.

SPARC architecture On the SPARC architecture, the kernel treats each thread as a non-user or FPU user and the thread must be tagged by one of the following techniques:

- A statically-created thread can be tagged by passing the *K_FP_REGS* option to *K_THREAD_DEFINE*.
- A dynamically-created thread can be tagged by passing the *K_FP_REGS* to *k_thread_create()*.

During thread context switch at exit from interrupt handler, the SPARC kernel saves *all* floating point registers, if the FPU was enabled in the switched-out thread. Floating point registers are saved on the thread's stack. Floating point registers are restored when a thread context is restored iff they were saved at the context save. Saving and restoring of the floating point registers is synchronous and thus not lazy. The FPU is always disabled when an ISR is called (independent of CONFIG_FPU_SHARING).

Floating point disabling with *k_float_disable()* is not implemented.

When CONFIG_FPU_SHARING is used, then 136 bytes of stack space is required for each FPU user thread to load and store floating point registers. No extra stack is required if CONFIG_FPU_SHARING is not used.

x86 architecture On the x86 architecture the kernel treats each thread as a non-user, FPU user or SSE user on a case-by-case basis. A "lazy save" algorithm is used during context switching which updates the floating point registers only when it is absolutely necessary. For example, the registers are *not* saved when switching from an FPU user to a non-user thread, and then back to the original FPU user. The following table indicates the amount of additional stack space a thread must provide so the registers can be saved properly.

Thread type	FP register use	Extra stack space required
cooperative	any	0 bytes
preemptive	none	0 bytes
preemptive	FPU	108 bytes
preemptive	SSE	464 bytes

The x86 kernel automatically detects that a given thread is using the floating point registers the first time the thread accesses them. The thread is tagged as an SSE user if the kernel has been configured to support the SSE registers, or as an FPU user if the SSE registers are not supported. If this would result in a thread that is an FPU user being tagged as an SSE user, or if the application wants to avoid the exception handling overhead involved in auto-tagging threads, it is possible to pretag a thread using one of the techniques listed below.

- A statically-created x86 thread can be pretagged by passing the *K_FP_REGS* or *K_SSE_REGS* option to *K_THREAD_DEFINE*.
- A dynamically-created x86 thread can be pretagged by passing the *K_FP_REGS* or *K_SSE_REGS* option to *k_thread_create()*.
- An already-created x86 thread can pretag itself once it has started by passing the *K_FP_REGS* or *K_SSE_REGS* option to *k_float_enable()*.

If an x86 thread uses the floating point registers infrequently it can call *k_float_disable()* to remove its tagging as an FPU user or SSE user. This eliminates the need for the kernel to take steps to preserve the contents of the floating point registers during context switches when there is no need to do so. When the thread again needs to use the floating point registers it can re-tag itself as an FPU user or SSE user by calling *k_float_enable()*.

Implementation

Performing Floating Point Arithmetic No special coding is required for a thread to use floating point arithmetic if the kernel is properly configured.

The following code shows how a routine can use floating point arithmetic to avoid overflow issues when computing the average of a series of integer values.

```
int average(int *values, int num_values)
{
    double sum;
    int i;

    sum = 0.0;

    for (i = 0; i < num_values; i++) {
        sum += *values;
        values++;
    }

    return (int)((sum / num_values) + 0.5);
}
```

Suggested Uses Use the kernel floating point services when an application needs to perform floating point operations.

Configuration Options To configure unshared FP registers mode, enable the CONFIG_FPU configuration option and leave the CONFIG_FPU_SHARING configuration option disabled.

To configure shared FP registers mode, enable both the CONFIG_FPU configuration option and the CONFIG_FPU_SHARING configuration option. Also, ensure that any thread that uses the floating point registers has sufficient added stack space for saving floating point register values during context switches, as described above.

For x86, use the CONFIG_X86_SSE configuration option to enable support for SSEx instructions.

API Reference

group `float_apis`

Functions

int `k_float_disable`(struct *k_thread* *thread)

Disable preservation of floating point context information.

This routine informs the kernel that the specified thread will no longer be using the floating point registers.

Warning

Some architectures apply restrictions on how the disabling of floating point preservation may be requested, see arch_float_disable.

Warning

This routine should only be used to disable floating point support for a thread that currently has such support enabled.

Parameters

- `thread` – ID of thread.

Return values

- `0` – On success.
- `-ENOTSUP` – If the floating point disabling is not implemented. -EINVAL If the floating point disabling could not be performed.

int `k_float_enable`(struct *k_thread* *thread, unsigned int options)

Enable preservation of floating point context information.

This routine informs the kernel that the specified thread will use the floating point registers.

Invoking this routine initializes the thread's floating point context info to that of an FPU that has been reset. The next time the thread is scheduled by z_swap() it will either inherit an FPU that is guaranteed to be in a "sane" state (if the most recent user of the FPU was cooperatively swapped out) or the thread's own floating point context will be loaded (if the most recent user of the FPU was preempted, or if this thread is the first user of the FPU). Thereafter, the kernel will protect the thread's FP context so that it is not altered during a preemptive context switch.

The *options* parameter indicates which floating point register sets will be used by the specified thread.

For x86 options:

- K_FP_REGS indicates x87 FPU and MMX registers only
- K_SSE_REGS indicates SSE registers (and also x87 FPU and MMX registers)

Warning

Some architectures apply restrictions on how the enabling of floating point preservation may be requested, see arch_float_enable.

Warning

This routine should only be used to enable floating point support for a thread that currently has such support enabled.

Parameters

- `thread` – ID of thread.
- `options` – architecture dependent options

Return values

- `0` – On success.
- `-ENOTSUP` – If the floating point enabling is not implemented. -EINVAL If the floating point enabling could not be performed.

Version

Kernel version handling and APIs related to kernel version being used.

API Reference

uint32_t `sys_kernel_version_get`(void)

Return the kernel version of the present build.

The kernel version is a four-byte value, whose format is described in the file "kernel_version.h".

Returns

kernel version

`SYS_KERNEL_VER_MAJOR`(ver)

`SYS_KERNEL_VER_MINOR`(ver)

`SYS_KERNEL_VER_PATCHLEVEL`(ver)

Fatal Errors

Software Errors Triggered in Source Code Zephyr provides several methods for inducing fatal error conditions through either build-time checks, conditionally compiled assertions, or deliberately invoked panic or oops conditions.

Runtime Assertions Zephyr provides some macros to perform runtime assertions which may be conditionally compiled. Their definitions may be found in include/zephyr/sys/__assert.h.

Assertions are enabled by setting the `__ASSERT_ON` preprocessor symbol to a non-zero value. There are two ways to do this:

- Use the `CONFIG_ASSERT` and `CONFIG_ASSERT_LEVEL` kconfig options.
- Add `-D__ASSERT_ON=<level>` to the project's CFLAGS, either on the build command line or in a CMakeLists.txt.

The `__ASSERT_ON` method takes precedence over the kconfig option if both are used.

Specifying an assertion level of 1 causes the compiler to issue warnings that the kernel contains debug-type `__ASSERT()` statements; this reminder is issued since assertion code is not normally present in a final product. Specifying assertion level 2 suppresses these warnings.

Assertions are enabled by default when running Zephyr test cases, as configured by the `CONFIG_TEST` option.

The policy for what to do when encountering a failed assertion is controlled by the implementation of `assert_post_action()`. Zephyr provides a default implementation with weak linkage which invokes a kernel oops if the thread that failed the assertion was running in user mode, and a kernel panic otherwise.

__ASSERT() The `__ASSERT()` macro can be used inside kernel and application code to perform optional runtime checks which will induce a fatal error if the check does not pass. The macro takes a string message which will be printed to provide context to the assertion. In addition, the kernel will print a text representation of the expression code that was evaluated, and the file and line number where the assertion can be found.

For example:

```
__ASSERT(foo == 0xF0CACC1A, "Invalid value of foo, got 0x%x", foo);
```

If at runtime `foo` had some unexpected value, the error produced may look like the following:

```
ASSERTION FAIL [foo == 0xF0CACC1A] @ ZEPHYR_BASE/tests/kernel/fatal/src/main.c:367
        Invalid value of foo, got 0xdeadbeef
[00:00:00.000,000] <err> os: r0/a1:  0x00000004  r1/a2:  0x0000016f  r2/a3:  0x00000000
[00:00:00.000,000] <err> os: r3/a4:  0x00000000 r12/ip:  0x00000000 r14/lr:  0x00000a6d
[00:00:00.000,000] <err> os:  xpsr:  0x61000000
[00:00:00.000,000] <err> os: Faulting instruction address (r15/pc): 0x00009fe4
[00:00:00.000,000] <err> os: >>> ZEPHYR FATAL ERROR 4: Kernel panic
[00:00:00.000,000] <err> os: Current thread: 0x20000414 (main)
[00:00:00.000,000] <err> os: Halting system
```

__ASSERT_EVAL() The `__ASSERT_EVAL()` macro can also be used inside kernel and application code, with special semantics for the evaluation of its arguments.

It makes use of the `__ASSERT()` macro, but has some extra flexibility. It allows the developer to specify different actions depending whether the `__ASSERT()` macro is enabled or not. This can be particularly useful to prevent the compiler from generating comments (errors, warnings or remarks) about variables that are only used with `__ASSERT()` being assigned a value, but otherwise unused when the `__ASSERT()` macro is disabled.

Consider the following example:

```
int x;
x = foo();
__ASSERT(x != 0, "foo() returned zero!");
```

If __ASSERT() is disabled, then 'x' is assigned a value, but never used. This type of situation can be resolved using the __ASSERT_EVAL() macro.

```
__ASSERT_EVAL ((void) foo(),
               int x = foo(),
               x != 0,
               "foo() returned zero!");
```

The first parameter tells __ASSERT_EVAL() what to do if __ASSERT() is disabled. The second parameter tells __ASSERT_EVAL() what to do if __ASSERT() is enabled. The third and fourth parameters are the parameters it passes to __ASSERT().

__ASSERT_NO_MSG() The __ASSERT_NO_MSG() macro can be used to perform an assertion that reports the failed test and its location, but lacks additional debugging information provided to assist the user in diagnosing the problem; its use is discouraged.

Build Assertions Zephyr provides two macros for performing build-time assertion checks. These are evaluated completely at compile-time, and are always checked.

BUILD_ASSERT() This has the same semantics as C's _Static_assert or C++'s static_assert. If the evaluation fails, a build error will be generated by the compiler. If the compiler supports it, the provided message will be printed to provide further context.

Unlike __ASSERT(), the message must be a static string, without printf()-like format codes or extra arguments.

For example, suppose this check fails:

```
BUILD_ASSERT(FOO == 2000, "Invalid value of FOO");
```

With GCC, the output resembles:

```
tests/kernel/fatal/src/main.c: In function 'test_main':
include/toolchain/gcc.h:28:37: error: static assertion failed: "Invalid value of FOO"
 #define BUILD_ASSERT(EXPR, MSG) _Static_assert(EXPR, "" MSG)
                                 ^~~~~~~~~~~~~~
tests/kernel/fatal/src/main.c:370:2: note: in expansion of macro 'BUILD_ASSERT'
  BUILD_ASSERT(FOO == 2000,
  ^~~~~~~~~~~~~~~~
```

Kernel Oops A kernel oops is a software triggered fatal error invoked by k_oops(). This should be used to indicate an unrecoverable condition in application logic.

The fatal error reason code generated will be K_ERR_KERNEL_OOPS.

Kernel Panic A kernel error is a software triggered fatal error invoked by k_panic(). This should be used to indicate that the Zephyr kernel is in an unrecoverable state. Implementations of *k_sys_fatal_error_handler()* should not return if the kernel encounters a panic condition, as the entire system needs to be reset.

Threads running in user mode are not permitted to invoke k_panic(), and doing so will generate a kernel oops instead. Otherwise, the fatal error reason code generated will be K_ERR_KERNEL_PANIC.

Exceptions

Spurious Interrupts If the CPU receives a hardware interrupt on an interrupt line that has not had a handler installed with `IRQ_CONNECT()` or *irq_connect_dynamic()*, then the kernel will generate a fatal error with the reason code `K_ERR_SPURIOUS_IRQ()`.

Stack Overflows In the event that a thread pushes more data onto its execution stack than its stack buffer provides, the kernel may be able to detect this situation and generate a fatal error with a reason code of `K_ERR_STACK_CHK_FAIL`.

If a thread is running in user mode, then stack overflows are always caught, as the thread will simply not have permission to write to adjacent memory addresses outside of the stack buffer. Because this is enforced by the memory protection hardware, there is no risk of data corruption to memory that the thread would not otherwise be able to write to.

If a thread is running in supervisor mode, or if `CONFIG_USERSPACE` is not enabled, depending on configuration stack overflows may or may not be caught. `CONFIG_HW_STACK_PROTECTION` is supported on some architectures and will catch stack overflows in supervisor mode, including when handling a system call on behalf of a user thread. Typically this is implemented via dedicated CPU features, or read-only MMU/MPU guard regions placed immediately adjacent to the stack buffer. Stack overflows caught in this way can detect the overflow, but cannot guarantee against data corruption and should be treated as a very serious condition impacting the health of the entire system.

If a platform lacks memory management hardware support, `CONFIG_STACK_SENTINEL` is a software-only stack overflow detection feature which periodically checks if a sentinel value at the end of the stack buffer has been corrupted. It does not require hardware support, but provides no protection against data corruption. Since the checks are typically done at interrupt exit, the overflow may be detected a nontrivial amount of time after the stack actually overflowed.

Finally, Zephyr supports GCC compiler stack canaries via `CONFIG_STACK_CANARIES`. If enabled, the compiler will insert a canary value randomly generated at boot into function stack frames, checking that the canary has not been overwritten at function exit. If the check fails, the compiler invokes `__stack_chk_fail()`, whose Zephyr implementation invokes a fatal stack overflow error. An error in this case does not indicate that the entire stack buffer has overflowed, but instead that the current function stack frame has been corrupted. See the compiler documentation for more details.

Other Exceptions Any other type of unhandled CPU exception will generate an error code of `K_ERR_CPU_EXCEPTION`.

Fatal Error Handling The policy for what to do when encountering a fatal error is determined by the implementation of the *k_sys_fatal_error_handler()* function. This function has a default implementation with weak linkage that calls `LOG_PANIC()` to dump all pending logging messages and then unconditionally halts the system with *k_fatal_halt()*.

Applications are free to implement their own error handling policy by overriding the implementation of *k_sys_fatal_error_handler()*. If the implementation returns, the faulting thread will be aborted and the system will otherwise continue to function. See the documentation for this function for additional details and constraints.

API Reference

group `fatal_apis`

Functions

FUNC_NORETURN void **k_fatal_halt**(unsigned int reason)

Halt the system on a fatal error.

Invokes architecture-specific code to power off or halt the system in a low power state. Lacking that, lock interrupts and sit in an idle loop.

Parameters

- **reason** – Fatal exception reason code

void **k_sys_fatal_error_handler**(unsigned int reason, const struct arch_esf *esf)

Fatal error policy handler.

This function is not invoked by application code, but is declared as a weak symbol so that applications may introduce their own policy.

The default implementation of this function halts the system unconditionally. Depending on architecture support, this may be a simple infinite loop, power off the hardware, or exit an emulator.

If this function returns, then the currently executing thread will be aborted.

A few notes for custom implementations:

- If the error is determined to be unrecoverable, *LOG_PANIC()* should be invoked to flush any pending logging buffers.
- K_ERR_KERNEL_PANIC indicates a severe unrecoverable error in the kernel itself, and should not be considered recoverable. There is an assertion in z_fatal_error() to enforce this.
- Even outside of a kernel panic, unless the fault occurred in user mode, the kernel itself may be in an inconsistent state, with API calls to kernel objects possibly exhibiting undefined behavior or triggering another exception.

Parameters

- **reason** – The reason for the fatal error
- **esf** – Exception context, with details and partial or full register state when the error occurred. May in some cases be NULL.

Thread Local Storage (TLS)

Thread Local Storage (TLS) allows variables to be allocated on a per-thread basis. These variables are stored in the thread stack which means every thread has its own copy of these variables.

Zephyr currently requires toolchain support for TLS.

Configuration To enable thread local storage in Zephyr, `CONFIG_THREAD_LOCAL_STORAGE` needs to be enabled. Note that this option may not be available if the architecture or the SoC does not have the hidden option `CONFIG_ARCH_HAS_THREAD_LOCAL_STORAGE` enabled, which means the architecture or the SoC does not have the necessary code to support thread local storage and/or the toolchain does not support TLS.

`CONFIG_ERRNO_IN_TLS` can be enabled together with `CONFIG_ERRNO` to let the variable `errno` be a thread local variable. This allows user threads to access the value of `errno` without making a system call.

Declaring and Using Thread Local Variables The keyword `__thread` can be used to declare thread local variables.

For example, to declare a thread local variable in header files:

```
extern __thread int i;
```

And to declare the actual variable in source files:

```
__thread int i;
```

Keyword `static` can also be used to limit the variable within a source file:

```
static __thread int j;
```

Using the thread local variable is the same as using other variable, for example:

```
void testing(void) {
    i = 10;
}
```

3.2 Device Driver Model

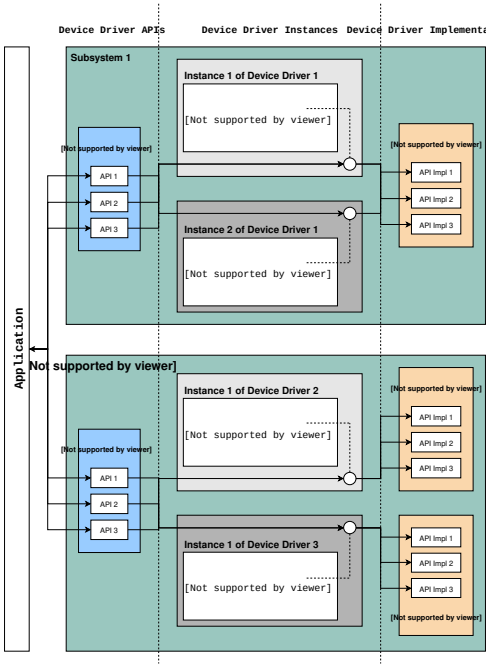
3.2.1 Introduction

The Zephyr kernel supports a variety of device drivers. Whether a driver is available depends on the board and the driver.

The Zephyr device model provides a consistent device model for configuring the drivers that are part of a system. The device model is responsible for initializing all the drivers configured into the system.

Each type of driver (e.g. UART, SPI, I2C) is supported by a generic type API.

In this model the driver fills in the pointer to the structure containing the function pointers to its API functions during driver initialization. These structures are placed into the RAM section in initialization level order.



3.2.2 Standard Drivers

Device drivers which are present on all supported board configurations are listed below.

- **Interrupt controller**: This device driver is used by the kernel's interrupt management subsystem.
- **Timer**: This device driver is used by the kernel's system clock and hardware clock subsystem.
- **Serial communication**: This device driver is used by the kernel's system console subsystem.
- **Entropy**: This device driver provides a source of entropy numbers for the random number generator subsystem.

> **Important**
>
> Use the *random API functions* for random values. *Entropy functions* should not be directly used as a random number generator source as some hardware implementations are designed to be an entropy seed source for random number generators and will not provide cryptographically secure random number streams.

3.2.3 Synchronous Calls

Zephyr provides a set of device drivers for multiple boards. Each driver should support an interrupt-based implementation, rather than polling, unless the specific hardware does not provide any interrupt.

High-level calls accessed through device-specific APIs, such as `i2c.h` or `spi.h`, are usually intended as synchronous. Thus, these calls should be blocking.

3.2.4 Driver APIs

The following APIs for device drivers are provided by `device.h`. The APIs are intended for use in device drivers only and should not be used in applications.

DEVICE_DEFINE()
: Create device object and related data structures including setting it up for boot-time initialization.

DEVICE_NAME_GET()
: Converts a device identifier to the global identifier for a device object.

DEVICE_GET()
: Obtain a pointer to a device object by name.

DEVICE_DECLARE()
: Declare a device object. Use this when you need a forward reference to a device that has not yet been defined.

3.2.5 Driver Data Structures

The device initialization macros populate some data structures at build time which are split into read-only and runtime-mutable parts. At a high level we have:

```
struct device {
      const char *name;
      const void *config;
      const void *api;
      void * const data;
};
```

The `config` member is for read-only configuration data set at build time. For example, base memory mapped IO addresses, IRQ line numbers, or other fixed physical characteristics of the device. This is the `config` pointer passed to `DEVICE_DEFINE()` and related macros.

The `data` struct is kept in RAM, and is used by the driver for per-instance runtime housekeeping. For example, it may contain reference counts, semaphores, scratch buffers, etc.

The `api` struct maps generic subsystem APIs to the device-specific implementations in the driver. It is typically read-only and populated at build time. The next section describes this in more detail.

3.2.6 Subsystems and API Structures

Most drivers will be implementing a device-independent subsystem API. Applications can simply program to that generic API, and application code is not specific to any particular driver implementation.

A subsystem API definition typically looks like this:

```
typedef int (*subsystem_do_this_t)(const struct device *dev, int foo, int bar);
typedef void (*subsystem_do_that_t)(const struct device *dev, void *baz);

struct subsystem_api {
      subsystem_do_this_t do_this;
      subsystem_do_that_t do_that;
};

static inline int subsystem_do_this(const struct device *dev, int foo, int bar)
{
      struct subsystem_api *api;

      api = (struct subsystem_api *)dev->api;
      return api->do_this(dev, foo, bar);
}

static inline void subsystem_do_that(const struct device *dev, void *baz)
{
      struct subsystem_api *api;

      api = (struct subsystem_api *)dev->api;
      api->do_that(dev, baz);
}
```

A driver implementing a particular subsystem will define the real implementation of these APIs, and populate an instance of subsystem_api structure:

```
static int my_driver_do_this(const struct device *dev, int foo, int bar)
{
      ...
}

static void my_driver_do_that(const struct device *dev, void *baz)
{
      ...
```

(continues on next page)

(continued from previous page)

```
}

static struct subsystem_api my_driver_api_funcs = {
      .do_this = my_driver_do_this,
      .do_that = my_driver_do_that
};
```

The driver would then pass `my_driver_api_funcs` as the `api` argument to `DEVICE_DEFINE()`.

> **Note**
>
> Since pointers to the API functions are referenced in the `api` struct, they will always be included in the binary even if unused; `gc-sections` linker option will always see at least one reference to them. Providing for link-time size optimizations with driver APIs in most cases requires that the optional feature be controlled by a Kconfig option.

3.2.7 Device-Specific API Extensions

Some devices can be cast as an instance of a driver subsystem such as GPIO, but provide additional functionality that cannot be exposed through the standard API. These devices combine subsystem operations with device-specific APIs, described in a device-specific header.

A device-specific API definition typically looks like this:

```
#include <zephyr/drivers/subsystem.h>

/* When extensions need not be invoked from user mode threads */
int specific_do_that(const struct device *dev, int foo);

/* When extensions must be invokable from user mode threads */
__syscall int specific_from_user(const struct device *dev, int bar);

/* Only needed when extensions include syscalls */
#include <zephyr/syscalls/specific.h>
```

A driver implementing extensions to the subsystem will define the real implementation of both the subsystem API and the specific APIs:

```
static int generic_do_this(const struct device *dev, void *arg)
{
   ...
}

static struct generic_api api {
   ...
   .do_this = generic_do_this,
   ...
};

/* supervisor-only API is globally visible */
int specific_do_that(const struct device *dev, int foo)
{
   ...
}

/* syscall API passes through a translation */
int z_impl_specific_from_user(const struct device *dev, int bar)
{
```

(continues on next page)

(continued from previous page)

```
    ...
}

#ifdef CONFIG_USERSPACE

#include <zephyr/internal/syscall_handler.h>

int z_vrfy_specific_from_user(const struct device *dev, int bar)
{
    K_OOPS(K_SYSCALL_SPECIFIC_DRIVER(dev, K_OBJ_DRIVER_GENERIC, &api));
    return z_impl_specific_do_that(dev, bar)
}

#include <zephyr/syscalls/specific_from_user_mrsh.c>

#endif /* CONFIG_USERSPACE */
```

Applications use the device through both the subsystem and specific APIs.

> **Note**
>
> Public API for device-specific extensions should be prefixed with the compatible for the device to which it applies. For example, if adding special functions to support the Maxim DS3231 the identifier fragment `specific` in the examples above would be `maxim_ds3231`.

3.2.8 Single Driver, Multiple Instances

Some drivers may be instantiated multiple times in a given system. For example there can be multiple GPIO banks, or multiple UARTS. Each instance of the driver will have a different `config` struct and `data` struct.

Configuring interrupts for multiple drivers instances is a special case. If each instance needs to configure a different interrupt line, this can be accomplished through the use of per-instance configuration functions, since the parameters to `IRQ_CONNECT()` need to be resolvable at build time.

For example, let's say we need to configure two instances of `my_driver`, each with a different interrupt line. In `drivers/subsystem/subsystem_my_driver.h`:

```
typedef void (*my_driver_config_irq_t)(const struct device *dev);

struct my_driver_config {
      DEVICE_MMIO_ROM;
      my_driver_config_irq_t config_func;
};
```

In the implementation of the common init function:

```
void my_driver_isr(const struct device *dev)
{
      /* Handle interrupt */
      ...
}

int my_driver_init(const struct device *dev)
{
      const struct my_driver_config *config = dev->config;
```

(continues on next page)

(continued from previous page)

```
	DEVICE_MMIO_MAP(dev, K_MEM_CACHE_NONE);

	/* Do other initialization stuff */
	...

	config->config_func(dev);

	return 0;
}
```

Then when the particular instance is declared:

```
#if CONFIG_MY_DRIVER_0

DEVICE_DECLARE(my_driver_0);

static void my_driver_config_irq_0(const struct device *dev)
{
	IRQ_CONNECT(MY_DRIVER_0_IRQ, MY_DRIVER_0_PRI, my_driver_isr,
		    DEVICE_GET(my_driver_0), MY_DRIVER_0_FLAGS);
}

const static struct my_driver_config my_driver_config_0 = {
	DEVICE_MMIO_ROM_INIT(DT_DRV_INST(0)),
	.config_func = my_driver_config_irq_0
}

static struct my_data_0;

DEVICE_DEFINE(my_driver_0, MY_DRIVER_0_NAME, my_driver_init,
	      NULL, &my_data_0, &my_driver_config_0,
	      POST_KERNEL, MY_DRIVER_0_PRIORITY, &my_api_funcs);

#endif /* CONFIG_MY_DRIVER_0 */
```

Note the use of `DEVICE_DECLARE()` to avoid a circular dependency on providing the IRQ handler argument and the definition of the device itself.

3.2.9 Initialization Levels

Drivers may depend on other drivers being initialized first, or require the use of kernel services. *DEVICE_DEFINE()* and related APIs allow the user to specify at what time during the boot sequence the init function will be executed. Any driver will specify one of four initialization levels:

PRE_KERNEL_1
: Used for devices that have no dependencies, such as those that rely solely on hardware present in the processor/SOC. These devices cannot use any kernel services during configuration, since the kernel services are not yet available. The interrupt subsystem will be configured however so it's OK to set up interrupts. Init functions at this level run on the interrupt stack.

PRE_KERNEL_2
: Used for devices that rely on the initialization of devices initialized as part of the `PRE_KERNEL_1` level. These devices cannot use any kernel services during configuration, since the kernel services are not yet available. Init functions at this level run on the interrupt stack.

POST_KERNEL
: Used for devices that require kernel services during configuration. Init functions at this

level run in context of the kernel main task.

Within each initialization level you may specify a priority level, relative to other devices in the same initialization level. The priority level is specified as an integer value in the range 0 to 99; lower values indicate earlier initialization. The priority level must be a decimal integer literal without leading zeroes or sign (e.g. 32), or an equivalent symbolic name (e.g. `\#define MY_INIT_PRIO 32`); symbolic expressions are *not* permitted (e.g. `CONFIG_KERNEL_INIT_PRIORITY_DEFAULT + 5`).

Drivers and other system utilities can determine whether startup is still in pre-kernel states by using the *k_is_pre_kernel()* function.

3.2.10 Deferred initialization

Initialization of devices can also be deferred to a later time. In this case, the device is not automatically initialized by Zephyr at boot time. Instead, the device is initialized when the application calls *device_init()*. To defer a device driver initialization, add the property `zephyr,deferred-init` to the associated device node in the DTS file. For example:

```
/ {
        a-driver@40000000 {
                reg = <0x40000000 0x1000>;
                zephyr,deferred-init;
        };
};
```

3.2.11 System Drivers

In some cases you may just need to run a function at boot. For such cases, the `SYS_INIT` can be used. This macro does not take any config or runtime data structures and there isn't a way to later get a device pointer by name. The same device policies for initialization level and priority apply.

3.2.12 Inspecting the initialization sequence

Device drivers declared with *DEVICE_DEFINE* (or any variations of it) and `SYS_INIT` are processed at boot time and the corresponding initialization functions are called sequentially according to their specified level and priority.

Sometimes it's useful to inspect the final sequence of initialization function call as produced by the linker. To do that, use the `initlevels` CMake target, for example `west build -t initlevels`.

3.2.13 Error handling

In general, it's best to use `__ASSERT()` macros instead of propagating return values unless the failure is expected to occur during the normal course of operation (such as a storage device full). Bad parameters, programming errors, consistency checks, pathological/unrecoverable failures, etc., should be handled by assertions.

When it is appropriate to return error conditions for the caller to check, 0 should be returned on success and a POSIX `errno.h` code returned on failure. See https://github.com/zephyrproject-rtos/zephyr/wiki/Naming-Conventions#return-codes for details about this.

3.2.14 Memory Mapping

On some systems, the linear address of peripheral memory-mapped I/O (MMIO) regions cannot be known at build time:

- The I/O ranges must be probed at runtime from the bus, such as with PCI express
- A memory management unit (MMU) is active, and the physical address of the MMIO range must be mapped into the page tables at some virtual memory location determined by the kernel.

These systems must maintain storage for the MMIO range within RAM and establish the mapping within the driver's init function. Other systems do not care about this and can use MMIO physical addresses directly from DTS and do not need any RAM-based storage for it.

For drivers that may need to deal with this situation, a set of APIs under the DEVICE_MMIO scope are defined, along with a mapping function `device_map()`.

Device Model Drivers with one MMIO region

The simplest case is for drivers which need to maintain one MMIO region. These drivers will need to use the `DEVICE_MMIO_ROM` and `DEVICE_MMIO_RAM` macros in the definitions for their `config_info` and `driver_data` structures, with initialization of the `config_info` from DTS using `DEVICE_MMIO_ROM_INIT`. A call to `DEVICE_MMIO_MAP()` is made within the init function:

```
struct my_driver_config {
   DEVICE_MMIO_ROM; /* Must be first */
   ...
}

struct my_driver_dev_data {
   DEVICE_MMIO_RAM; /* Must be first */
   ...
}

const static struct my_driver_config my_driver_config_0 = {
   DEVICE_MMIO_ROM_INIT(DT_DRV_INST(...)),
   ...
}

int my_driver_init(const struct device *dev)
{
   ...
   DEVICE_MMIO_MAP(dev, K_MEM_CACHE_NONE);
   ...
}

int my_driver_some_function(const struct device *dev)
{
   ...
   /* Write some data to the MMIO region */
   sys_write32(0xDEADBEEF, DEVICE_MMIO_GET(dev));
   ...
}
```

The particular expansion of these macros depends on configuration. On a device with no MMU or PCI-e, `DEVICE_MMIO_MAP` and `DEVICE_MMIO_RAM` expand to nothing.

Device Model Drivers with multiple MMIO regions

Some drivers may have multiple MMIO regions. In addition, some drivers may already be implementing a form of inheritance which requires some other data to be placed first in the `config_info` and `driver_data` structures.

This can be managed with the `DEVICE_MMIO_NAMED` variant macros. These require that `DEV_CFG()` and `DEV_DATA()` macros be defined to obtain a properly typed pointer to the driver's config_info or dev_data structs. For example:

```
struct my_driver_config {
   ...
     DEVICE_MMIO_NAMED_ROM(corge);
     DEVICE_MMIO_NAMED_ROM(grault);
   ...
}

struct my_driver_dev_data {
        ...
     DEVICE_MMIO_NAMED_RAM(corge);
     DEVICE_MMIO_NAMED_RAM(grault);
     ...
}

#define DEV_CFG(_dev) \
   ((const struct my_driver_config *)((_dev)->config))

#define DEV_DATA(_dev) \
   ((struct my_driver_dev_data *)((_dev)->data))

const static struct my_driver_config my_driver_config_0 = {
   ...
   DEVICE_MMIO_NAMED_ROM_INIT(corge, DT_DRV_INST(...)),
   DEVICE_MMIO_NAMED_ROM_INIT(grault, DT_DRV_INST(...)),
   ...
}

int my_driver_init(const struct device *dev)
{
   ...
   DEVICE_MMIO_NAMED_MAP(dev, corge, K_MEM_CACHE_NONE);
   DEVICE_MMIO_NAMED_MAP(dev, grault, K_MEM_CACHE_NONE);
   ...
}

int my_driver_some_function(const struct device *dev)
{
   ...
   /* Write some data to the MMIO regions */
   sys_write32(0xDEADBEEF, DEVICE_MMIO_GET(dev, grault));
   sys_write32(0xF0CCAC1A, DEVICE_MMIO_GET(dev, corge));
   ...
}
```

Device Model Drivers with multiple MMIO regions in the same DT node

Some drivers may have multiple MMIO regions defined into the same DT device node using the `reg-names` property to differentiate them, for example:

```
/dts-v1/;

/ {
        a-driver@40000000 {
                reg = <0x40000000 0x1000>,
                      <0x40001000 0x1000>;
                reg-names = "corge", "grault";
        };
};
```

This can be managed as seen in the previous section but this time using the DEVICE_MMIO_NAMED_ROM_INIT_BY_NAME macro instead. So the only difference would be in the driver config struct:

```
const static struct my_driver_config my_driver_config_0 = {
   ...
   DEVICE_MMIO_NAMED_ROM_INIT_BY_NAME(corge, DT_DRV_INST(...)),
   DEVICE_MMIO_NAMED_ROM_INIT_BY_NAME(grault, DT_DRV_INST(...)),
   ...
}
```

Drivers that do not use Zephyr Device Model

Some drivers or driver-like code may not user Zephyr's device model, and alternative storage must be arranged for the MMIO data. An example of this are timer drivers, or interrupt controller code.

This can be managed with the DEVICE_MMIO_TOPLEVEL set of macros, for example:

```
DEVICE_MMIO_TOPLEVEL_STATIC(my_regs, DT_DRV_INST(..));

void some_init_code(...)
{
   ...
   DEVICE_MMIO_TOPLEVEL_MAP(my_regs, K_MEM_CACHE_NONE);
   ...
}

void some_function(...)
   ...
   sys_write32(DEVICE_MMIO_TOPLEVEL_GET(my_regs), 0xDEADBEEF);
   ...
}
```

Drivers that do not use DTS

Some drivers may not obtain the MMIO physical address from DTS, such as is the case with PCI-E. In this case the device_map() function may be used directly:

```
void some_init_code(...)
{
   ...
   struct pcie_bar mbar;
   bool bar_found = pcie_get_mbar(bdf, index, &mbar);

   device_map(DEVICE_MMIO_RAM_PTR(dev), mbar.phys_addr, mbar.size, K_MEM_CACHE_NONE);
   ...
}
```

For these cases, DEVICE_MMIO_ROM directives may be omitted.

3.2.15 API Reference

group `device_model`

Device Model.

Since
1.0

Version
1.1.0

Defines

`DEVICE_HANDLE_NULL`

Flag value used to identify an unknown device.

`DEVICE_NAME_GET`(dev_id)

Expands to the name of a global device object.

Return the full name of a device object symbol created by *DEVICE_DEFINE()*, using the `dev_id` provided to *DEVICE_DEFINE()*. This is the name of the global variable storing the device structure, not a pointer to the string in the *device::name* field.

It is meant to be used for declaring extern symbols pointing to device objects before using the DEVICE_GET macro to get the device object.

This macro is normally only useful within device driver source code. In other situations, you are probably looking for *device_get_binding()*.

Parameters

- `dev_id` – Device identifier.

Returns
The full name of the device object defined by device definition macros.

`DEVICE_DEFINE`(dev_id, name, init_fn, pm, data, config, level, prio, api)

Create a device object and set it up for boot time initialization.

This macro defines a *device* that is automatically configured by the kernel during system initialization. This macro should only be used when the device is not being allocated from a devicetree node. If you are allocating a device from a devicetree node, use *DEVICE_DT_DEFINE()* or *DEVICE_DT_INST_DEFINE()* instead.

Parameters

- `dev_id` – A unique token which is used in the name of the global device structure as a C identifier.
- `name` – A string name for the device, which will be stored in *device::name*. This name can be used to look up the device with *device_get_binding()*. This must be less than Z_DEVICE_MAX_NAME_LEN characters (including terminating `NULL`) in order to be looked up from user mode.
- `init_fn` – Pointer to the device's initialization function, which will be run by the kernel during system initialization. Can be `NULL`.

- **pm** – Pointer to the device's power management resources, a *pm_device*, which will be stored in device::pm field. Use NULL if the device does not use PM.
- **data** – Pointer to the device's private mutable data, which will be stored in *device::data*.
- **config** – Pointer to the device's private constant data, which will be stored in *device::config*.
- **level** – The device's initialization level (PRE_KERNEL_1, PRE_KERNEL_2 or POST_KERNEL).
- **prio** – The device's priority within its initialization level. See SYS_INIT() for details.
- **api** – Pointer to the device's API structure. Can be NULL.

DEVICE_DT_NAME(node_id)

Return a string name for a devicetree node.

This macro returns a string literal usable as a device's name from a devicetree node identifier.

Parameters

- **node_id** – The devicetree node identifier.

Returns

The value of the node's label property, if it has one. Otherwise, the node's full name in node-name@unit-address form.

DEVICE_DT_DEFER(node_id)

Determine if a devicetree node initialization should be deferred.

Parameters

- **node_id** – The devicetree node identifier.

Returns

Boolean stating if node initialization should be deferred.

DEVICE_DT_DEFINE(node_id, init_fn, pm, data, config, level, prio, api, ...)

Create a device object from a devicetree node identifier and set it up for boot time initialization.

This macro defines a *device* that is automatically configured by the kernel during system initialization. The global device object's name as a C identifier is derived from the node's dependency ordinal. *device::name* is set to *DEVICE_DT_NAME(node_id)*.

The device is declared with extern visibility, so a pointer to a global device object can be obtained with *DEVICE_DT_GET(node_id)* from any source file that includes <zephyr/device.h>. Before using the pointer, the referenced object should be checked using *device_is_ready()*.

Parameters

- **node_id** – The devicetree node identifier.
- **init_fn** – Pointer to the device's initialization function, which will be run by the kernel during system initialization. Can be NULL.
- **pm** – Pointer to the device's power management resources, a *pm_device*, which will be stored in device::pm. Use NULL if the device does not use PM.
- **data** – Pointer to the device's private mutable data, which will be stored in *device::data*.

- **config** – Pointer to the device's private constant data, which will be stored in *device::config* field.
- **level** – The device's initialization level (PRE_KERNEL_1, PRE_KERNEL_2 or POST_KERNEL).
- **prio** – The device's priority within its initialization level. See SYS_INIT() for details.
- **api** – Pointer to the device's API structure. Can be `NULL`.

`DEVICE_DT_INST_DEFINE`(inst, ...)

Like *DEVICE_DT_DEFINE()*, but uses an instance of a `DT_DRV_COMPAT` compatible instead of a node identifier.

Parameters

- **inst** – Instance number. The `node_id` argument to *DEVICE_DT_DEFINE()* is set to *DT_DRV_INST(inst)*.
- **...** – Other parameters as expected by *DEVICE_DT_DEFINE()*.

`DEVICE_DT_NAME_GET`(node_id)

The name of the global device object for `node_id`.

Returns the name of the global device structure as a C identifier. The device must be allocated using *DEVICE_DT_DEFINE()* or *DEVICE_DT_INST_DEFINE()* for this to work.

This macro is normally only useful within device driver source code. In other situations, you are probably looking for *DEVICE_DT_GET()*.

Parameters

- **node_id** – Devicetree node identifier

Returns
The name of the device object as a C identifier

`DEVICE_DT_GET`(node_id)

Get a *device* reference from a devicetree node identifier.

Returns a pointer to a device object created from a devicetree node, if any device was allocated by a driver.

If no such device was allocated, this will fail at linker time. If you get an error that looks like `undefined reference to __device_dts_ord_<N>`, that is what happened. Check to make sure your device driver is being compiled, usually by enabling the Kconfig options it requires.

Parameters

- **node_id** – A devicetree node identifier

Returns
A pointer to the device object created for that node

`DEVICE_DT_INST_GET`(inst)

Get a *device* reference for an instance of a `DT_DRV_COMPAT` compatible.

This is equivalent to *DEVICE_DT_GET(DT_DRV_INST(inst))*.

Parameters

- **inst** – `DT_DRV_COMPAT` instance number

Returns
A pointer to the device object created for that instance

DEVICE_DT_GET_ANY(compat)

Get a *device* reference from a devicetree compatible.

If an enabled devicetree node has the given compatible and a device object was created from it, this returns a pointer to that device.

If there no such devices, this returns NULL.

If there are multiple, this returns an arbitrary one.

If this returns non-NULL, the device must be checked for readiness before use, e.g. with *device_is_ready()*.

Parameters

- **compat** – lowercase-and-underscores devicetree compatible

Returns
a pointer to a device, or NULL

DEVICE_DT_GET_ONE(compat)

Get a *device* reference from a devicetree compatible.

If an enabled devicetree node has the given compatible and a device object was created from it, this returns a pointer to that device.

If there are no such devices, this will fail at compile time.

If there are multiple, this returns an arbitrary one.

If this returns non-NULL, the device must be checked for readiness before use, e.g. with *device_is_ready()*.

Parameters

- **compat** – lowercase-and-underscores devicetree compatible

Returns
a pointer to a device

DEVICE_DT_GET_OR_NULL(node_id)

Utility macro to obtain an optional reference to a device.

If the node identifier refers to a node with status `okay`, this returns *DEVICE_DT_GET(node_id)*. Otherwise, it returns `NULL`.

Parameters

- **node_id** – devicetree node identifier

Returns
a *device* reference for the node identifier, which may be `NULL`.

DEVICE_GET(dev_id)

Obtain a pointer to a device object by name.

Return the address of a device object created by *DEVICE_DEFINE()*, using the dev_id provided to *DEVICE_DEFINE()*.

Parameters

- **dev_id** – Device identifier.

Returns
A pointer to the device object created by *DEVICE_DEFINE()*

DEVICE_DECLARE(dev_id)

Declare a static device object.

This macro can be used at the top-level to declare a device, such that *DEVICE_GET()* may be used before the full declaration in *DEVICE_DEFINE()*.

This is often useful when configuring interrupts statically in a device's init or per-instance config function, as the init function itself is required by *DEVICE_DEFINE()* and use of *DEVICE_GET()* inside it creates a circular dependency.

Parameters

- dev_id – Device identifier.

DEVICE_INIT_DT_GET(node_id)

Get a init_entry reference from a devicetree node.

Parameters

- node_id – A devicetree node identifier

Returns

A pointer to the init_entry object created for that node

DEVICE_INIT_GET(dev_id)

Get a init_entry reference from a device identifier.

Parameters

- dev_id – Device identifier.

Returns

A pointer to the init_entry object created for that device

Typedefs

typedef int16_t device_handle_t

Type used to represent a "handle" for a device.

Every *device* has an associated handle. You can get a pointer to a *device* from its handle and vice versa, but the handle uses less space than a pointer. The device.h API mainly uses handles to store lists of multiple devices in a compact way.

The extreme values and zero have special significance. Negative values identify functionality that does not correspond to a Zephyr device, such as the system clock or a SYS_INIT() function.

See also

device_handle_get()

See also

device_from_handle()

typedef int (*`device_visitor_callback_t`)(const struct *device* *dev, void *context)

Prototype for functions used when iterating over a set of devices.

Such a function may be used in API that identifies a set of devices and provides a visitor API supporting caller-specific interaction with each device in the set.

The visit is said to succeed if the visitor returns a non-negative value.

See also

device_required_foreach()

See also

device_supported_foreach()

Param dev
a device in the set being iterated

Param context
state used to support the visitor function

Return
A non-negative number to allow walking to continue, and a negative error code to case the iteration to stop.

Functions

static inline *device_handle_t* `device_handle_get`(const struct *device* *dev)

Get the handle for a given device.

Parameters

- `dev` – the device for which a handle is desired.

Returns
the handle for the device, or DEVICE_HANDLE_NULL if the device does not have an associated handle.

static inline const struct *device* *`device_from_handle`(*device_handle_t* dev_handle)

Get the device corresponding to a handle.

Parameters

- `dev_handle` – the device handle

Returns
the device that has that handle, or a null pointer if `dev_handle` does not identify a device.

static inline const *device_handle_t* *`device_required_handles_get`(const struct *device* *dev, size_t *count)

Get the device handles for devicetree dependencies of this device.

This function returns a pointer to an array of device handles. The length of the array is stored in the `count` parameter.

The array contains a handle for each device that dev requires directly, as determined from the devicetree. This does not include transitive dependencies; you must recursively determine those.

Parameters

- **dev** – the device for which dependencies are desired.
- **count** – pointer to where this function should store the length of the returned array. No value is stored if the call returns a null pointer. The value may be set to zero if the device has no devicetree dependencies.

Returns

a pointer to a sequence of count device handles, or a null pointer if dev does not have any dependency data.

static inline const *device_handle_t* ***device_injected_handles_get**(const struct *device* *dev, size_t *count)

Get the device handles for injected dependencies of this device.

This function returns a pointer to an array of device handles. The length of the array is stored in the count parameter.

The array contains a handle for each device that dev manually injected as a dependency, via providing extra arguments to Z_DEVICE_DEFINE. This does not include transitive dependencies; you must recursively determine those.

Parameters

- **dev** – the device for which injected dependencies are desired.
- **count** – pointer to where this function should store the length of the returned array. No value is stored if the call returns a null pointer. The value may be set to zero if the device has no devicetree dependencies.

Returns

a pointer to a sequence of *count device handles, or a null pointer if dev does not have any dependency data.

static inline const *device_handle_t* ***device_supported_handles_get**(const struct *device* *dev, size_t *count)

Get the set of handles that this device supports.

This function returns a pointer to an array of device handles. The length of the array is stored in the count parameter.

The array contains a handle for each device that dev “supports” — that is, devices that require dev directly — as determined from the devicetree. This does not include transitive dependencies; you must recursively determine those.

Parameters

- **dev** – the device for which supports are desired.
- **count** – pointer to where this function should store the length of the returned array. No value is stored if the call returns a null pointer. The value may be set to zero if nothing in the devicetree depends on dev.

Returns

a pointer to a sequence of *count device handles, or a null pointer if dev does not have any dependency data.

int **device_required_foreach**(const struct *device* *dev, *device_visitor_callback_t* visitor_cb, void *context)

Visit every device that dev directly requires.

Zephyr maintains information about which devices are directly required by another device; for example an I2C-based sensor driver will require an I2C controller for communication. Required devices can derive from statically-defined devicetree relationships or dependencies registered at runtime.

This API supports operating on the set of required devices. Example uses include making sure required devices are ready before the requiring device is used, and releasing them when the requiring device is no longer needed.

There is no guarantee on the order in which required devices are visited.

If the `visitor_cb` function returns a negative value iteration is halted, and the returned value from the visitor is returned from this function.

Note

This API is not available to unprivileged threads.

Parameters

- **dev** – a device of interest. The devices that this device depends on will be used as the set of devices to visit. This parameter must not be null.
- **visitor_cb** – the function that should be invoked on each device in the dependency set. This parameter must not be null.
- **context** – state that is passed through to the visitor function. This parameter may be null if `visitor_cb` tolerates a null `context`.

Returns

The number of devices that were visited if all visits succeed, or the negative value returned from the first visit that did not succeed.

int **device_supported_foreach**(const struct *device* *dev, *device_visitor_callback_t* visitor_cb, void *context)

Visit every device that `dev` directly supports.

Zephyr maintains information about which devices are directly supported by another device; for example an I2C controller will support an I2C-based sensor driver. Supported devices can derive from statically-defined devicetree relationships.

This API supports operating on the set of supported devices. Example uses include iterating over the devices connected to a regulator when it is powered on.

There is no guarantee on the order in which required devices are visited.

If the `visitor_cb` function returns a negative value iteration is halted, and the returned value from the visitor is returned from this function.

Note

This API is not available to unprivileged threads.

Parameters

- **dev** – a device of interest. The devices that this device supports will be used as the set of devices to visit. This parameter must not be null.
- **visitor_cb** – the function that should be invoked on each device in the support set. This parameter must not be null.

- **context** – state that is passed through to the visitor function. This parameter may be null if `visitor_cb` tolerates a null `context`.

Returns
The number of devices that were visited if all visits succeed, or the negative value returned from the first visit that did not succeed.

const struct *device* ***device_get_binding**(const char *name)

Get a *device* reference from its *device::name* field.

This function iterates through the devices on the system. If a device with the given `name` field is found, and that device initialized successfully at boot time, this function returns a pointer to the device.

If no device has the given `name`, this function returns `NULL`.

This function also returns NULL when a device is found, but it failed to initialize successfully at boot time. (To troubleshoot this case, set a breakpoint on your device driver's initialization function.)

Parameters

- **name** – device name to search for. A null pointer, or a pointer to an empty string, will cause NULL to be returned.

Returns
pointer to device structure with the given name; `NULL` if the device is not found or if the device with that name's initialization function failed.

bool **device_is_ready**(const struct *device* *dev)

Verify that a device is ready for use.

Indicates whether the provided device pointer is for a device known to be in a state where it can be used with its standard API.

This can be used with device pointers captured from *DEVICE_DT_GET()*, which does not include the readiness checks of *device_get_binding()*. At minimum this means that the device has been successfully initialized.

Parameters

- **dev** – pointer to the device in question.

Return values

- **true** – If the device is ready for use.
- **false** – If the device is not ready for use or if a NULL device pointer is passed as argument.

int **device_init**(const struct *device* *dev)

Initialize a device.

A device whose initialization was deferred (by marking it as `zephyr,deferred-init` on devicetree) needs to be initialized manually via this call. Note that only devices whose initialization was deferred can be initialized via this call - one can not try to initialize a non initialization deferred device that failed initialization with this call.

Parameters

- **dev** – device to be initialized.

Return values

- **-ENOENT** – If device was not found - or isn't a deferred one.
- **-errno** – For other errors.

struct **device_state**

#include <device.h> Runtime device dynamic structure (in RAM) per driver instance.

Fields in this are expected to be default-initialized to zero. The kernel driver infrastructure and driver access functions are responsible for ensuring that any non-zero initialization is done before they are accessed.

Public Members

uint8_t **init_res**

Device initialization return code (positive errno value).

Device initialization functions return a negative errno code if they fail. In Zephyr, errno values do not exceed 255, so we can store the positive result value in a uint8_t type.

bool **initialized**

Indicates the device initialization function has been invoked.

struct **device**

#include <device.h> Runtime device structure (in ROM) per driver instance.

Public Members

const char ***name**

Name of the device instance.

const void ***config**

Address of device instance config information.

const void ***api**

Address of the API structure exposed by the device instance.

struct *device_state* ***state**

Address of the common device state.

void ***data**

Address of the device instance private data.

const *device_handle_t* ***deps**

Optional pointer to dependencies associated with the device.

This encodes a sequence of sets of device handles that have some relationship to this node. The individual sets are extracted with dedicated API, such as *device_required_handles_get()*. Only available if `CONFIG_DEVICE_DEPS` is enabled.

union device

Reference to the device PM resources (only available if `CONFIG_PM_DEVICE` is enabled).

3.3 User Mode

Zephyr offers the capability to run threads at a reduced privilege level which we call user mode. The current implementation is designed for devices with MPU hardware.

For details on creating threads that run in user mode, please see *Lifecycle*.

3.3.1 Overview

Threat Model

User mode threads are considered to be untrusted by Zephyr and are therefore isolated from other user mode threads and from the kernel. A flawed or malicious user mode thread cannot leak or modify the private data/resources of another thread or the kernel, and cannot interfere with or control another user mode thread or the kernel.

Example use-cases of Zephyr's user mode features:

- The kernel can protect against many unintentional programming errors which could otherwise silently or spectacularly corrupt the system.
- The kernel can sandbox complex data parsers such as interpreters, network protocols, and filesystems such that malicious third-party code or data cannot compromise the kernel or other threads.
- The kernel can support the notion of multiple logical "applications", each with their own group of threads and private data structures, which are isolated from each other if one crashes or is otherwise compromised.

Design Goals For threads running in a non-privileged CPU state (hereafter referred to as 'user mode') we aim to protect against the following:

- We prevent access to memory not specifically granted, or incorrect access to memory that has an incompatible policy, such as attempting to write to a read-only area.
  - Access to thread stack buffers will be controlled with a policy which partially depends on the underlying memory protection hardware.
    - A user thread will by default have read/write access to its own stack buffer.
    - A user thread will never by default have access to user thread stacks that are not members of the same memory domain.
    - A user thread will never by default have access to thread stacks owned by a supervisor thread, or thread stacks used to handle system call privilege elevations, interrupts, or CPU exceptions.
    - A user thread may have read/write access to the stacks of other user threads in the same memory domain, depending on hardware.
      - On MPU systems, threads may only access their own stack buffer.
      - On MMU systems, threads may access any user thread stack in the same memory domain. Portable code should not assume this.
  - By default, program text and read-only data are accessible to all threads on read-only basis, kernel-wide. This policy may be adjusted.
  - User threads by default are not granted default access to any memory except what is noted above.
- We prevent use of device drivers or kernel objects not specifically granted, with the permission granularity on a per object or per driver instance basis.

- We validate kernel or driver API calls with incorrect parameters that would otherwise cause a crash or corruption of data structures private to the kernel. This includes:
  - Using the wrong kernel object type.
  - Using parameters outside of proper bounds or with nonsensical values.
  - Passing memory buffers that the calling thread does not have sufficient access to read or write, depending on the semantics of the API.
  - Use of kernel objects that are not in a proper initialization state.
- We ensure the detection and safe handling of user mode stack overflows.
- We prevent invoking system calls to functions excluded by the kernel configuration.
- We prevent disabling of or tampering with kernel-defined and hardware- enforced memory protections.
- We prevent re-entry from user to supervisor mode except through the kernel- defined system calls and interrupt handlers.
- We prevent the introduction of new executable code by user mode threads, except to the extent to which this is supported by kernel system calls.

We are specifically not protecting against the following attacks:

- The kernel itself, and any threads that are executing in supervisor mode, are assumed to be trusted.
- The toolchain and any supplemental programs used by the build system are assumed to be trusted.
- The kernel build is assumed to be trusted. There is considerable build-time logic for creating the tables of valid kernel objects, defining system calls, and configuring interrupts. The .elf binary files that are worked with during this process are all assumed to be trusted code.
- We can't protect against mistakes made in memory domain configuration done in kernel mode that exposes private kernel data structures to a user thread. RAM for kernel objects should always be configured as supervisor-only.
- It is possible to make top-level declarations of user mode threads and assign them permissions to kernel objects. In general, all C and header files that are part of the kernel build producing zephyr.elf are assumed to be trusted.
- We do not protect against denial of service attacks through thread CPU starvation. Zephyr has no thread priority aging and a user thread of a particular priority can starve all threads of lower priority, and also other threads of the same priority if time-slicing is not enabled.
- There are build-time defined limits on how many threads can be active simultaneously, after which creation of new user threads will fail.
- Stack overflows for threads running in supervisor mode may be caught, but the integrity of the system cannot be guaranteed.

High-level Policy Details

Broadly speaking, we accomplish these thread-level memory protection goals through the following mechanisms:

- Any user thread will only have access to a subset of memory: typically its stack, program text, read-only data, and any partitions configured in the *Memory Protection Design* it belongs to. Access to any other RAM must be done on the thread's behalf through system calls, or specifically granted by a supervisor thread using the memory domain APIs. Newly created threads inherit the memory domain configuration of the parent. Threads may communicate with each other by having shared membership of the same memory domains, or via kernel objects such as semaphores and pipes.

- User threads cannot directly access memory belonging to kernel objects. Although pointers to kernel objects are used to reference them, actual manipulation of kernel objects is done through system call interfaces. Device drivers and threads stacks are also considered kernel objects. This ensures that any data inside a kernel object that is private to the kernel cannot be tampered with.
- User threads by default have no permission to access any kernel object or driver other than their own thread object. Such access must be granted by another thread that is either in supervisor mode or has permission on both the receiving thread object and the kernel object being granted access to. The creation of new threads has an option to automatically inherit permissions of all kernel objects granted to the parent, except the parent thread itself.
- For performance and footprint reasons Zephyr normally does little or no parameter error checking for kernel object or device driver APIs. Access from user mode through system calls involves an extra layer of handler functions, which are expected to rigorously validate access permissions and type of the object, check the validity of other parameters through bounds checking or other means, and verify proper read/write access to any memory buffers involved.
- Thread stacks are defined in such a way that exceeding the specified stack space will generate a hardware fault. The way this is done specifically varies per architecture.

Constraints

All kernel objects, thread stacks, and device driver instances must be defined at build time if they are to be used from user mode. Dynamic use-cases for kernel objects will need to go through predefined pools of available objects.

There are some constraints if additional application binary data is loaded for execution after the kernel starts:

- Loaded object code will not be able to define any kernel objects that will be recognized by the kernel. This code will instead need to use APIs for requesting kernel objects from pools.
- Similarly, since the loaded object code will not be part of the kernel build process, this code will not be able to install interrupt handlers, instantiate device drivers, or define system calls, regardless of what mode it runs in.
- Loaded object code that does not come from a verified source should always be entered with the CPU already in user mode.

3.3.2 Memory Protection Design

Zephyr's memory protection design is geared towards microcontrollers with MPU (Memory Protection Unit) hardware. We do support some architectures, such as x86, which have a paged MMU (Memory Management Unit), but in that case the MMU is used like an MPU with an identity page table.

All of the discussion below will be using MPU terminology; systems with MMUs can be considered to have an MPU with an unlimited number of programmable regions.

There are a few different levels on how memory access is configured when Zephyr memory protection features are enabled, which we will describe here:

Boot Time Memory Configuration

This is the configuration of the MPU after the kernel has started up. It should contain the following:

- Any configuration of memory regions which need to have special caching or write-back policies for basic hardware and driver function. Note that most MPUs have the concept of a default memory access policy map, which can be enabled as a "background" mapping for any area of memory that doesn't have an MPU region configuring it. It is strongly recommended to use this to maximize the number of available MPU regions for the end user. On ARMv7-M/ARMv8-M this is called the System Address Map, other CPUs may have similar capabilities.
- A read-only, executable region or regions for program text and ro-data, that is accessible to user mode. This could be further sub-divided into a read-only region for ro-data, and a read-only, executable region for text, but this will require an additional MPU region. This is required so that threads running in user mode can read ro-data and fetch instructions.
- Depending on configuration, user-accessible read-write regions to support extra features like GCOV, HEP, etc.

Assuming there is a background map which allows supervisor mode to access any memory it needs, and regions are defined which grant user mode access to text/ro-data, this is sufficient for the boot time configuration.

Hardware Stack Overflow

`CONFIG_HW_STACK_PROTECTION` is an optional feature which detects stack buffer overflows when the system is running in supervisor mode. This catches issues when the entire stack buffer has overflowed, and not individual stack frames, use compiler-assisted `CONFIG_STACK_CANARIES` for that.

Like any crash in supervisor mode, no guarantees can be made about the overall health of the system after a supervisor mode stack overflow, and any instances of this should be treated as a serious error. However it's still very useful to know when these overflows happen, as without robust detection logic the system will either crash in mysterious ways or behave in an undefined manner when the stack buffer overflows.

Some systems implement this feature by creating at runtime a 'guard' MPU region which is set to be read-only and is at either the beginning or immediately preceding the supervisor mode stack buffer. If the stack overflows an exception will be generated.

This feature is optional and is not required to catch stack overflows in user mode; disabling this may free 1-2 MPU regions depending on the MPU design.

Other systems may have dedicated CPU support for catching stack overflows and no extra MPU regions will be required.

Thread Stack

Any thread running in user mode will need access to its own stack buffer. On context switch into a user mode thread, a dedicated MPU region or MMU page table entries will be programmed with the bounds of the stack buffer. A thread exceeding its stack buffer will start pushing data onto memory it doesn't have access to and a memory access violation exception will be generated.

Note that user threads have access to the stacks of other user threads in the same memory domain. This is the minimum required for architectures to support memory domains. Architecture can further restrict access to stacks so each user thread only has access to its own stack if such architecture advertises this capability via `CONFIG_ARCH_MEM_DOMAIN_SUPPORTS_ISOLATED_STACKS`. This behavior is enabled by default if supported and can be selectively disabled via `CONFIG_MEM_DOMAIN_ISOLATED_STACKS` if architecture supports both operating modes. However, some architectures may decide to enable this all the time, and thus this option cannot be disabled. Regardless of these kconfigs, user threads cannot access the stacks of other user threads outside of their memory domains.

Thread Resource Pools

A small subset of kernel APIs, invoked as system calls, require heap memory allocations. This memory is used only by the kernel and is not accessible directly by user mode. In order to use these system calls, invoking threads must assign themselves to a resource pool, which is a *k_heap* object. Memory is drawn from a thread's resource pool using `z_thread_malloc()` and freed with *k_free()*.

The APIs which use resource pools are as follows, with any alternatives noted for users who do not want heap allocations within their application:

- *k_stack_alloc_init()* sets up a k_stack with its storage buffer allocated out of a resource pool instead of a buffer provided by the user. An alternative is to declare k_stacks that are automatically initialized at boot with *K_STACK_DEFINE()*, or to initialize the k_stack in supervisor mode with *k_stack_init()*.
- *k_pipe_alloc_init()* sets up a k_pipe object with its storage buffer allocated out of a resource pool instead of a buffer provided by the user. An alternative is to declare k_pipes that are automatically initialized at boot with *K_PIPE_DEFINE()*, or to initialize the k_pipe in supervisor mode with *k_pipe_init()*.
- *k_msgq_alloc_init()* sets up a k_msgq object with its storage buffer allocated out of a resource pool instead of a buffer provided by the user. An alternative is to declare a k_msgq that is automatically initialized at boot with *K_MSGQ_DEFINE()*, or to initialize the k_msgq in supervisor mode with *k_msgq_init()*.
- *k_poll()* when invoked from user mode, needs to make a kernel-side copy of the provided events array while waiting for an event. This copy is freed when *k_poll()* returns for any reason.
- *k_queue_alloc_prepend()* and *k_queue_alloc_append()* allocate a container structure to place the data in, since the internal bookkeeping information that defines the queue cannot be placed in the memory provided by the user.
- *k_object_alloc()* allows for entire kernel objects to be dynamically allocated at runtime and a usable pointer to them returned to the caller.

The relevant API is *k_thread_heap_assign()* which assigns a k_heap to draw these allocations from for the target thread.

If the system heap is enabled, then the system heap may be used with `k_thread_system_pool_assign()`, but it is preferable for different logical applications running on the system to have their own pools.

Memory Domains

The kernel ensures that any user thread will have access to its own stack buffer, plus program text and read-only data. The memory domain APIs are the way to grant access to additional blocks of memory to a user thread.

Conceptually, a memory domain is a collection of some number of memory partitions. The maximum number of memory partitions in a domain is limited by the number of available MPU regions. This is why it is important to minimize the number of boot-time MPU regions.

Memory domains are *not* intended to control access to memory from supervisor mode. In some cases this may be unavoidable; for example some architectures do not allow for the definition of regions which are read-only to user mode but read-write to supervisor mode. A great deal of care must be taken when working with such regions to not unintentionally cause the kernel to crash when accessing such a region. Any attempt to use memory domain APIs to control supervisor mode access is at best undefined behavior; supervisor mode access policy is only intended to be controlled by boot-time memory regions.

Memory domain APIs are only available to supervisor mode. The only control user mode has over memory domains is that any user thread's child threads will automatically become members of the parent's domain.

All threads are members of a memory domain, including supervisor threads (even though this has no implications on their memory access). There is a default domain `k_mem_domain_default` which will be assigned to threads if they have not been specifically assigned to a domain, or inherited a memory domain membership from their parent thread. The main thread starts as a member of the default domain.

Memory Partitions Each memory partition consists of a memory address, a size, and access attributes. It is intended that memory partitions are used to control access to system memory. Defining memory partitions are subject to the following constraints:

- The partition must represent a memory region that can be programmed by the underlying memory management hardware, and needs to conform to any underlying hardware constraints. For example, many MPU-based systems require that partitions be sized to some power of two, and aligned to their own size. For MMU-based systems, the partition must be aligned to a page and the size some multiple of the page size.
- Partitions within the same memory domain may not overlap each other. There is no notion of precedence among partitions within a memory domain. Partitions within a memory domain are assumed to have a higher precedence than any boot-time memory regions, however whether a memory domain partition can overlap a boot-time memory region is architecture specific.
- The same partition may be specified in multiple memory domains. For example there may be a shared memory area that multiple domains grant access to.
- Care must be taken in determining what memory to expose in a partition. It is not appropriate to provide direct user mode access to any memory containing private kernel data.
- Memory domain partitions are intended to control access to system RAM. Configuration of memory partitions which do not correspond to RAM may not be supported by the architecture; this is true for MMU-based systems.

There are two ways to define memory partitions: either manually or automatically.

Manual Memory Partitions The following code declares a global array `buf`, and then declares a read-write partition for it which may be added to a domain:

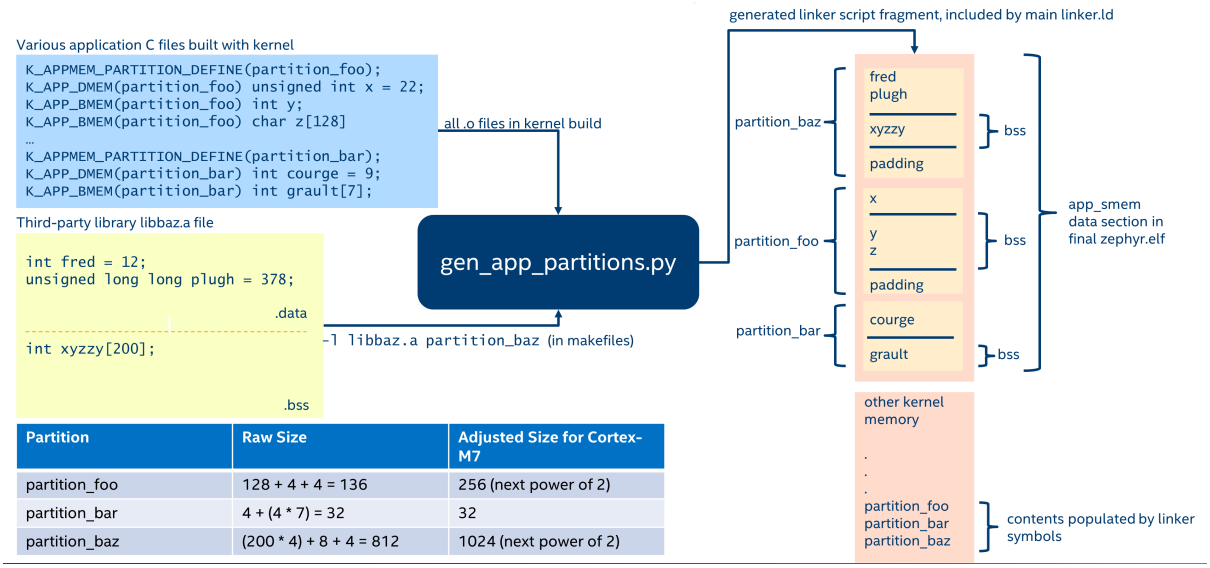
```
uint8_t __aligned(32) buf[32];

K_MEM_PARTITION_DEFINE(my_partition, buf, sizeof(buf),
                       K_MEM_PARTITION_P_RW_U_RW);
```

This does not scale particularly well when we are trying to contain multiple objects spread out across several C files into a single partition.

Automatic Memory Partitions Automatic memory partitions are created by the build system. All globals which need to be placed inside a partition are tagged with their destination partition. The build system will then coalesce all of these into a single contiguous block of memory, zero any BSS variables at boot, and define a memory partition of appropriate base address and size which contains all the tagged data.

Automatic memory partitions are only configured as read-write regions. They are defined with `K_APPMEM_PARTITION_DEFINE()`. Global variables are then routed to this partition using `K_APP_DMEM()` for initialized data and `K_APP_BMEM()` for BSS.



Partition	Raw Size	Adjusted Size for Cortex-M7
partition_foo	128 + 4 + 4 = 136	256 (next power of 2)
partition_bar	4 + (4 * 7) = 32	32
partition_baz	(200 * 4) + 8 + 4 = 812	1024 (next power of 2)

Fig. 2: Automatic Memory Domain build flow

```
#include <zephyr/app_memory/app_memdomain.h>

/* Declare a k_mem_partition "my_partition" that is read-write to
 * user mode. Note that we do not specify a base address or size.
 */
K_APPMEM_PARTITION_DEFINE(my_partition);

/* The global variable var1 will be inside the bounds of my_partition
 * and be initialized with 37 at boot.
 */
K_APP_DMEM(my_partition) int var1 = 37;

/* The global variable var2 will be inside the bounds of my_partition
 * and be zeroed at boot size K_APP_BMEM() was used, indicating a BSS
 * variable.
 */
K_APP_BMEM(my_partition) int var2;
```

The build system will ensure that the base address of `my_partition` will be properly aligned, and the total size of the region conforms to the memory management hardware requirements, adding padding if necessary.

If multiple partitions are being created, a variadic preprocessor macro can be used as provided in `app_macro_support.h`:

```
FOR_EACH(K_APPMEM_PARTITION_DEFINE, part0, part1, part2);
```

Automatic Partitions for Static Library Globals The build-time logic for setting up automatic memory partitions is in `scripts/build/gen_app_partitions.py`. If a static library is linked into Zephyr, it is possible to route all the globals in that library to a specific memory partition with the `--library` argument.

For example, if the Newlib C library is enabled, the Newlib globals all need to be placed in `z_libc_partition`. The invocation of the script in the top-level `CMakeLists.txt` adds the following:

```
gen_app_partitions.py ... --library libc.a z_libc_partition ..
```

For pre-compiled libraries there is no support for expressing this in the project-level configuration or build files; the toplevel `CMakeLists.txt` must be edited.

For Zephyr libraries created using `zephyr_library` or `zephyr_library_named` the `zephyr_library_app_memory` function can be used to specify the memory partition where all globals in the library should be placed.

Pre-defined Memory Partitions There are a few memory partitions which are pre-defined by the system:

- `z_malloc_partition` - This partition contains the system-wide pool of memory used by libc malloc(). Due to possible starvation issues, it is not recommended to draw heap memory from a global pool, instead it is better to define various sys_heap objects and assign them to specific memory domains.
- `z_libc_partition` - Contains globals required by the C library and runtime. Required when using either the Minimal C library or the Newlib C Library. Required when `CONFIG_STACK_CANARIES` is enabled.

Library-specific partitions are listed in `include/app_memory/partitions.h`. For example, to use the MBEDTLS library from user mode, the `k_mbedtls_partition` must be added to the domain.

Memory Domain Usage

Create a Memory Domain A memory domain is defined using a variable of type *k_mem_domain*. It must then be initialized by calling *k_mem_domain_init()*.

The following code defines and initializes an empty memory domain.

```
struct k_mem_domain app0_domain;

k_mem_domain_init(&app0_domain, 0, NULL);
```

Add Memory Partitions into a Memory Domain There are two ways to add memory partitions into a memory domain.

This first code sample shows how to add memory partitions while creating a memory domain.

```
/* the start address of the MPU region needs to align with its size */
uint8_t __aligned(32) app0_buf[32];
uint8_t __aligned(32) app1_buf[32];

K_MEM_PARTITION_DEFINE(app0_part0, app0_buf, sizeof(app0_buf),
                       K_MEM_PARTITION_P_RW_U_RW);

K_MEM_PARTITION_DEFINE(app0_part1, app1_buf, sizeof(app1_buf),
                       K_MEM_PARTITION_P_RW_U_RO);

struct k_mem_partition *app0_parts[] = {
    app0_part0,
    app0_part1
};

k_mem_domain_init(&app0_domain, ARRAY_SIZE(app0_parts), app0_parts);
```

This second code sample shows how to add memory partitions into an initialized memory domain one by one.

```
/* the start address of the MPU region needs to align with its size */
uint8_t __aligned(32) app0_buf[32];
uint8_t __aligned(32) app1_buf[32];

K_MEM_PARTITION_DEFINE(app0_part0, app0_buf, sizeof(app0_buf),
                       K_MEM_PARTITION_P_RW_U_RW);

K_MEM_PARTITION_DEFINE(app0_part1, app1_buf, sizeof(app1_buf),
                       K_MEM_PARTITION_P_RW_U_RO);

k_mem_domain_add_partition(&app0_domain, &app0_part0);
k_mem_domain_add_partition(&app0_domain, &app0_part1);
```

Note

The maximum number of memory partitions is limited by the maximum number of MPU regions or the maximum number of MMU tables.

Memory Domain Assignment Any thread may join a memory domain, and any memory domain may have multiple threads assigned to it. Threads are assigned to memory domains with an API call:

```
k_mem_domain_add_thread(&app0_domain, app_thread_id);
```

If the thread was already a member of some other domain (including the default domain), it will be removed from it in favor of the new one.

In addition, if a thread is a member of a memory domain, and it creates a child thread, that thread will belong to the domain as well.

Remove a Memory Partition from a Memory Domain The following code shows how to remove a memory partition from a memory domain.

```
k_mem_domain_remove_partition(&app0_domain, &app0_part1);
```

The k_mem_domain_remove_partition() API finds the memory partition that matches the given parameter and removes that partition from the memory domain.

Available Partition Attributes When defining a partition, we need to set access permission attributes to the partition. Since the access control of memory partitions relies on either an MPU or MMU, the available partition attributes would be architecture dependent.

The complete list of available partition attributes for a specific architecture is found in the architecture-specific include file `include/zephyr/arch/<arch name>/arch.h`, (for example, `include/zehpyr/arch/arm/arch.h`.) Some examples of partition attributes are:

```
/* Denote partition is privileged read/write, unprivileged read/write */
K_MEM_PARTITION_P_RW_U_RW
/* Denote partition is privileged read/write, unprivileged read-only */
K_MEM_PARTITION_P_RW_U_RO
```

In almost all cases `K_MEM_PARTITION_P_RW_U_RW` is the right choice.

Configuration Options

Related configuration options:

- CONFIG_MAX_DOMAIN_PARTITIONS

API Reference

The following memory domain APIs are provided by include/zephyr/kernel.h:

group mem_domain_apis

Defines

K_MEM_PARTITION_DEFINE(name, start, size, attr)

Statically declare a memory partition.

Functions

int k_mem_domain_init(struct *k_mem_domain* *domain, uint8_t num_parts, struct *k_mem_partition* *parts[])

Initialize a memory domain.

Initialize a memory domain with given name and memory partitions.

See documentation for *k_mem_domain_add_partition()* for details about partition constraints.

Do not call *k_mem_domain_init()* on the same memory domain more than once, doing so is undefined behavior.

Parameters

- **domain** – The memory domain to be initialized.
- **num_parts** – The number of array items of "parts" parameter.
- **parts** – An array of pointers to the memory partitions. Can be NULL if num_parts is zero.

Return values

- **0** – if successful
- **-EINVAL** – if invalid parameters supplied
- **-ENOMEM** – if insufficient memory

int k_mem_domain_add_partition(struct *k_mem_domain* *domain, struct *k_mem_partition* *part)

Add a memory partition into a memory domain.

Add a memory partition into a memory domain. Partitions must conform to the following constraints:

- Partitions in the same memory domain may not overlap each other.
- Partitions must not be defined which expose private kernel data structures or kernel objects.
- The starting address alignment, and the partition size must conform to the constraints of the underlying memory management hardware, which varies per architecture.

- Memory domain partitions are only intended to control access to memory from user mode threads.
- If CONFIG_EXECUTE_XOR_WRITE is enabled, the partition must not allow both writes and execution.

Violating these constraints may lead to CPU exceptions or undefined behavior.

Parameters

- `domain` – The memory domain to be added a memory partition.
- `part` – The memory partition to be added

Return values

- `0` – if successful
- `-EINVAL` – if invalid parameters supplied
- `-ENOSPC` – if no free partition slots available

int `k_mem_domain_remove_partition`(struct *k_mem_domain* *domain, struct *k_mem_partition* *part)

Remove a memory partition from a memory domain.

Remove a memory partition from a memory domain.

Parameters

- `domain` – The memory domain to be removed a memory partition.
- `part` – The memory partition to be removed

Return values

- `0` – if successful
- `-EINVAL` – if invalid parameters supplied
- `-ENOENT` – if no matching partition found

int `k_mem_domain_add_thread`(struct *k_mem_domain* *domain, k_tid_t thread)

Add a thread into a memory domain.

Add a thread into a memory domain. It will be removed from whatever memory domain it previously belonged to.

Parameters

- `domain` – The memory domain that the thread is going to be added into.
- `thread` – ID of thread going to be added into the memory domain.

Returns

0 if successful, fails otherwise.

Variables

struct *k_mem_domain* `k_mem_domain_default`

Default memory domain.

All threads are a member of some memory domain, even if running in supervisor mode. Threads belong to this default memory domain if they haven't been added to or inherited membership from some other domain.

This memory domain has the z_libc_partition partition for the C library added to it if exists.

struct **k_mem_partition**

#include <mem_domain.h> Memory Partition.

A memory partition is a region of memory in the linear address space with a specific access policy.

The alignment of the starting address, and the alignment of the size value may have varying requirements based on the capabilities of the underlying memory management hardware; arbitrary values are unlikely to work.

Public Members

uintptr_t **start**

start address of memory partition

size_t **size**

size of memory partition

k_mem_partition_attr_t **attr**

attribute of memory partition

struct **k_mem_domain**

#include <mem_domain.h> Memory Domain.

A memory domain is a collection of memory partitions, used to represent a user thread's access policy for the linear address space. A thread may be a member of only one memory domain, but any memory domain may have multiple threads that are members.

Supervisor threads may also be a member of a memory domain; this has no implications on their memory access but can be useful as any child threads inherit the memory domain membership of the parent.

A user thread belonging to a memory domain with no active partitions will have guaranteed access to its own stack buffer, program text, and read-only data.

Public Members

struct *k_mem_partition* **partitions**[CONFIG_MAX_DOMAIN_PARTITIONS]

partitions in the domain

sys_dlist_t **mem_domain_q**

Doubly linked list of member threads.

uint8_t **num_partitions**

number of active partitions in the domain

3.3.3 Kernel Objects

A kernel object can be one of three classes of data:

- A core kernel object, such as a semaphore, thread, pipe, etc.

- A thread stack, which is an array of `z_thread_stack_element` and declared with *K_THREAD_STACK_DEFINE()*
- A device driver instance (const struct device) that belongs to one of a defined set of subsystems

The set of known kernel objects and driver subsystems is defined in include/kernel.h as `k_objects`.

Kernel objects are completely opaque to user threads. User threads work with addresses to kernel objects when making API calls, but may never dereference these addresses, doing so will cause a memory protection fault. All kernel objects must be placed in memory that is not accessible by user threads.

Since user threads may not directly manipulate kernel objects, all use of them must go through system calls. In order to perform a system call on a kernel object, checks are performed by system call handler functions that the kernel object address is valid and that the calling thread has sufficient permissions to work with it.

Permission on an object also has the semantics of a reference to an object. This is significant for certain object APIs which do temporary allocations, or objects which themselves have been allocated from a runtime memory pool.

If an object loses all references, two events may happen:

- If the object has an associated cleanup function, the cleanup function may be called to release any runtime-allocated buffers the object was using.
- If the object itself was dynamically allocated, the memory for the object will be freed.

Object Placement

Kernel objects that are only used by supervisor threads have no restrictions and can be located anywhere in the binary, or even declared on stacks. However, to prevent accidental or intentional corruption by user threads, they must not be located in any memory that user threads have direct access to.

In order for a static kernel object to be usable by a user thread via system call APIs, several conditions must be met on how the kernel object is declared:

- The object must be declared as a top-level global at build time, such that it appears in the ELF symbol table. It is permitted to declare kernel objects with static scope. The post-build script *scripts/build/gen_kobject_list.py* scans the generated ELF file to find kernel objects and places their memory addresses in a special table of kernel object metadata. Kernel objects may be members of arrays or embedded within other data structures.
- Kernel objects must be located in memory reserved for the kernel. They must not be located in any memory partitions that are user-accessible.
- Any memory reserved for a kernel object must be used exclusively for that object. Kernel objects may not be members of a union data type.

Kernel objects that are found but do not meet the above conditions will not be included in the generated table that is used to validate kernel object pointers passed in from user mode.

The debug output of the *scripts/build/gen_kobject_list.py* script may be useful when debugging why some object was unexpectedly not being tracked. This information will be printed if the script is run with the `--verbose` flag, or if the build system is invoked with verbose output.

Dynamic Objects

Kernel objects may also be allocated at runtime if `CONFIG_DYNAMIC_OBJECTS` is enabled. In this case, the *k_object_alloc()* API may be used to instantiate an object from the calling thread's

resource pool. Such allocations may be freed in two ways:

- Supervisor threads may call *k_object_free()* to force a dynamic object to be released.
- If an object's references drop to zero (which happens when no threads have permissions on it) the object will be automatically freed. User threads may drop their own permission on an object with *k_object_release()*, and their permissions are automatically cleared when a thread terminates. Supervisor threads may additionally revoke references for another thread using *k_object_access_revoke()*.

Because permissions are also used for reference counting, it is important for supervisor threads to acquire permissions on objects they are using even though the access control aspects of the permission system are not enforced.

Implementation Details The *scripts/build/gen_kobject_list.py* script is a post-build step which finds all the valid kernel object instances in the binary. It accomplishes this by parsing the DWARF debug information present in the generated ELF file for the kernel.

Any instances of structs or arrays corresponding to kernel objects that meet the object placement criteria will have their memory addresses placed in a special perfect hash table of kernel objects generated by the 'gperf' tool. When a system call is made and the kernel is presented with a memory address of what may or may not be a valid kernel object, the address can be validated with a constant-time lookup in this table.

Drivers are a special case. All drivers are instances of *device*, but it is important to know what subsystem a driver belongs to so that incorrect operations, such as calling a UART API on a sensor driver object, can be prevented. When a device struct is found, its API pointer is examined to determine what subsystem the driver belongs to.

The table itself maps kernel object memory addresses to instances of `z_object`, which has all the metadata for that object. This includes:

- A bitfield indicating permissions on that object. All threads have a numerical ID assigned to them at build time, used to index the permission bitfield for an object to see if that thread has permission on it. The size of this bitfield is controlled by the `CONFIG_MAX_THREAD_BYTES` option and the build system will generate an error if this value is too low.
- A type field indicating what kind of object this is, which is some instance of `k_objects`.
- A set of flags for that object. This is currently used to track initialization state and whether an object is public or not.
- An extra data field. The semantics of this field vary by object type, see the definition of `z_object_data`.

Dynamic objects allocated at runtime are tracked in a runtime red/black tree which is used in parallel to the gperf table when validating object pointers.

Supervisor Thread Access Permission

Supervisor threads can access any kernel object. However, permissions for supervisor threads are still tracked for two reasons:

- If a supervisor thread calls *k_thread_user_mode_enter()*, the thread will then run in user mode with any permissions it had been granted (in many cases, by itself) when it was a supervisor thread.
- If a supervisor thread creates a user thread with the *K_INHERIT_PERMS* option, the child thread will be granted the same permissions as the parent thread, except the parent thread object.

User Thread Access Permission

By default, when a user thread is created, it will only have access permissions on its own thread object. Other kernel objects by default are not usable. Access to them needs to be explicitly or implicitly granted. There are several ways to do this.

- If a thread is created with the *K_INHERIT_PERMS*, that thread will inherit all the permissions of the parent thread, except the parent thread object.
- A thread that has permission on an object, or is running in supervisor mode, may grant permission on that object to another thread via the *k_object_access_grant()* API. The convenience pseudo-function *k_thread_access_grant()* may also be used, which accepts an arbitrary number of pointers to kernel objects and calls *k_object_access_grant()* on each of them. The thread being granted permission, or the object whose access is being granted, do not need to be in an initialized state. If the caller is from user mode, the caller must have permissions on both the kernel object and the target thread object.
- Supervisor threads may declare a particular kernel object to be a public object, usable by all current and future threads with the *k_object_access_all_grant()* API. You must assume that any untrusted or exploited code will then be able to access the object. Use this API with caution!
- If a thread was declared statically with *K_THREAD_DEFINE()*, then the *K_THREAD_ACCESS_GRANT()* may be used to grant that thread access to a set of kernel objects at boot time.

Once a thread has been granted access to an object, such access may be removed with the *k_object_access_revoke()* API. This API is not available to user threads, however user threads may use *k_object_release()* to relinquish their own permissions on an object.

API calls from supervisor mode to set permissions on kernel objects that are not being tracked by the kernel will be no-ops. Doing the same from user mode will result in a fatal error for the calling thread.

Objects allocated with *k_object_alloc()* implicitly grant permission on the allocated object to the calling thread.

Initialization State

Most operations on kernel objects will fail if the object is considered to be in an uninitialized state. The appropriate init function for the object must be performed first.

Some objects will be implicitly initialized at boot:

- Kernel objects that were declared with static initialization macros (such as *K_SEM_DEFINE* for semaphores) will be in an initialized state at build time.
- Device driver objects are considered initialized after their init function is run by the kernel early in the boot process.

If a kernel object is initialized with a private static initializer, the object must have `k_object_init()` called on it at some point by a supervisor thread, otherwise the kernel will consider the object uninitialized if accessed by a user thread. This is very uncommon, typically only for kernel objects that are embedded within some larger struct and initialized statically.

```
struct foo {
    struct k_sem sem;
    ...
};

struct foo my_foo = {
    .sem = Z_SEM_INITIALIZER(my_foo.sem, 0, 1),
```

(continues on next page)

(continued from previous page)

```
    ...
};

...
k_object_init(&my_foo.sem);
...
```

Creating New Kernel Object Types

When implementing new kernel features or driver subsystems, it may be necessary to define some new kernel object types. There are different steps needed for creating core kernel objects and new driver subsystems.

Creating New Core Kernel Objects

- In `scripts/build/gen_kobject_list.py`, add the name of the struct to the `kobjects` list.

Instances of the new struct should now be tracked.

Creating New Driver Subsystem Kernel Objects All driver instances are *device*. They are differentiated by what API struct they are set to.

- In `scripts/build/gen_kobject_list.py`, add the name of the API struct for the new subsystem to the `subsystems` list.

Driver instances of the new subsystem should now be tracked.

Configuration Options

Related configuration options:

- `CONFIG_USERSPACE`
- `CONFIG_MAX_THREAD_BYTES`

API Reference

group `usermode_apis`

Defines

`K_THREAD_ACCESS_GRANT`(name_, ...)

Grant a static thread access to a list of kernel objects.

For threads declared with *K_THREAD_DEFINE()*, grant the thread access to a set of kernel objects. These objects do not need to be in an initialized state. The permissions will be granted when the threads are initialized in the early boot sequence.

All arguments beyond the first must be pointers to kernel objects.

Parameters

- `name_` – Name of the thread, as passed to *K_THREAD_DEFINE()*

K_OBJ_FLAG_INITIALIZED
: Object initialized.

K_OBJ_FLAG_PUBLIC
: Object is Public.

K_OBJ_FLAG_ALLOC
: Object allocated.

K_OBJ_FLAG_DRIVER
: Driver Object.

Functions

void **k_object_access_grant**(const void *object, struct *k_thread* *thread)

Grant a thread access to a kernel object.

The thread will be granted access to the object if the caller is from supervisor mode, or the caller is from user mode AND has permissions on both the object and the thread whose access is being granted.

Parameters

- **object** – Address of kernel object
- **thread** – Thread to grant access to the object

void **k_object_access_revoke**(const void *object, struct *k_thread* *thread)

Revoke a thread's access to a kernel object.

The thread will lose access to the object if the caller is from supervisor mode, or the caller is from user mode AND has permissions on both the object and the thread whose access is being revoked.

Parameters

- **object** – Address of kernel object
- **thread** – Thread to remove access to the object

void **k_object_release**(const void *object)

Release an object.

Allows user threads to drop their own permission on an object Their permissions are automatically cleared when a thread terminates.

Parameters

- **object** – The object to be released

void **k_object_access_all_grant**(const void *object)

Grant all present and future threads access to an object.

If the caller is from supervisor mode, or the caller is from user mode and have sufficient permissions on the object, then that object will have permissions granted to it for *all* current and future threads running in the system, effectively becoming a public kernel object.

Use of this API should be avoided on systems that are running untrusted code as it is possible for such code to derive the addresses of kernel objects and perform unwanted operations on them.

It is not possible to revoke permissions on public objects; once public, any thread may use it.

Parameters

- **object** – Address of kernel object

bool **k_object_is_valid**(const void *obj, enum k_objects otype)

Check if a kernel object is of certain type and is valid.

This checks if the kernel object exists, of certain type, and has been initialized.

Parameters

- **obj** – Address of the kernel object
- **otype** – Object type (use K_OBJ_ANY for ignoring type checking)

Returns

True if kernel object (*obj*) exists, of certain type, and has been initialized. False otherwise.

void *__k_object_alloc__(enum k_objects otype)

Allocate a kernel object of a designated type.

This will instantiate at runtime a kernel object of the specified type, returning a pointer to it. The object will be returned in an uninitialized state, with the calling thread being granted permission on it. The memory for the object will be allocated out of the calling thread's resource pool.

Note

This function is available only if CONFIG_DYNAMIC_OBJECTS is selected.

Note

Thread stack object has to use *k_object_alloc_size()* since stacks may have different sizes.

Parameters

- **otype** – Requested kernel object type

Returns

A pointer to the allocated kernel object, or NULL if memory wasn't available

void *__k_object_alloc_size__(enum k_objects otype, size_t size)

Allocate a kernel object of a designated type and a given size.

This will instantiate at runtime a kernel object of the specified type, returning a pointer to it. The object will be returned in an uninitialized state, with the calling thread being granted permission on it. The memory for the object will be allocated out of the calling thread's resource pool.

This function is specially helpful for thread stack objects because their sizes can vary. Other objects should probably look *k_object_alloc()*.

Note

This function is available only if CONFIG_DYNAMIC_OBJECTS is selected.

Parameters

- **otype** – Requested kernel object type
- **size** – Requested kernel object size

Returns
A pointer to the allocated kernel object, or NULL if memory wasn't available

void **k_object_free**(void *obj)

Free a kernel object previously allocated with *k_object_alloc()*

This will return memory for a kernel object back to resource pool it was allocated from. Care must be exercised that the object will not be used during or after when this call is made.

> **Note**
> This function is available only if CONFIG_DYNAMIC_OBJECTS is selected.

Parameters

- **obj** – Pointer to the kernel object memory address.

3.3.4 System Calls

User threads run with a reduced set of privileges than supervisor threads: certain CPU instructions may not be used, and they have access to only a limited part of the memory map. System calls (may) allow user threads to perform operations not directly available to them.

When defining system calls, it is very important to ensure that access to the API's private data is done exclusively through system call interfaces. Private kernel data should never be made available to user mode threads directly. For example, the k_queue APIs were intentionally not made available as they store bookkeeping information about the queue directly in the queue buffers which are visible from user mode.

APIs that allow the user to register callback functions that run in supervisor mode should never be exposed as system calls. Reserve these for supervisor-mode access only.

This section describes how to declare new system calls and discusses a few implementation details relevant to them.

Components

All system calls have the following components:

- A **C prototype** prefixed with __syscall for the API. It will be declared in some header under include/ or in another SYSCALL_INCLUDE_DIRS directory. This prototype is never implemented manually, instead it gets created by the *scripts/build/gen_syscalls.py* script. What gets generated is an inline function which either calls the implementation function directly (if called from supervisor mode) or goes through privilege elevation and validation steps (if called from user mode).
- An **implementation function**, which is the real implementation of the system call. The implementation function may assume that all parameters passed in have been validated if it was invoked from user mode.
- A **verification function**, which wraps the implementation function and does validation of all the arguments passed in.

- An **unmarshalling function**, which is an automatically generated handler that must be included by user source code.

C Prototype

The C prototype represents how the API is invoked from either user or supervisor mode. For example, to initialize a semaphore:

```
__syscall void k_sem_init(struct k_sem *sem, unsigned int initial_count,
                          unsigned int limit);
```

The `__syscall` attribute is very special. To the C compiler, it simply expands to 'static inline'. However to the post-build *scripts/build/parse_syscalls.py* script, it indicates that this API is a system call. The *scripts/build/parse_syscalls.py* script does some parsing of the function prototype, to determine the data types of its return value and arguments, and has some limitations:

- Array arguments must be passed in as pointers, not arrays. For example, `int foo[]` or `int foo[12]` is not allowed, but should instead be expressed as `int *foo`.
- Function pointers horribly confuse the limited parser. The workaround is to typedef them first, and then express in the argument list in terms of that typedef.
- `__syscall` must be the first thing in the prototype.

The preprocessor is intentionally not used when determining the set of system calls to generate. However, any generated system calls that don't actually have a verification function defined (because the related feature is not enabled in the kernel configuration) will instead point to a special verification for unimplemented system calls. Data type definitions for APIs should not have conditional visibility to the compiler.

Any header file that declares system calls must include a special generated header at the very bottom of the header file. This header follows the naming convention `syscalls/<name of header file>`. For example, at the bottom of `include/sensor.h`:

```
#include <zephyr/syscalls/sensor.h>
```

C prototype functions must be declared in one of the directories listed in the CMake variable `SYSCALL_INCLUDE_DIRS`. This list always contains `APPLICATION_SOURCE_DIR` when `CONFIG_APPLICATION_DEFINED_SYSCALL` is set, or `${ZEPHYR_BASE}/subsys/testsuite/ztest/include` when `CONFIG_ZTEST` is set. Additional paths can be added to the list through the CMake command line or in CMake code that is run before `find_package(Zephyr ...)` is run. `${ZEPHYR_BASE}/include` is always scanned for potential syscall prototypes.

Note that not all syscalls will be included in the final binaries. CMake functions `zephyr_syscall_header` and `zephyr_syscall_header_ifdef` are used to specify which header files contain syscall prototypes where those syscalls must be present in the final binaries. Note that header files inside directories listed in CMake variable `SYSCALL_INCLUDE_DIRS` will always have their syscalls present in final binaries. To force all syscalls to be included in the final binaries, turn on `CONFIG_EMIT_ALL_SYSCALLS`.

Invocation Context Source code that uses system call APIs can be made more efficient if it is known that all the code inside a particular C file runs exclusively in user mode, or exclusively in supervisor mode. The system will look for the definition of macros `__ZEPHYR_SUPERVISOR__` or `__ZEPHYR_USER__`, typically these will be added to the compiler flags in the build system for the related files.

- If `CONFIG_USERSPACE` is not enabled, all APIs just directly call the implementation function.
- Otherwise, the default case is to make a runtime check to see if the processor is currently running in user mode, and either make the system call or directly call the implementation function as appropriate.

- If __ZEPHYR_SUPERVISOR__ is defined, then it is assumed that all the code runs in supervisor mode and all APIs just directly call the implementation function. If the code was actually running in user mode, there will be a CPU exception as soon as it tries to do something it isn't allowed to do.
- If __ZEPHYR_USER__ is defined, then it is assumed that all the code runs in user mode and system calls are unconditionally made.

Implementation Details Declaring an API with __syscall causes some code to be generated in C and header files by the *scripts/build/gen_syscalls.py* script, all of which can be found in the project out directory under include/generated/:

- The system call is added to the enumerated type of system call IDs, which is expressed in include/generated/zephyr/syscall_list.h. It is the name of the API in uppercase, prefixed with K_SYSCALL_.
- An entry for the system call is created in the dispatch table _k_syscall_table, expressed in include/generated/zephyr/syscall_dispatch.c
  - This table only contains syscalls where their corresponding prototypes are declared in header files when CONFIG_EMIT_ALL_SYSCALLS is enabled:
    * Indicated by CMake functions zephyr_syscall_header and zephyr_syscall_header_ifdef, or
    * Under directories specified in CMake variable SYSCALL_INCLUDE_DIRS.
- A weak verification function is declared, which is just an alias of the 'unimplemented system call' verifier. This is necessary since the real verification function may or may not be built depending on the kernel configuration. For example, if a user thread makes a sensor subsystem API call, but the sensor subsystem is not enabled, the weak verifier will be invoked instead.
- An unmarshalling function is defined in include/generated/<name>_mrsh.c

The body of the API is created in the generated system header. Using the example of *k_sem_init()*, this API is declared in include/kernel.h. At the bottom of include/kernel.h is:

```
#include <zephyr/syscalls/kernel.h>
```

Inside this header is the body of *k_sem_init()*:

```
static inline void k_sem_init(struct k_sem * sem, unsigned int initial_count, unsigned int␣
↪limit)
{
#ifdef CONFIG_USERSPACE
        if (z_syscall_trap()) {
                arch_syscall_invoke3(*(uintptr_t *)&sem, *(uintptr_t *)&initial_count,␣
↪*(uintptr_t *)&limit, K_SYSCALL_K_SEM_INIT);
                return;
        }
        compiler_barrier();
#endif
        z_impl_k_sem_init(sem, initial_count, limit);
}
```

This generates an inline function that takes three arguments with void return value. Depending on context it will either directly call the implementation function or go through a system call elevation. A prototype for the implementation function is also automatically generated.

The final layer is the invocation of the system call itself. All architectures implementing system calls must implement the seven inline functions _arch_syscall_invoke0() through _arch_syscall_invoke6(). These functions marshal arguments into designated CPU registers

and perform the necessary privilege elevation. Parameters of API inline function, before being passed as arguments to system call, are C casted to `uintptr_t` which matches size of register. Exception to above is passing 64-bit parameters on 32-bit systems, in which case 64-bit parameters are split into lower and higher part and passed as two consecutive arguments. There is always a `uintptr_t` type return value, which may be neglected if not needed.

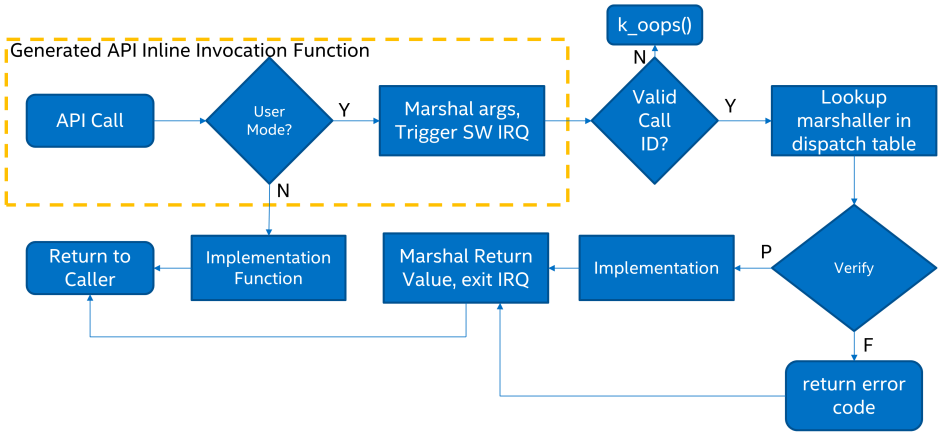


Fig. 3: System Call execution flow

Some system calls may have more than six arguments, but number of arguments passed via registers is limited to six for all architectures. Additional arguments will need to be passed in an array in the source memory space, which needs to be treated as untrusted memory in the verification function. This code (packing, unpacking and validation) is generated automatically as needed in the stub above and in the unmarshalling function.

System calls return `uintptr_t` type value that is C casted, by wrapper, to a return type of API prototype declaration. This means that 64-bit value may not be directly returned, from a system call to its wrapper, on 32-bit systems. To solve the problem the automatically generated wrapper function defines 64-bit intermediate variable, which is considered **untrusted** buffer, on its stack and passes pointer to that variable to the system call, as a final argument. Upon return from the system call the value written to that buffer will be returned by the wrapper function. The problem does not exist on 64-bit systems which are able to return 64-bit values directly.

Implementation Function

The implementation function is what actually does the work for the API. Zephyr normally does little to no error checking of arguments, or does this kind of checking with assertions. When writing the implementation function, validation of any parameters is optional and should be done with assertions.

All implementation functions must follow the naming convention, which is the name of the API prefixed with `z_impl_`. Implementation functions may be declared in the same header as the API as a static inline function or declared in some C file. There is no prototype needed for implementation functions, these are automatically generated.

Verification Function

The verification function runs on the kernel side when a user thread makes a system call. When the user thread makes a software interrupt to elevate to supervisor mode, the common system call entry point uses the system call ID provided by the user to look up the appropriate unmarshalling function for that system call and jump into it. This in turn calls the verification function.

Verification and unmarshalling functions only run when system call APIs are invoked from user mode. If an API is invoked from supervisor mode, the implementation is simply called and there is no software trap.

The purpose of the verification function is to validate all the arguments passed in. This includes:

- Any kernel object pointers provided. For example, the semaphore APIs must ensure that the semaphore object passed in is a valid semaphore and that the calling thread has permission on it.
- Any memory buffers passed in from user mode. Checks must be made that the calling thread has read or write permissions on the provided buffer.
- Any other arguments that have a limited range of valid values.

Verification functions involve a great deal of boilerplate code which has been made simpler by some macros in include/zephyr/internal/syscall_handler.h. Verification functions should be declared using these macros.

Argument Validation Several macros exist to validate arguments:

- `K_SYSCALL_OBJ()` Checks a memory address to assert that it is a valid kernel object of the expected type, that the calling thread has permissions on it, and that the object is initialized.
- `K_SYSCALL_OBJ_INIT()` is the same as `K_SYSCALL_OBJ()`, except that the provided object may be uninitialized. This is useful for verifiers of object init functions.
- `K_SYSCALL_OBJ_NEVER_INIT()` is the same as `K_SYSCALL_OBJ()`, except that the provided object must be uninitialized. This is not used very often, currently only for *k_thread_create()*.
- `K_SYSCALL_MEMORY_READ()` validates a memory buffer of a particular size. The calling thread must have read permissions on the entire buffer.
- `K_SYSCALL_MEMORY_WRITE()` is the same as `K_SYSCALL_MEMORY_READ()` but the calling thread must additionally have write permissions.
- `K_SYSCALL_MEMORY_ARRAY_READ()` validates an array whose total size is expressed as separate arguments for the number of elements and the element size. This macro correctly accounts for multiplication overflow when computing the total size. The calling thread must have read permissions on the total size.
- `K_SYSCALL_MEMORY_ARRAY_WRITE()` is the same as `K_SYSCALL_MEMORY_ARRAY_READ()` but the calling thread must additionally have write permissions.
- `K_SYSCALL_VERIFY_MSG()` does a runtime check of some boolean expression which must evaluate to true otherwise the check will fail. A variant `K_SYSCALL_VERIFY` exists which does not take a message parameter, instead printing the expression tested if it fails. The latter should only be used for the most obvious of tests.
- `K_SYSCALL_DRIVER_OP()` checks at runtime if a driver instance is capable of performing a particular operation. While this macro can be used by itself, it's mostly a building block for macros that are automatically generated for every driver subsystem. For instance, to validate the GPIO driver, one could use the `K_SYSCALL_DRIVER_GPIO()` macro.
- `K_SYSCALL_SPECIFIC_DRIVER()` is a runtime check to verify that a provided pointer is a valid instance of a specific device driver, that the calling thread has permissions on it, and that the driver has been initialized. It does this by checking the API structure pointer that is stored within the driver instance and ensuring that it matches the provided value, which should be the address of the specific driver's API structure.

If any check fails, the macros will return a nonzero value. The macro `K_OOPS()` can be used to induce a kernel oops which will kill the calling thread. This is done instead of returning some error condition to keep the APIs the same when calling from supervisor mode.

Verifier Definition All system calls are dispatched to a verifier function with a prefixed `z_vrfy_` name based on the system call. They have exactly the same return type and argument types as the wrapped system call. Their job is to execute the system call (generally by calling the implementation function) after having validated all arguments.

The verifier is itself invoked by an automatically generated unmarshaller function which takes care of unpacking the register arguments from the architecture layer and casting them to the correct type. This is defined in a header file that must be included from user code, generally somewhere after the definition of the verifier in a translation unit (so that it can be inlined).

For example:

```
static int z_vrfy_k_sem_take(struct k_sem *sem, int32_t timeout)
{
    K_OOPS(K_SYSCALL_OBJ(sem, K_OBJ_SEM));
    return z_impl_k_sem_take(sem, timeout);
}
#include <zephyr/syscalls/k_sem_take_mrsh.c>
```

Verification Memory Access Policies Parameters passed to system calls by reference require special handling, because the value of these parameters can be changed at any time by any user thread that has access to the memory that parameter points to. If the kernel makes any logical decisions based on the contents of this memory, this can open up the kernel to attacks even if checking is done. This is a class of exploits known as TOCTOU (Time Of Check to Time Of Use).

The proper procedure to mitigate these attacks is to make a copies in the verification function, and only perform parameter checks on the copies, which user threads will never have access to. The implementation functions get passed the copy and not the original data sent by the user. The `k_usermode_to_copy()` and `k_usermode_from_copy()` APIs exist for this purpose.

There is one exception in place, with respect to large data buffers which are only used to provide a memory area that is either only written to, or whose contents are never used for any validation or control flow. Further discussion of this later in this section.

As a first example, consider a parameter which is used as an output parameter for some integral value:

```
int z_vrfy_some_syscall(int *out_param)
{
    int local_out_param;
    int ret;

    ret = z_impl_some_syscall(&local_out_param);
    K_OOPS(k_usermode_to_copy(out_param, &local_out_param, sizeof(*out_param)));
    return ret;
}
```

Here we have allocated `local_out_param` on the stack, passed its address to the implementation function, and then used `k_usermode_to_copy()` to fill in the memory passed in by the caller.

It might be tempting to do something more concise:

```
int z_vrfy_some_syscall(int *out_param)
{
    K_OOPS(K_SYSCALL_MEMORY_WRITE(out_param, sizeof(*out_param)));
    return z_impl_some_syscall(out_param);
}
```

However, this is unsafe if the implementation ever does any reads to this memory as part of its logic. For example, it could be used to store some counter value, and this could be meddled with by user threads that have access to its memory. It is by far safest for small integral values to do the copying as shown in the first example.

Some parameters may be input/output. For instance, it's not uncommon to see APIs which pass in a pointer to some `size_t` which is a maximum allowable size, which is then updated by the implementation to reflect the actual number of bytes processed. This too should use a stack copy:

```
int z_vrfy_in_out_syscall(size_t *size_ptr)
{
    size_t size;
    int ret;

    K_OOPS(k_usermode_from_copy(&size, size_ptr, sizeof(size));
    ret = z_impl_in_out_syscall(&size);
    K_OOPS(k_usermode_to_copy(size_ptr, &size, sizeof(size)));
    return ret;
}
```

Many system calls pass in structures or even linked data structures. All should be copied. Typically this is done by allocating copies on the stack:

```
struct bar {
    ...
};

struct foo {
    ...
    struct bar *bar_left;
    struct bar *bar_right;
};

int z_vrfy_must_alloc(struct foo *foo)
{
    int ret;
    struct foo foo_copy;
    struct bar bar_right_copy;
    struct bar bar_left_copy;

    K_OOPS(k_usermode_from_copy(&foo_copy, foo, sizeof(*foo)));
    K_OOPS(k_usermode_from_copy(&bar_right_copy, foo_copy.bar_right,
                            sizeof(struct bar)));
    foo_copy.bar_right = &bar_right_copy;
    K_OOPS(k_usermode_from_copy(&bar_left_copy, foo_copy.bar_left,
                            sizeof(struct bar)));
    foo_copy.bar_left = &bar_left_copy;

    return z_impl_must_alloc(&foo_copy);
}
```

In some cases the amount of data isn't known at compile time or may be too large to allocate on the stack. In this scenario, it may be necessary to draw memory from the caller's resource pool via `z_thread_malloc()`. This should always be considered last resort. Functional safety programming guidelines heavily discourage usage of heap and the fact that a resource pool is used must be clearly documented. Any issues with allocation must be reported, to a caller, with returning the `-ENOMEM` . The `K_OOPS()` should never be used to verify if resource allocation has been successful.

```
struct bar {
    ...
};

struct foo {
    size_t count;
    struct bar *bar_list; /* array of struct bar of size count */
};
```

(continues on next page)

(continued from previous page)

```
int z_vrfy_must_alloc(struct foo *foo)
{
    int ret;
    struct foo foo_copy;
    struct bar *bar_list_copy;
    size_t bar_list_bytes;

    /* Safely copy foo into foo_copy */
    K_OOPS(k_usermode_from_copy(&foo_copy, foo, sizeof(*foo)));

    /* Bounds check the count member, in the copy we made */
    if (foo_copy.count > 32) {
        return -EINVAL;
    }

    /* Allocate RAM for the bar_list, replace the pointer in
     * foo_copy */
    bar_list_bytes = foo_copy.count * sizeof(struct_bar);
    bar_list_copy = z_thread_malloc(bar_list_bytes);
    if (bar_list_copy == NULL) {
        return -ENOMEM;
    }
    K_OOPS(k_usermode_from_copy(bar_list_copy, foo_copy.bar_list,
                                bar_list_bytes));
    foo_copy.bar_list = bar_list_copy;

    ret = z_impl_must_alloc(&foo_copy);

    /* All done with the memory, free it and return */
    k_free(foo_copy.bar_list_copy);
    return ret;
}
```

Finally, we must consider large data buffers. These represent areas of user memory which either have data copied out of, or copied into. It is permitted to pass these pointers to the implementation function directly. The caller's access to the buffer still must be validated with K_SYSCALL_MEMORY APIs. The following constraints need to be met:

- If the buffer is used by the implementation function to write data, such as data captured from some MMIO region, the implementation function must only write this data, and never read it.
- If the buffer is used by the implementation function to read data, such as a block of memory to write to some hardware destination, this data must be read without any processing. No conditional logic can be implemented due to the data buffer's contents. If such logic is required a copy must be made.
- The buffer must only be used synchronously with the call. The implementation must not ever save the buffer address and use it asynchronously, such as when an interrupt fires.

```
int z_vrfy_get_data_from_kernel(void *buf, size_t size)
{
    K_OOPS(K_SYSCALL_MEMORY_WRITE(buf, size));
    return z_impl_get_data_from_kernel(buf, size);
}
```

Verification Return Value Policies When verifying system calls, it's important to note which kinds of verification failures should propagate a return value to the caller, and which should simply invoke K_OOPS() which kills the calling thread. The current conventions are as follows:

1. For system calls that are defined but not compiled, invocations of these missing system calls are routed to `handler_no_syscall()` which invokes `K_OOPS()`.
2. Any invalid access to memory found by the set of `K_SYSCALL_MEMORY` APIs, `k_usermode_from_copy()`, `k_usermode_to_copy()` should trigger a `K_OOPS`. This happens when the caller doesn't have appropriate permissions on the memory buffer or some size calculation overflowed.
3. Most system calls take kernel object pointers as an argument, checked either with one of the `K_SYSCALL_OBJ` functions, `K_SYSCALL_DRIVER_nnnnn`, or manually using `k_object_validate()`. These can fail for a variety of reasons: missing driver API, bad kernel object pointer, wrong kernel object type, or improper initialization state. These issues should always invoke `K_OOPS()`.
4. Any error resulting from a failed memory heap allocation, often from invoking `z_thread_malloc()`, should propagate `-ENOMEM` to the caller.
5. General parameter checks should be done in the implementation function, in most cases using `CHECKIF()`.
    - The behavior of `CHECKIF()` depends on the kernel configuration, but if user mode is enabled, `CONFIG_RUNTIME_ERROR_CHECKS` is enforced, which guarantees that these checks will be made and a return value propagated.
6. It is totally forbidden for any kind of kernel mode callback function to be registered from user mode. APIs which simply install callbacks shall not be exposed as system calls. Some driver subsystem APIs may take optional function callback pointers. User mode verification functions for these APIs must enforce that these are NULL and should invoke `K_OOPS()` if not.
7. Some parameter checks are enforced only from user mode. These should be checked in the verification function and propagate a return value to the caller if possible.

There are some known exceptions to these policies currently in Zephyr:

- *`k_thread_join()`* and *`k_thread_abort()`* are no-ops if the thread object isn't initialized. This is because for threads, the initialization bit pulls double-duty to indicate whether a thread is running, cleared upon exit. See #23030.
- *`k_thread_create()`* invokes `K_OOPS()` for parameter checks, due to a great deal of existing code ignoring the return value. This will also be addressed by #23030.
- *`k_thread_abort()`* invokes `K_OOPS()` if an essential thread is aborted, as the function has no return value.
- Various system calls related to logging invoke `K_OOPS()` when bad parameters are passed in as they do not propagate errors.

Configuration Options

Related configuration options:

- `CONFIG_USERSPACE`
- `CONFIG_EMIT_ALL_SYSCALLS`

APIs

Helper macros for creating system call verification functions are provided in include/zephyr/internal/syscall_handler.h:

- `K_SYSCALL_OBJ()`
- `K_SYSCALL_OBJ_INIT()`

- K_SYSCALL_OBJ_NEVER_INIT()
- K_OOPS()
- K_SYSCALL_MEMORY_READ()
- K_SYSCALL_MEMORY_WRITE()
- K_SYSCALL_MEMORY_ARRAY_READ()
- K_SYSCALL_MEMORY_ARRAY_WRITE()
- K_SYSCALL_VERIFY_MSG()
- K_SYSCALL_VERIFY

Functions for invoking system calls are defined in include/zephyr/syscall.h:

- _arch_syscall_invoke0()
- _arch_syscall_invoke1()
- _arch_syscall_invoke2()
- _arch_syscall_invoke3()
- _arch_syscall_invoke4()
- _arch_syscall_invoke5()
- _arch_syscall_invoke6()

3.3.5 MPU Stack Objects

Thread Stack Creation

Thread stacks are declared statically with *K_THREAD_STACK_DEFINE()*.

For architectures which utilize memory protection unit (MPU) hardware, stacks are physically contiguous allocations. This contiguous allocation has implications for the placement of stacks in memory, as well as the implementation of other features such as stack protection and userspace. The implications for placement are directly attributed to the alignment requirements for MPU regions. This is discussed in the memory placement section below.

Stack Guards

Stack protection mechanisms require hardware support that can restrict access to memory. Memory protection units can provide this kind of support. The MPU provides a fixed number of regions. Each region contains information about the start, end, size, and access attributes to be enforced on that particular region.

Stack guards are implemented by using a single MPU region and setting the attributes for that region to not allow write access. If invalid accesses occur, a fault ensues. The stack guard is defined at the bottom (the lowest address) of the stack.

Memory Placement

During stack creation, a set of constraints are enforced on the allocation of memory. These constraints include determining the alignment of the stack and the correct sizing of the stack. During linking of the binary, these constraints are used to place the stacks properly.

The main source of the memory constraints is the MPU design for the SoC. The MPU design may require specific constraints on the region definition. These can include alignment of beginning and end addresses, sizes of allocations, or even interactions between overlapping regions.

Some MPUs require that each region be aligned to a power of two. These SoCs will have CONFIG_MPU_REQUIRES_POWER_OF_TWO_ALIGNMENT defined. This means that a 1500 byte stack should be aligned to a 2kB boundary and the stack size should also be adjusted to 2kB to ensure that nothing else is placed in the remainder of the region. SoCs which include the unmodified ARM v7m MPU will have these constraints.

Some ARM MPUs use start and end addresses to define MPU regions and both the start and end addresses require 32 byte alignment. An example of this kind of MPU is found in the NXP FRDM K64F.

MPUs may have a region priority mechanisms that use the highest priority region that covers the memory access to determine the enforcement policy. Others may logically OR regions to determine enforcement policy.

Size and alignment constraints may result in stack allocations being larger than the requested size. Region priority mechanisms may result in some added complexity when implementing stack guards.

3.3.6 MPU Backed Userspace

The MPU backed userspace implementation requires the creation of a secondary set of stacks. These stacks exist in a 1:1 relationship with each thread stack defined in the system. The privileged stacks are created as a part of the build process.

A post-build script *scripts/build/gen_kobject_list.py* scans the generated ELF file and finds all of the thread stack objects. A set of privileged stacks, a lookup table, and a set of helper functions are created and added to the image.

During the process of dropping a thread to user mode, the privileged stack information is filled in and later used by the swap and system call infrastructure to configure the MPU regions properly for the thread stack and guard (if applicable).

During system calls, the user mode thread's access to the system call and the passed-in parameters are all validated. The user mode thread is then elevated to privileged mode, the stack is switched to use the privileged stack, and the call is made to the specified kernel API. On return from the kernel API, the thread is set back to user mode and the stack is restored to the user stack.

3.4 Memory Management

The following contains various topics regarding memory management.

3.4.1 Memory Heaps

Zephyr provides a collection of utilities that allow threads to dynamically allocate memory.

Synchronized Heap Allocator

Creating a Heap The simplest way to define a heap is statically, with the *K_HEAP_DEFINE* macro. This creates a static *k_heap* variable with a given name that manages a memory region of the specified size.

Heaps can also be created to manage arbitrary regions of application-controlled memory using *k_heap_init()*.

Allocating Memory Memory can be allocated from a heap using *k_heap_alloc()*, passing it the address of the heap object and the number of bytes desired. This functions similarly to standard C `malloc()`, returning a NULL pointer on an allocation failure.

The heap supports blocking operation, allowing threads to go to sleep until memory is available. The final argument is a *k_timeout_t* timeout value indicating how long the thread may sleep before returning, or else one of the constant timeout values *K_NO_WAIT* or *K_FOREVER*.

Releasing Memory Memory allocated with *k_heap_alloc()* must be released using *k_heap_free()*. Similar to standard C `free()`, the pointer provided must be either a `NULL` value or a pointer previously returned by *k_heap_alloc()* for the same heap. Freeing a `NULL` value is defined to have no effect.

Low Level Heap Allocator

The underlying implementation of the *k_heap* abstraction is provided a data structure named `sys_heap`. This implements exactly the same allocation semantics, but provides no kernel synchronization tools. It is available for applications that want to manage their own blocks of memory in contexts (for example, userspace) where synchronization is unavailable or more complicated. Unlike `k_heap`, all calls to any `sys_heap` functions on a single heap must be serialized by the caller. Simultaneous use from separate threads is disallowed.

Implementation Internally, the `sys_heap` memory block is partitioned into "chunks" of 8 bytes. All allocations are made out of a contiguous region of chunks. The first chunk of every allocation or unused block is prefixed by a chunk header that stores the length of the chunk, the length of the next lower ("left") chunk in physical memory, a bit indicating whether the chunk is in use, and chunk-indexed link pointers to the previous and next chunk in a "free list" to which unused chunks are added.

The heap code takes reasonable care to avoid fragmentation. Free block lists are stored in "buckets" by their size, each bucket storing blocks within one power of two (i.e. a bucket for blocks of 3-4 chunks, another for 5-8, 9-16, etc...) this allows new allocations to be made from the smallest/most-fragmented blocks available. Also, as allocations are freed and added to the heap, they are automatically combined with adjacent free blocks to prevent fragmentation.

All metadata is stored at the beginning of the contiguous block of heap memory, including the variable-length list of bucket list heads (which depend on heap size). The only external memory required is the `sys_heap` structure itself.

The `sys_heap` functions are unsynchronized. Care must be taken by any users to prevent concurrent access. Only one context may be inside one of the API functions at a time.

The heap code takes care to present high performance and reliable latency. All `sys_heap` API functions are guaranteed to complete within constant time. On typical architectures, they will all complete within 1-200 cycles. One complexity is that the search of the minimum bucket size for an allocation (the set of free blocks that "might fit") has a compile-time upper bound of iterations to prevent unbounded list searches, at the expense of some fragmentation resistance. This `CONFIG_SYS_HEAP_ALLOC_LOOPS` value may be chosen by the user at build time, and defaults to a value of 3.

Multi-Heap Wrapper Utility

The sys_heap utility requires that all managed memory be in a single contiguous block. It is common for complicated microcontroller applications to have more complicated memory setups that they still want to manage dynamically as a "heap". For example, the memory might exist as separate discontiguous regions, different areas may have different cache, performance or power behavior, peripheral devices may only be able to perform DMA to certain regions, etc...

For those situations, Zephyr provides a sys_multi_heap utility. Effectively this is a simple wrapper around a set of one or more sys_heap objects. It should be initialized after its child heaps via *sys_multi_heap_init()*, after which each heap can be added to the managed set via *sys_multi_heap_add_heap()*. No destruction utility is provided; just as for sys_heap, applications that want to destroy a multi heap should simply ensure all allocated blocks are freed (or at least will never be used again) and repurpose the underlying memory for another usage.

It has a single pair of allocation entry points, *sys_multi_heap_alloc()* and *sys_multi_heap_aligned_alloc()*. These behave identically to the sys_heap functions with similar names, except that they also accept an opaque "configuration" parameter. This pointer is uninspected by the multi heap code itself; instead it is passed to a callback function provided at initialization time. This application-provided callback is responsible for doing the underlying allocation from one of the managed heaps, and may use the configuration parameter in any way it likes to make that decision.

When unused, a multi heap may be freed via *sys_multi_heap_free()*. The application does not need to pass a configuration parameter. Memory allocated from any of the managed sys_heap objects may be freed with in the same way.

System Heap

The *system heap* is a predefined memory allocator that allows threads to dynamically allocate memory from a common memory region in a malloc()-like manner.

Only a single system heap is defined. Unlike other heaps or memory pools, the system heap cannot be directly referenced using its memory address.

The size of the system heap is configurable to arbitrary sizes, subject to space availability.

A thread can dynamically allocate a chunk of heap memory by calling *k_malloc()*. The address of the allocated chunk is guaranteed to be aligned on a multiple of pointer sizes. If a suitable chunk of heap memory cannot be found NULL is returned.

When the thread is finished with a chunk of heap memory it can release the chunk back to the system heap by calling *k_free()*.

Defining the Heap Memory Pool The size of the heap memory pool is specified using the CONFIG_HEAP_MEM_POOL_SIZE configuration option.

By default, the heap memory pool size is zero bytes. This value instructs the kernel not to define the heap memory pool object. The maximum size is limited by the amount of available memory in the system. The project build will fail in the link stage if the size specified can not be supported.

In addition, each subsystem (board, driver, library, etc) can set a custom requirement by defining a Kconfig option with the prefix HEAP_MEM_POOL_ADD_SIZE_ (this value is in bytes). If multiple subsystems specify custom values, the sum of these will be used as the minimum requirement. If the application tries to set a value that's less than the minimum value, this will be ignored and the minimum value will be used instead.

To force a smaller than minimum value to be used, the application may enable the CONFIG_HEAP_MEM_POOL_IGNORE_MIN option. This can be useful when optimizing the heap size and the minimum requirement can be more accurately determined for a specific application.

Allocating Memory A chunk of heap memory is allocated by calling *k_malloc()*.

The following code allocates a 200 byte chunk of heap memory, then fills it with zeros. A warning is issued if a suitable chunk is not obtained.

```
char *mem_ptr;

mem_ptr = k_malloc(200);
if (mem_ptr != NULL)) {
    memset(mem_ptr, 0, 200);
    ...
} else {
    printf("Memory not allocated");
}
```

Releasing Memory A chunk of heap memory is released by calling *k_free()*.

The following code allocates a 75 byte chunk of memory, then releases it once it is no longer needed.

```
char *mem_ptr;

mem_ptr = k_malloc(75);
... /* use memory block */
k_free(mem_ptr);
```

Suggested Uses Use the heap memory pool to dynamically allocate memory in a `malloc()`-like manner.

Configuration Options Related configuration options:

- `CONFIG_HEAP_MEM_POOL_SIZE`

API Reference

group `heap_apis`

Defines

`K_HEAP_DEFINE`(name, bytes)

Define a static *k_heap*.

This macro defines and initializes a static memory region and *k_heap* of the requested size. After kernel start, &name can be used as if *k_heap_init()* had been called.

Note that this macro enforces a minimum size on the memory region to accommodate metadata requirements. Very small heaps will be padded to fit.

Parameters

- `name` – Symbol name for the struct *k_heap* object
- `bytes` – Size of memory region, in bytes

K_HEAP_DEFINE_NOCACHE(name, bytes)

Define a static *k_heap* in uncached memory.

This macro defines and initializes a static memory region and *k_heap* of the requested size in uncached memory. After kernel start, &name can be used as if *k_heap_init()* had been called.

Note that this macro enforces a minimum size on the memory region to accommodate metadata requirements. Very small heaps will be padded to fit.

Parameters

- `name` – Symbol name for the struct *k_heap* object
- `bytes` – Size of memory region, in bytes

Functions

void **k_heap_init**(struct *k_heap* *h, void *mem, size_t bytes)

Initialize a *k_heap*.

This constructs a synchronized *k_heap* object over a memory region specified by the user. Note that while any alignment and size can be passed as valid parameters, internal alignment restrictions inside the inner sys_heap mean that not all bytes may be usable as allocated memory.

Parameters

- `h` – Heap struct to initialize
- `mem` – Pointer to memory.
- `bytes` – Size of memory region, in bytes

void ***k_heap_aligned_alloc**(struct *k_heap* *h, size_t align, size_t bytes, *k_timeout_t* timeout)

Allocate aligned memory from a *k_heap*.

Behaves in all ways like *k_heap_alloc()*, except that the returned memory (if available) will have a starting address in memory which is a multiple of the specified power-of-two alignment value in bytes. The resulting memory can be returned to the heap using *k_heap_free()*.

Function properties (list may not be complete)

isr-ok

> **Note**
>
> *timeout* must be set to K_NO_WAIT if called from ISR.

> **Note**
>
> When CONFIG_MULTITHREADING=n any *timeout* is treated as K_NO_WAIT.

Parameters

- `h` – Heap from which to allocate
- `align` – Alignment in bytes, must be a power of two

- **bytes** – Number of bytes requested
- **timeout** – How long to wait, or K_NO_WAIT

Returns
Pointer to memory the caller can now use

void *k_heap_alloc(struct *k_heap* *h, size_t bytes, *k_timeout_t* timeout)

Allocate memory from a *k_heap*.

Allocates and returns a memory buffer from the memory region owned by the heap. If no memory is available immediately, the call will block for the specified timeout (constructed via the standard timeout API, or K_NO_WAIT or K_FOREVER) waiting for memory to be freed. If the allocation cannot be performed by the expiration of the timeout, NULL will be returned. Allocated memory is aligned on a multiple of pointer sizes.

Function properties (list may not be complete)
isr-ok

Note

timeout must be set to K_NO_WAIT if called from ISR.

Note

When CONFIG_MULTITHREADING=n any *timeout* is treated as K_NO_WAIT.

Parameters

- **h** – Heap from which to allocate
- **bytes** – Desired size of block to allocate
- **timeout** – How long to wait, or K_NO_WAIT

Returns
A pointer to valid heap memory, or NULL

void *k_heap_realloc(struct *k_heap* *h, void *ptr, size_t bytes, *k_timeout_t* timeout)

Reallocate memory from a *k_heap*.

Reallocates and returns a memory buffer from the memory region owned by the heap. If no memory is available immediately, the call will block for the specified timeout (constructed via the standard timeout API, or K_NO_WAIT or K_FOREVER) waiting for memory to be freed. If the allocation cannot be performed by the expiration of the timeout, NULL will be returned. Reallocated memory is aligned on a multiple of pointer sizes.

Function properties (list may not be complete)
isr-ok

Note

timeout must be set to K_NO_WAIT if called from ISR.

Note

When CONFIG_MULTITHREADING=n any *timeout* is treated as K_NO_WAIT.

Parameters

- **h** – Heap from which to allocate
- **ptr** – Original pointer returned from a previous allocation
- **bytes** – Desired size of block to allocate
- **timeout** – How long to wait, or K_NO_WAIT

Returns
Pointer to memory the caller can now use, or NULL

void **k_heap_free**(struct *k_heap* *h, void *mem)

Free memory allocated by *k_heap_alloc()*

Returns the specified memory block, which must have been returned from *k_heap_alloc()*, to the heap for use by other callers. Passing a NULL block is legal, and has no effect.

Parameters

- **h** – Heap to which to return the memory
- **mem** – A valid memory block, or NULL

void ***k_aligned_alloc**(size_t align, size_t size)

Allocate memory from the heap with a specified alignment.

This routine provides semantics similar to aligned_alloc(); memory is allocated from the heap with a specified alignment. However, one minor difference is that *k_aligned_alloc()* accepts any non-zero `size`, whereas aligned_alloc() only accepts a `size` that is an integral multiple of `align`.

Above, aligned_alloc() refers to: C11 standard (ISO/IEC 9899:2011): 7.22.3.1 The aligned_alloc function (p: 347-348)

Parameters

- **align** – Alignment of memory requested (in bytes).
- **size** – Amount of memory requested (in bytes).

Returns
Address of the allocated memory if successful; otherwise NULL.

void ***k_malloc**(size_t size)

Allocate memory from the heap.

This routine provides traditional malloc() semantics. Memory is allocated from the heap memory pool. Allocated memory is aligned on a multiple of pointer sizes.

Parameters

- **size** – Amount of memory requested (in bytes).

Returns
Address of the allocated memory if successful; otherwise NULL.

void k_free(void *ptr)

Free memory allocated from heap.

This routine provides traditional free() semantics. The memory being returned must have been allocated from the heap memory pool.

If *ptr* is NULL, no operation is performed.

Parameters

- ptr – Pointer to previously allocated memory.

void *k_calloc(size_t nmemb, size_t size)

Allocate memory from heap, array style.

This routine provides traditional calloc() semantics. Memory is allocated from the heap memory pool and zeroed.

Parameters

- nmemb – Number of elements in the requested array
- size – Size of each array element (in bytes).

Returns

Address of the allocated memory if successful; otherwise NULL.

void *k_realloc(void *ptr, size_t size)

Expand the size of an existing allocation.

Returns a pointer to a new memory region with the same contents, but a different allocated size. If the new allocation can be expanded in place, the pointer returned will be identical. Otherwise the data will be copies to a new block and the old one will be freed as per *sys_heap_free()*. If the specified size is smaller than the original, the block will be truncated in place and the remaining memory returned to the heap. If the allocation of a new block fails, then NULL will be returned and the old block will not be freed or modified.

Parameters

- ptr – Original pointer returned from a previous allocation
- size – Amount of memory requested (in bytes).

Returns

Pointer to memory the caller can now use, or NULL.

struct k_heap

#include <kernel.h>

group low_level_heap_allocator

Defines

sys_heap_realloc(heap, ptr, bytes)

Functions

void sys_heap_init(struct sys_heap *heap, void *mem, size_t bytes)

Initialize sys_heap.

Initializes a sys_heap struct to manage the specified memory.

Parameters

- `heap` – Heap to initialize
- `mem` – Untyped pointer to unused memory
- `bytes` – Size of region pointed to by *mem*

void *`sys_heap_alloc`(struct sys_heap *heap, size_t bytes)

Allocate memory from a sys_heap.

Returns a pointer to a block of unused memory in the heap. This memory will not otherwise be used until it is freed with *sys_heap_free()*. If no memory can be allocated, NULL will be returned. The allocated memory is guaranteed to have a starting address which is a multiple of sizeof(void *). If a bigger alignment is necessary then *sys_heap_aligned_alloc()* should be used instead.

Note

The sys_heap implementation is not internally synchronized. No two sys_heap functions should operate on the same heap at the same time. All locking must be provided by the user.

Parameters

- `heap` – Heap from which to allocate
- `bytes` – Number of bytes requested

Returns

Pointer to memory the caller can now use

void *`sys_heap_aligned_alloc`(struct sys_heap *heap, size_t align, size_t bytes)

Allocate aligned memory from a sys_heap.

Behaves in all ways like *sys_heap_alloc()*, except that the returned memory (if available) will have a starting address in memory which is a multiple of the specified power-of-two alignment value in bytes. With align=0 this behaves exactly like *sys_heap_alloc()*. The resulting memory can be returned to the heap using *sys_heap_free()*.

Parameters

- `heap` – Heap from which to allocate
- `align` – Alignment in bytes, must be a power of two
- `bytes` – Number of bytes requested

Returns

Pointer to memory the caller can now use

void `sys_heap_free`(struct sys_heap *heap, void *mem)

Free memory into a sys_heap.

De-allocates a pointer to memory previously returned from sys_heap_alloc such that it can be used for other purposes. The caller must not use the memory region after entry to this function.

Note

The sys_heap implementation is not internally synchronized. No two sys_heap functions should operate on the same heap at the same time. All locking must be provided by the user.

Parameters

- `heap` – Heap to which to return the memory
- `mem` – A pointer previously returned from *sys_heap_alloc()*

void *`sys_heap_aligned_realloc`(struct sys_heap *heap, void *ptr, size_t align, size_t bytes)

Expand the size of an existing allocation.

Returns a pointer to a new memory region with the same contents, but a different allocated size. If the new allocation can be expanded in place, the pointer returned will be identical. Otherwise the data will be copies to a new block and the old one will be freed as per *sys_heap_free()*. If the specified size is smaller than the original, the block will be truncated in place and the remaining memory returned to the heap. If the allocation of a new block fails, then NULL will be returned and the old block will not be freed or modified.

Parameters

- `heap` – Heap from which to allocate
- `ptr` – Original pointer returned from a previous allocation
- `align` – Alignment in bytes, must be a power of two
- `bytes` – Number of bytes requested for the new block

Returns

Pointer to memory the caller can now use, or NULL

size_t `sys_heap_usable_size`(struct sys_heap *heap, void *mem)

Return allocated memory size.

Returns the size, in bytes, of a block returned from a successful *sys_heap_alloc()* or sys_heap_alloc_aligned() call. The value returned is the size of the heap-managed memory, which may be larger than the number of bytes requested due to allocation granularity. The heap code is guaranteed to make no access to this region of memory until a subsequent *sys_heap_free()* on the same pointer.

Parameters

- `heap` – Heap containing the block
- `mem` – Pointer to memory allocated from this heap

Returns

Size in bytes of the memory region

bool `sys_heap_validate`(struct sys_heap *heap)

Validate heap integrity.

Validates the internal integrity of a sys_heap. Intended for unit test and validation code, though potentially useful as a user API for applications with complicated runtime reliability requirements. Note: this cannot catch every possible error, but if it returns true then the heap is in a consistent state and can correctly handle any *sys_heap_alloc()* request and free any live pointer returned from a previous allocation.

Parameters

- `heap` – Heap to validate

Returns
true, if the heap is valid, otherwise false

void **sys_heap_stress**(void *(*alloc_fn)(void *arg, size_t bytes), void (*free_fn)(void *arg, void *p), void *arg, size_t total_bytes, uint32_t op_count, void *scratch_mem, size_t scratch_bytes, int target_percent, struct z_heap_stress_result *result)

sys_heap stress test rig

Test rig for heap allocation validation. This will loop for *op_count* cycles, in each iteration making a random choice to allocate or free a pointer of randomized (power law) size based on heuristics designed to keep the heap in a state where it is near *target_percent* full. Allocation and free operations are provided by the caller as callbacks (i.e. this can in theory test any heap). Results, including counts of frees and successful/unsuccessful allocations, are returned via the *result* struct.

Parameters

- **alloc_fn** – Callback to perform an allocation. Passes back the *arg* parameter as a context handle.
- **free_fn** – Callback to perform a free of a pointer returned from *alloc*. Passes back the *arg* parameter as a context handle.
- **arg** – Context handle to pass back to the callbacks
- **total_bytes** – Size of the byte array the heap was initialized in
- **op_count** – How many iterations to test
- **scratch_mem** – A pointer to scratch memory to be used by the test. Should be about 1/2 the size of the heap for tests that need to stress fragmentation.
- **scratch_bytes** – Size of the memory pointed to by *scratch_mem*
- **target_percent** – Percentage fill value (1-100) to which the random allocation choices will seek. High values will result in significant allocation failures and a very fragmented heap.
- **result** – Struct into which to store test results.

void **sys_heap_print_info**(struct sys_heap *heap, bool dump_chunks)

Print heap internal structure information to the console.

Print information on the heap structure such as its size, chunk buckets, chunk list and some statistics for debugging purpose.

Parameters

- **heap** – Heap to print information about
- **dump_chunks** – True to print the entire heap chunk list

group **multi_heap_wrapper**

Typedefs

typedef void *(***sys_multi_heap_fn_t**)(struct *sys_multi_heap* *mheap, void *cfg, size_t align, size_t size)

Multi-heap choice function.

This is a user-provided functions whose responsibility is selecting a specific sys_heap backend based on the opaque cfg value, which is specified by the user as an argument

to *sys_multi_heap_alloc()*, and performing the allocation on behalf of the caller. The callback is free to choose any registered heap backend to perform the allocation, and may choose to pad the user-provided values as needed, and to use an aligned allocation where required by the specified configuration.

NULL may be returned, which will cause the allocation to fail and a NULL reported to the calling code.

Param mheap
Multi-heap structure.

Param cfg
An opaque user-provided value. It may be interpreted in any way by the application

Param align
Alignment of requested memory (or zero for no alignment)

Param size
The user-specified allocation size in bytes

Return
A pointer to the allocated memory

Functions

void **sys_multi_heap_init**(struct *sys_multi_heap* *heap, *sys_multi_heap_fn_t* choice_fn)

Initialize multi-heap.

Initialize a *sys_multi_heap* struct with the specified choice function. Note that individual heaps must be added later with sys_multi_heap_add_heap so that the heap bounds can be tracked by the multi heap code.

Note

In general a multiheap is likely to be instantiated semi-statically from system configuration (for example, via linker-provided bounds on available memory in different regions, or from devicetree definitions of hardware-provided addressable memory, etc...). The general expectation is that a soc- or board-level platform device will be initialized at system boot from these upstream configuration sources and not that an application will assemble a multi-heap on its own.

Parameters

- **heap** – A *sys_multi_heap* to initialize
- **choice_fn** – A sys_multi_heap_fn_t callback used to select heaps at allocation time

void **sys_multi_heap_add_heap**(struct *sys_multi_heap* *mheap, struct sys_heap *heap, void *user_data)

Add sys_heap to multi heap.

This adds a known sys_heap backend to an existing multi heap, allowing the multi heap internals to track the bounds of the heap and determine which heap (if any) from which a freed block was allocated.

Parameters

- **mheap** – A *sys_multi_heap* to which to add a heap
- **heap** – The heap to add

- **user_data** – pointer to any data for the heap

void ***sys_multi_heap_alloc**(struct *sys_multi_heap* *mheap, void *cfg, size_t bytes)

Allocate memory from multi heap.

Just as for *sys_heap_alloc()*, allocates a block of memory of the specified size in bytes. Takes an opaque configuration pointer passed to the multi heap choice function, which is used by integration code to choose a heap backend.

Parameters

- **mheap** – Multi heap pointer
- **cfg** – Opaque configuration parameter, as for sys_multi_heap_fn_t
- **bytes** – Requested size of the allocation, in bytes

Returns

A valid pointer to heap memory, or NULL if no memory is available

void ***sys_multi_heap_aligned_alloc**(struct *sys_multi_heap* *mheap, void *cfg, size_t align, size_t bytes)

Allocate aligned memory from multi heap.

Just as for *sys_multi_heap_alloc()*, allocates a block of memory of the specified size in bytes. Takes an additional parameter specifying a power of two alignment, in bytes.

Parameters

- **mheap** – Multi heap pointer
- **cfg** – Opaque configuration parameter, as for sys_multi_heap_fn_t
- **align** – Power of two alignment for the returned pointer, in bytes
- **bytes** – Requested size of the allocation, in bytes

Returns

A valid pointer to heap memory, or NULL if no memory is available

const struct *sys_multi_heap_rec* ***sys_multi_heap_get_heap**(const struct *sys_multi_heap* *mheap, void *addr)

Get a specific heap for provided address.

Finds a single system heap (with user_data) controlling the provided pointer

Parameters

- **mheap** – Multi heap pointer
- **addr** – address to be found, must be a pointer to a block allocated by sys_multi_heap_alloc

Returns

0 multi_heap_rec pointer to a structure to be filled with return data or NULL if the heap has not been found

void **sys_multi_heap_free**(struct *sys_multi_heap* *mheap, void *block)

Free memory allocated from multi heap.

Returns the specified block, which must be the return value of a previously successful *sys_multi_heap_alloc()* or *sys_multi_heap_aligned_alloc()* call, to the heap backend from which it was allocated.

Accepts NULL as a block parameter, which is specified to have no effect.

Parameters

- **mheap** – Multi heap pointer

- **block** – Block to free, must be a pointer to a block allocated by sys_multi_heap_alloc

struct **sys_multi_heap_rec**

#include <*multi_heap.h*>

struct **sys_multi_heap**

#include <*multi_heap.h*>

Heap listener

group **heap_listener_apis**

Defines

HEAP_ID_FROM_POINTER(heap_pointer)

Construct heap identifier from heap pointer.

Construct a heap identifier from a pointer to the heap object, such as sys_heap.

Parameters

- **heap_pointer** – Pointer to the heap object

HEAP_ID_LIBC

Libc heap identifier.

Identifier of the global libc heap.

HEAP_LISTENER_ALLOC_DEFINE(name, _heap_id, _alloc_cb)

Define heap event listener node for allocation event.

Sample usage:

```
void on_heap_alloc(uintptr_t heap_id, void *mem, size_t bytes)
{
  LOG_INF("Memory allocated at %p, size %ld", mem, bytes);
}

HEAP_LISTENER_ALLOC_DEFINE(my_listener, HEAP_ID_LIBC, on_heap_alloc);
```

Parameters

- **name** – Name of the heap event listener object
- **_heap_id** – Identifier of the heap to be listened
- **_alloc_cb** – Function to be called for allocation event

HEAP_LISTENER_FREE_DEFINE(name, _heap_id, _free_cb)

Define heap event listener node for free event.

Sample usage:

```
void on_heap_free(uintptr_t heap_id, void *mem, size_t bytes)
{
  LOG_INF("Memory freed at %p, size %ld", mem, bytes);
}

HEAP_LISTENER_FREE_DEFINE(my_listener, HEAP_ID_LIBC, on_heap_free);
```

Parameters

- **name** – Name of the heap event listener object
- **_heap_id** – Identifier of the heap to be listened
- **_free_cb** – Function to be called for free event

HEAP_LISTENER_RESIZE_DEFINE(name, _heap_id, _resize_cb)

Define heap event listener node for resize event.

Sample usage:

```
void on_heap_resized(uintptr_t heap_id, void *old_heap_end, void *new_heap_end)
{
  LOG_INF("Libc heap end moved from %p to %p", old_heap_end, new_heap_end);
}

HEAP_LISTENER_RESIZE_DEFINE(my_listener, HEAP_ID_LIBC, on_heap_resized);
```

Parameters

- **name** – Name of the heap event listener object
- **_heap_id** – Identifier of the heap to be listened
- **_resize_cb** – Function to be called when the listened heap is resized

Typedefs

typedef void (*heap_listener_resize_cb_t)(uintptr_t heap_id, void *old_heap_end, void *new_heap_end)

Callback used when heap is resized.

> **Note**
>
> Minimal C library does not emit this event.

Param heap_id
Identifier of heap being resized

Param old_heap_end
Pointer to end of heap before resize

Param new_heap_end
Pointer to end of heap after resize

typedef void (*heap_listener_alloc_cb_t)(uintptr_t heap_id, void *mem, size_t bytes)

Callback used when there is heap allocation.

Note

Heaps managed by libraries outside of code in Zephyr main code repository may not emit this event.

Note

The number of bytes allocated may not match exactly to the request to the allocation function. Internal mechanism of the heap may allocate more than requested.

Param heap_id
Heap identifier

Param mem
Pointer to the allocated memory

Param bytes
Size of allocated memory

typedef void (*`heap_listener_free_cb_t`)(uintptr_t heap_id, void *mem, size_t bytes)

Callback used when memory is freed from heap.

Note

Heaps managed by libraries outside of code in Zephyr main code repository may not emit this event.

Note

The number of bytes freed may not match exactly to the request to the allocation function. Internal mechanism of the heap dictates how memory is allocated or freed.

Param heap_id
Heap identifier

Param mem
Pointer to the freed memory

Param bytes
Size of freed memory

Enums

enum `heap_event_types`

Values:

enumerator `HEAP_EVT_UNKNOWN` = 0

enumerator `HEAP_RESIZE`

enumerator HEAP_ALLOC

enumerator HEAP_FREE

enumerator HEAP_REALLOC

enumerator HEAP_MAX_EVENTS

Functions

void heap_listener_register(struct *heap_listener* *listener)

Register heap event listener.

Add the listener to the global list of heap listeners that can be notified by different heap implementations upon certain events related to the heap usage.

Parameters

- listener – Pointer to the *heap_listener* object

void heap_listener_unregister(struct *heap_listener* *listener)

Unregister heap event listener.

Remove the listener from the global list of heap listeners that can be notified by different heap implementations upon certain events related to the heap usage.

Parameters

- listener – Pointer to the *heap_listener* object

void heap_listener_notify_alloc(uintptr_t heap_id, void *mem, size_t bytes)

Notify listeners of heap allocation event.

Notify registered heap event listeners with matching heap identifier that an allocation has been done on heap

Parameters

- heap_id – Heap identifier
- mem – Pointer to the allocated memory
- bytes – Size of allocated memory

void heap_listener_notify_free(uintptr_t heap_id, void *mem, size_t bytes)

Notify listeners of heap free event.

Notify registered heap event listeners with matching heap identifier that memory is freed on heap

Parameters

- heap_id – Heap identifier
- mem – Pointer to the freed memory
- bytes – Size of freed memory

void heap_listener_notify_resize(uintptr_t heap_id, void *old_heap_end, void *new_heap_end)

Notify listeners of heap resize event.

Notify registered heap event listeners with matching heap identifier that the heap has been resized.

Parameters

- `heap_id` – Heap identifier
- `old_heap_end` – Address of the heap end before the change
- `new_heap_end` – Address of the heap end after the change

struct `heap_listener`

#include <heap_listener.h>

Public Members

sys_snode_t `node`

Singly linked list node.

uintptr_t `heap_id`

Identifier of the heap whose events are listened.

It can be a heap pointer, if the heap is represented as an object, or 0 in the case of the global libc heap.

enum *heap_event_types* `event`

The heap event to be notified.

3.4.2 Shared Multi Heap

The shared multi-heap memory pool manager uses the multi-heap allocator to manage a set of reserved memory regions with different capabilities / attributes (cacheable, non-cacheable, etc...).

All the different regions can be added at run-time to the shared multi-heap pool providing an opaque "attribute" value (an integer or enum value) that can be used by drivers or applications to request memory with certain capabilities.

This framework is commonly used as follow:

1. At boot time some platform code initialize the shared multi-heap framework using *shared_multi_heap_pool_init()* and add the memory regions to the pool with *shared_multi_heap_add()*, possibly gathering the needed information for the regions from the DT.
2. Each memory region encoded in a *shared_multi_heap_region* structure. This structure is also carrying an opaque and user-defined integer value that is used to define the region capabilities (for example: cacheability, cpu affinity, etc...)

```
// Init the shared multi-heap pool
shared_multi_heap_pool_init()

// Fill the struct with the data for cacheable memory
struct shared_multi_heap_region cacheable_r0 = {
     .addr = addr_r0,
     .size = size_r0,
     .attr = SMH_REG_ATTR_CACHEABLE,
};

// Add the region to the pool
```

(continues on next page)

(continued from previous page)

```
shared_multi_heap_add(&cacheable_r0, NULL);

// Add another cacheable region
struct shared_multi_heap_region cacheable_r1 = {
     .addr = addr_r1,
     .size = size_r1,
     .attr = SMH_REG_ATTR_CACHEABLE,
};

shared_multi_heap_add(&cacheable_r0, NULL);

// Add a non-cacheable region
struct shared_multi_heap_region non_cacheable_r2 = {
     .addr = addr_r2,
     .size = size_r2,
     .attr = SMH_REG_ATTR_NON_CACHEABLE,
};

shared_multi_heap_add(&non_cacheable_r2, NULL);
```

3. When a driver or application needs some dynamic memory with a certain capability, it can use *shared_multi_heap_alloc()* (or the aligned version) to request the memory by using the opaque parameter to select the correct set of attributes for the needed memory. The framework will take care of selecting the correct heap (thus memory region) to carve memory from, based on the opaque parameter and the runtime state of the heaps (available memory, heap state, etc...)

```
// Allocate 4K from cacheable memory
shared_multi_heap_alloc(SMH_REG_ATTR_CACHEABLE, 0x1000);

// Allocate 4K from non-cacheable memory
shared_multi_heap_alloc(SMH_REG_ATTR_NON_CACHEABLE, 0x1000);
```

Adding new attributes

The API does not enforce any attributes, but at least it defines the two most common ones: *SMH_REG_ATTR_CACHEABLE* and *SMH_REG_ATTR_NON_CACHEABLE*.

group **shared_multi_heap**

Shared Multi-Heap (SMH) interface.

The shared multi-heap manager uses the multi-heap allocator to manage a set of memory regions with different capabilities / attributes (cacheable, non-cacheable, etc...).

All the different regions can be added at run-time to the shared multi-heap pool providing an opaque "attribute" value (an integer or enum value) that can be used by drivers or applications to request memory with certain capabilities.

This framework is commonly used as follow:

- At boot time some platform code initialize the shared multi-heap framework using *shared_multi_heap_pool_init* and add the memory regions to the pool with *shared_multi_heap_add*, possibly gathering the needed information for the regions from the DT.
- Each memory region encoded in a *shared_multi_heap_region* structure. This structure is also carrying an opaque and user-defined integer value that is used to define the region capabilities (for example: cacheability, cpu affinity, etc...)

- When a driver or application needs some dynamic memory with a certain capability, it can use *shared_multi_heap_alloc* (or the aligned version) to request the memory by using the opaque parameter to select the correct set of attributes for the needed memory. The framework will take care of selecting the correct heap (thus memory region) to carve memory from, based on the opaque parameter and the runtime state of the heaps (available memory, heap state, etc...)

Defines

`MAX_SHARED_MULTI_HEAP_ATTR`
Maximum number of standard attributes.

Enums

enum **shared_multi_heap_attr**
SMH region attributes enumeration type.

Enumeration type for some common memory region attributes.

Values:

enumerator `SMH_REG_ATTR_CACHEABLE`
cacheable

enumerator `SMH_REG_ATTR_NON_CACHEABLE`
non-cacheable

enumerator `SMH_REG_ATTR_NUM`
must be the last item

Functions

int **shared_multi_heap_pool_init**(void)
Init the pool.

This must be the first function to be called to initialize the shared multi-heap pool. All the individual heaps must be added later with *shared_multi_heap_add*.

> **Note**
> As for the generic multi-heap allocator the expectation is that this function will be called at soc- or board-level.

Return values

- `0` – on success.
- `-EALREADY` – when the pool was already inited.
- `other` – errno codes

void *shared_multi_heap_alloc(enum *shared_multi_heap_attr* attr, size_t bytes)

Allocate memory from the memory shared multi-heap pool.

Allocates a block of memory of the specified size in bytes and with a specified capability / attribute. The opaque attribute parameter is used by the backend to select the correct heap to allocate memory from.

Parameters

- `attr` – capability / attribute requested for the memory block.
- `bytes` – requested size of the allocation in bytes.

Return values

- `ptr` – a valid pointer to heap memory.
- `err` – NULL if no memory is available.

void *shared_multi_heap_aligned_alloc(enum *shared_multi_heap_attr* attr, size_t align, size_t bytes)

Allocate aligned memory from the memory shared multi-heap pool.

Allocates a block of memory of the specified size in bytes and with a specified capability / attribute. Takes an additional parameter specifying a power of two alignment in bytes.

Parameters

- `attr` – capability / attribute requested for the memory block.
- `align` – power of two alignment for the returned pointer, in bytes.
- `bytes` – requested size of the allocation in bytes.

Return values

- `ptr` – a valid pointer to heap memory.
- `err` – NULL if no memory is available.

void shared_multi_heap_free(void *block)

Free memory from the shared multi-heap pool.

Used to free the passed block of memory that must be the return value of a previously call to *shared_multi_heap_alloc* or *shared_multi_heap_aligned_alloc*.

Parameters

- `block` – block to free, must be a pointer to a block allocated by shared_multi_heap_alloc or shared_multi_heap_aligned_alloc.

int shared_multi_heap_add(struct *shared_multi_heap_region* *region, void *user_data)

Add an heap region to the shared multi-heap pool.

This adds a shared multi-heap region to the multi-heap pool.

Parameters

- `user_data` – pointer to any data for the heap.
- `region` – pointer to the memory region to be added.

Return values

- `0` – on success.
- `-EINVAL` – when the region attribute is out-of-bound.
- `-ENOMEM` – when there are no more heaps available.
- `other` – errno codes

struct **shared_multi_heap_region**

#include <shared_multi_heap.h> SMH region struct.

This struct is carrying information about the memory region to be added in the multi-heap pool.

Public Members

uint32_t **attr**

Memory heap attribute.

uintptr_t **addr**

Memory heap starting virtual address.

size_t **size**

Memory heap size in bytes.

3.4.3 Memory Slabs

A *memory slab* is a kernel object that allows memory blocks to be dynamically allocated from a designated memory region. All memory blocks in a memory slab have a single fixed size, allowing them to be allocated and released efficiently and avoiding memory fragmentation concerns.

- *Concepts*
  - *Internal Operation*
- *Implementation*
  - *Defining a Memory Slab*
  - *Allocating a Memory Block*
  - *Releasing a Memory Block*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

Concepts

Any number of memory slabs can be defined (limited only by available RAM). Each memory slab is referenced by its memory address.

A memory slab has the following key properties:

- The **block size** of each block, measured in bytes. It must be at least 4N bytes long, where N is greater than 0.
- The **number of blocks** available for allocation. It must be greater than zero.
- A **buffer** that provides the memory for the memory slab's blocks. It must be at least "block size" times "number of blocks" bytes long.

The memory slab's buffer must be aligned to an N-byte boundary, where N is a power of 2 larger than 2 (i.e. 4, 8, 16, ...). To ensure that all memory blocks in the buffer are similarly aligned to this boundary, the block size must also be a multiple of N.

A memory slab must be initialized before it can be used. This marks all of its blocks as unused.

A thread that needs to use a memory block simply allocates it from a memory slab. When the thread finishes with a memory block, it must release the block back to the memory slab so the block can be reused.

If all the blocks are currently in use, a thread can optionally wait for one to become available. Any number of threads may wait on an empty memory slab simultaneously; when a memory block becomes available, it is given to the highest-priority thread that has waited the longest.

Unlike a heap, more than one memory slab can be defined, if needed. This allows for a memory slab with smaller blocks and others with larger-sized blocks. Alternatively, a memory pool object may be used.

Internal Operation A memory slab's buffer is an array of fixed-size blocks, with no wasted space between the blocks.

The memory slab keeps track of unallocated blocks using a linked list; the first 4 bytes of each unused block provide the necessary linkage.

Implementation

Defining a Memory Slab A memory slab is defined using a variable of type `k_mem_slab`. It must then be initialized by calling *k_mem_slab_init()*.

The following code defines and initializes a memory slab that has 6 blocks that are 400 bytes long, each of which is aligned to a 4-byte boundary.

```
struct k_mem_slab my_slab;
char __aligned(4) my_slab_buffer[6 * 400];

k_mem_slab_init(&my_slab, my_slab_buffer, 400, 6);
```

Alternatively, a memory slab can be defined and initialized at compile time by calling *K_MEM_SLAB_DEFINE*.

The following code has the same effect as the code segment above. Observe that the macro defines both the memory slab and its buffer.

```
K_MEM_SLAB_DEFINE(my_slab, 400, 6, 4);
```

Similarly, you can define a memory slab in private scope:

```
K_MEM_SLAB_DEFINE_STATIC(my_slab, 400, 6, 4);
```

Allocating a Memory Block A memory block is allocated by calling *k_mem_slab_alloc()*.

The following code builds on the example above, and waits up to 100 milliseconds for a memory block to become available, then fills it with zeroes. A warning is printed if a suitable block is not obtained.

```
char *block_ptr;

if (k_mem_slab_alloc(&my_slab, (void **)&block_ptr, K_MSEC(100)) == 0) {
    memset(block_ptr, 0, 400);
    ...
```

(continues on next page)

(continued from previous page)

```
} else {
    printf("Memory allocation time-out");
}
```

Releasing a Memory Block A memory block is released by calling *k_mem_slab_free()*.

The following code builds on the example above, and allocates a memory block, then releases it once it is no longer needed.

```
char *block_ptr;

k_mem_slab_alloc(&my_slab, (void **)&block_ptr, K_FOREVER);
... /* use memory block pointed at by block_ptr */
k_mem_slab_free(&my_slab, (void *)block_ptr);
```

Suggested Uses

Use a memory slab to allocate and free memory in fixed-size blocks.

Use memory slab blocks when sending large amounts of data from one thread to another, to avoid unnecessary copying of the data.

Configuration Options

Related configuration options:

- CONFIG_MEM_SLAB_TRACE_MAX_UTILIZATION

API Reference

group **mem_slab_apis**

Defines

K_MEM_SLAB_DEFINE(name, slab_block_size, slab_num_blocks, slab_align)

Statically define and initialize a memory slab in a public (non-static) scope.

The memory slab's buffer contains *slab_num_blocks* memory blocks that are *slab_block_size* bytes long. The buffer is aligned to a *slab_align* -byte boundary. To ensure that each memory block is similarly aligned to this boundary, *slab_block_size* must also be a multiple of *slab_align*.

The memory slab can be accessed outside the module where it is defined using:

```
extern struct k_mem_slab <name>;
```

Note

This macro cannot be used together with a static keyword. If such a use-case is desired, use *K_MEM_SLAB_DEFINE_STATIC* instead.

Parameters

- `name` – Name of the memory slab.
- `slab_block_size` – Size of each memory block (in bytes).
- `slab_num_blocks` – Number memory blocks.
- `slab_align` – Alignment of the memory slab's buffer (power of 2).

`K_MEM_SLAB_DEFINE_STATIC`(name, slab_block_size, slab_num_blocks, slab_align)

Statically define and initialize a memory slab in a private (static) scope.

The memory slab's buffer contains *slab_num_blocks* memory blocks that are *slab_block_size* bytes long. The buffer is aligned to a *slab_align* -byte boundary. To ensure that each memory block is similarly aligned to this boundary, *slab_block_size* must also be a multiple of *slab_align*.

Parameters

- `name` – Name of the memory slab.
- `slab_block_size` – Size of each memory block (in bytes).
- `slab_num_blocks` – Number memory blocks.
- `slab_align` – Alignment of the memory slab's buffer (power of 2).

Functions

int `k_mem_slab_init`(struct k_mem_slab *slab, void *buffer, size_t block_size, uint32_t num_blocks)

Initialize a memory slab.

Initializes a memory slab, prior to its first use.

The memory slab's buffer contains *slab_num_blocks* memory blocks that are *slab_block_size* bytes long. The buffer must be aligned to an N-byte boundary matching a word boundary, where N is a power of 2 (i.e. 4 on 32-bit systems, 8, 16, ...). To ensure that each memory block is similarly aligned to this boundary, *slab_block_size* must also be a multiple of N.

Parameters

- `slab` – Address of the memory slab.
- `buffer` – Pointer to buffer used for the memory blocks.
- `block_size` – Size of each memory block (in bytes).
- `num_blocks` – Number of memory blocks.

Return values

- `0` – on success
- `-EINVAL` – invalid data supplied

int `k_mem_slab_alloc`(struct k_mem_slab *slab, void **mem, *k_timeout_t* timeout)

Allocate memory from a memory slab.

This routine allocates a memory block from a memory slab.

Function properties (list may not be complete)

isr-ok

Note

timeout must be set to K_NO_WAIT if called from ISR.

Note

When CONFIG_MULTITHREADING=n any *timeout* is treated as K_NO_WAIT.

Parameters

- `slab` – Address of the memory slab.
- `mem` – Pointer to block address area.
- `timeout` – Waiting period to wait for operation to complete. Use K_NO_WAIT to return without waiting, or K_FOREVER to wait as long as necessary.

Return values

- `0` – Memory allocated. The block address area pointed at by *mem* is set to the starting address of the memory block.
- `-ENOMEM` – Returned without waiting.
- `-EAGAIN` – Waiting period timed out.
- `-EINVAL` – Invalid data supplied

void **k_mem_slab_free**(struct k_mem_slab *slab, void *mem)

Free memory allocated from a memory slab.

This routine releases a previously allocated memory block back to its associated memory slab.

Parameters

- `slab` – Address of the memory slab.
- `mem` – Pointer to the memory block (as returned by *k_mem_slab_alloc()*).

static inline uint32_t **k_mem_slab_num_used_get**(struct k_mem_slab *slab)

Get the number of used blocks in a memory slab.

This routine gets the number of memory blocks that are currently allocated in *slab*.

Parameters

- `slab` – Address of the memory slab.

Returns

Number of allocated memory blocks.

static inline uint32_t **k_mem_slab_max_used_get**(struct k_mem_slab *slab)

Get the number of maximum used blocks so far in a memory slab.

This routine gets the maximum number of memory blocks that were allocated in *slab*.

Parameters

- `slab` – Address of the memory slab.

Returns

Maximum number of allocated memory blocks.

static inline uint32_t **k_mem_slab_num_free_get**(struct k_mem_slab *slab)

Get the number of unused blocks in a memory slab.

This routine gets the number of memory blocks that are currently unallocated in *slab*.

Parameters

- **slab** – Address of the memory slab.

Returns

Number of unallocated memory blocks.

int **k_mem_slab_runtime_stats_get**(struct k_mem_slab *slab, struct sys_memory_stats *stats)

Get the memory stats for a memory slab.

This routine gets the runtime memory usage stats for the slab *slab*.

Parameters

- **slab** – Address of the memory slab
- **stats** – Pointer to memory into which to copy memory usage statistics

Return values

- **0** – Success
- **-EINVAL** – Any parameter points to NULL

int **k_mem_slab_runtime_stats_reset_max**(struct k_mem_slab *slab)

Reset the maximum memory usage for a slab.

This routine resets the maximum memory usage for the slab *slab* to its current usage.

Parameters

- **slab** – Address of the memory slab

Return values

- **0** – Success
- **-EINVAL** – Memory slab is NULL

3.4.4 Memory Blocks Allocator

The Memory Blocks Allocator allows memory blocks to be dynamically allocated from a designated memory region, where:

- All memory blocks have a single fixed size.
- Multiple blocks can be allocated or freed at the same time.
- A group of blocks allocated together may not be contiguous. This is useful for operations such as scatter-gather DMA transfers.
- Bookkeeping of allocated blocks is done outside of the associated buffer (unlike memory slab). This allows the buffer to reside in memory regions where these can be powered down to conserve energy.

- *Concepts*
  - *Internal Operation*
- *Memory Blocks Allocator*

- *Multi Memory Blocks Allocator Group*
- *Usage*
  - *Defining a Memory Blocks Allocator*
  - *Allocating Memory Blocks*
  - *Releasing a Memory Block*
  - *Using Multi Memory Blocks Allocator Group*
- *API Reference*

Concepts

Any number of Memory Blocks Allocator can be defined (limited only by available RAM). Each allocator is referenced by its memory address.

A memory blocks allocator has the following key properties:

- The **block size** of each block, measured in bytes. It must be at least 4N bytes long, where N is greater than 0.
- The **number of blocks** available for allocation. It must be greater than zero.
- A **buffer** that provides the memory for the memory slab's blocks. It must be at least "block size" times "number of blocks" bytes long.
- A **blocks bitmap** to keep track of which block has been allocated.

The buffer must be aligned to an N-byte boundary, where N is a power of 2 larger than 2 (i.e. 4, 8, 16, ...). To ensure that all memory blocks in the buffer are similarly aligned to this boundary, the block size must also be a multiple of N.

Due to the use of internal bookkeeping structures and their creation, each memory blocks allocator must be declared and defined at compile time.

Internal Operation Each buffer associated with an allocator is an array of fixed-size blocks, with no wasted space between the blocks.

The memory blocks allocator keeps track of unallocated blocks using a bitmap.

Memory Blocks Allocator

Internally, the memory blocks allocator uses a bitmap to keep track of which blocks have been allocated. Each allocator, utilizing the `sys_bitarray` interface, gets memory blocks one by one from the backing buffer up to the requested number of blocks. All the metadata about an allocator is stored outside of the backing buffer. This allows the memory region of the backing buffer to be powered down to conserve energy, as the allocator code never touches the content of the buffer.

Multi Memory Blocks Allocator Group

The Multi Memory Blocks Allocator Group utility functions provide a convenient to manage a group of allocators. A custom allocator choosing function is used to choose which allocator to use among this group.

An allocator group should be initialized at runtime via *sys_multi_mem_blocks_init()*. Each allocator can then be added via *sys_multi_mem_blocks_add_allocator()*.

To allocate memory blocks from group, *sys_multi_mem_blocks_alloc()* is called with an opaque "configuration" parameter. This parameter is passed directly to the allocator choosing function so that an appropriate allocator can be chosen. After an allocator is chosen, memory blocks are allocated via *sys_mem_blocks_alloc()*.

Allocated memory blocks can be freed via *sys_multi_mem_blocks_free()*. The caller does not need to pass a configuration parameter. The allocator code matches the passed in memory addresses to find the correct allocator and then memory blocks are freed via *sys_mem_blocks_free()*.

Usage

Defining a Memory Blocks Allocator A memory blocks allocator is defined using a variable of type *sys_mem_blocks_t*. It needs to be defined and initialized at compile time by calling *SYS_MEM_BLOCKS_DEFINE*.

The following code defines and initializes a memory blocks allocator which has 4 blocks that are 64 bytes long, each of which is aligned to a 4-byte boundary:

```
SYS_MEM_BLOCKS_DEFINE(allocator, 64, 4, 4);
```

Similarly, you can define a memory blocks allocator in private scope:

```
SYS_MEM_BLOCKS_DEFINE_STATIC(static_allocator, 64, 4, 4);
```

A pre-defined buffer can also be provided to the allocator where the buffer can be placed separately. Note that the alignment of the buffer needs to be done at its definition.

```
uint8_t __aligned(4) backing_buffer[64 * 4];
SYS_MEM_BLOCKS_DEFINE_WITH_EXT_BUF(allocator, 64, 4, backing_buffer);
```

Allocating Memory Blocks Memory blocks can be allocated by calling *sys_mem_blocks_alloc()*.

```
int ret;
uintptr_t blocks[2];

ret = sys_mem_blocks_alloc(allocator, 2, blocks);
```

If `ret == 0`, the array `blocks` will contain an array of memory addresses pointing to the allocated blocks.

Releasing a Memory Block Memory blocks are released by calling *sys_mem_blocks_free()*.

The following code builds on the example above which allocates 2 memory blocks, then releases them once they are no longer needed.

```
int ret;
uintptr_t blocks[2];

ret = sys_mem_blocks_alloc(allocator, 2, blocks);
... /* perform some operations on the allocated memory blocks */
ret = sys_mem_blocks_free(allocator, 2, blocks);
```

Using Multi Memory Blocks Allocator Group The following code demonstrates how to initialize an allocator group:

```
sys_mem_blocks_t *choice_fn(struct sys_multi_mem_blocks *group, void *cfg)
{
    ...
}

SYS_MEM_BLOCKS_DEFINE(allocator0, 64, 4, 4);
SYS_MEM_BLOCKS_DEFINE(allocator1, 64, 4, 4);

static sys_multi_mem_blocks_t alloc_group;

sys_multi_mem_blocks_init(&alloc_group, choice_fn);
sys_multi_mem_blocks_add_allocator(&alloc_group, &allocator0);
sys_multi_mem_blocks_add_allocator(&alloc_group, &allocator1);
```

To allocate and free memory blocks from the group:

```
int ret;
uintptr_t blocks[1];
size_t blk_size;

ret = sys_multi_mem_blocks_alloc(&alloc_group, UINT_TO_POINTER(0),
                                 1, blocks, &blk_size);

ret = sys_multi_mem_blocks_free(&alloc_group, 1, blocks);
```

API Reference

group `mem_blocks_apis`

Defines

`SYS_MEM_BLOCKS_DEFINE`(name, blk_sz, num_blks, buf_align)

Create a memory block object with a new backing buffer.

Parameters

- `name` – Name of the memory block object.
- `blk_sz` – Size of each memory block (in bytes).
- `num_blks` – Total number of memory blocks.
- `buf_align` – Alignment of the memory block buffer (power of 2).

`SYS_MEM_BLOCKS_DEFINE_STATIC`(name, blk_sz, num_blks, buf_align)

Create a static memory block object with a new backing buffer.

Parameters

- `name` – Name of the memory block object.
- `blk_sz` – Size of each memory block (in bytes).
- `num_blks` – Total number of memory blocks.
- `buf_align` – Alignment of the memory block buffer (power of 2).

SYS_MEM_BLOCKS_DEFINE_WITH_EXT_BUF(name, blk_sz, num_blks, buf)

Create a memory block object with a providing backing buffer.

Parameters

- **name** – Name of the memory block object.
- **blk_sz** – Size of each memory block (in bytes).
- **num_blks** – Total number of memory blocks.
- **buf** – Backing buffer of type uint8_t.

SYS_MEM_BLOCKS_DEFINE_STATIC_WITH_EXT_BUF(name, blk_sz, num_blks, buf)

Create a static memory block object with a providing backing buffer.

Parameters

- **name** – Name of the memory block object.
- **blk_sz** – Size of each memory block (in bytes).
- **num_blks** – Total number of memory blocks.
- **buf** – Backing buffer of type uint8_t.

Typedefs

typedef struct sys_mem_blocks **sys_mem_blocks_t**

Memory Blocks Allocator.

typedef struct sys_multi_mem_blocks **sys_multi_mem_blocks_t**

Multi Memory Blocks Allocator.

typedef *sys_mem_blocks_t* *(***sys_multi_mem_blocks_choice_fn_t**)(struct sys_multi_mem_blocks *group, void *cfg)

Multi memory blocks allocator choice function.

This is a user-provided functions whose responsibility is selecting a specific memory blocks allocator based on the opaque cfg value, which is specified by the user as an argument to *sys_multi_mem_blocks_alloc()*. The callback returns a pointer to the chosen allocator where the allocation is performed.

NULL may be returned, which will cause the allocation to fail and a -EINVAL reported to the calling code.

Param group
Multi memory blocks allocator structure.

Param cfg
An opaque user-provided value. It may be interpreted in any way by the application.

Return
A pointer to the chosen allocator, or NULL if none is chosen.

Functions

int **sys_mem_blocks_alloc**(*sys_mem_blocks_t* *mem_block, size_t count, void **out_blocks)

Allocate multiple memory blocks.

Allocate multiple memory blocks, and place their pointers into the output array.

Parameters

- `mem_block` – **[in]** Pointer to memory block object.
- `count` – **[in]** Number of blocks to allocate.
- `out_blocks` – **[out]** Output array to be populated by pointers to the memory blocks. It must have at least `count` elements.

Return values

- `0` – Successful
- `-EINVAL` – Invalid argument supplied.
- `-ENOMEM` – Not enough blocks for allocation.

int **sys_mem_blocks_alloc_contiguous**(*sys_mem_blocks_t* *mem_block, size_t count, void **out_block)

Allocate a contiguous set of memory blocks.

Allocate multiple memory blocks, and place their pointers into the output array.

Parameters

- `mem_block` – **[in]** Pointer to memory block object.
- `count` – **[in]** Number of blocks to allocate.
- `out_block` – **[out]** Output pointer to the start of the allocated block set

Return values

- `0` – Successful
- `-EINVAL` – Invalid argument supplied.
- `-ENOMEM` – Not enough contiguous blocks for allocation.

int **sys_mem_blocks_get**(*sys_mem_blocks_t* *mem_block, void *in_block, size_t count)

Force allocation of a specified blocks in a memory block object.

Allocate a specified blocks in a memory block object. Note: use caution when mixing sys_mem_blocks_get and sys_mem_blocks_alloc, allocation may take any of the free memory space

Parameters

- `mem_block` – **[in]** Pointer to memory block object.
- `in_block` – **[in]** Address of the first required block to allocate
- `count` – **[in]** Number of blocks to allocate.

Return values

- `0` – Successful
- `-EINVAL` – Invalid argument supplied.
- `-ENOMEM` – Some of blocks are taken and cannot be allocated

int **sys_mem_blocks_is_region_free**(*sys_mem_blocks_t* *mem_block, void *in_block, size_t count)

check if the region is free

Parameters

- `mem_block` – **[in]** Pointer to memory block object.
- `in_block` – **[in]** Address of the first block to check
- `count` – **[in]** Number of blocks to check.

Return values

- `1` – All memory blocks are free
- `0` – At least one of the memory blocks is taken

int `sys_mem_blocks_free`(*sys_mem_blocks_t* *mem_block, size_t count, void **in_blocks)

Free multiple memory blocks.

Free multiple memory blocks according to the array of memory block pointers.

Parameters

- `mem_block` – **[in]** Pointer to memory block object.
- `count` – **[in]** Number of blocks to free.
- `in_blocks` – **[in]** Input array of pointers to the memory blocks.

Return values

- `0` – Successful
- `-EINVAL` – Invalid argument supplied.
- `-EFAULT` – Invalid pointers supplied.

int `sys_mem_blocks_free_contiguous`(*sys_mem_blocks_t* *mem_block, void *block, size_t count)

Free contiguous multiple memory blocks.

Free contiguous multiple memory blocks

Parameters

- `mem_block` – **[in]** Pointer to memory block object.
- `block` – **[in]** Pointer to the first memory block
- `count` – **[in]** Number of blocks to free.

Return values

- `0` – Successful
- `-EINVAL` – Invalid argument supplied.
- `-EFAULT` – Invalid pointer supplied.

void `sys_multi_mem_blocks_init`(*sys_multi_mem_blocks_t* *group, *sys_multi_mem_blocks_choice_fn_t* choice_fn)

Initialize multi memory blocks allocator group.

Initialize a sys_multi_mem_block struct with the specified choice function. Note that individual allocator must be added later with sys_multi_mem_blocks_add_allocator.

Parameters

- `group` – Multi memory blocks allocator structure.
- `choice_fn` – A sys_multi_mem_blocks_choice_fn_t callback used to select the allocator to be used at allocation time

void **sys_multi_mem_blocks_add_allocator**(*sys_multi_mem_blocks_t* *group, *sys_mem_blocks_t* *alloc)

Add an allocator to an allocator group.

This adds a known allocator to an existing multi memory blocks allocator group.

Parameters

- **group** – Multi memory blocks allocator structure.
- **alloc** – Allocator to add

int **sys_multi_mem_blocks_alloc**(*sys_multi_mem_blocks_t* *group, void *cfg, size_t count, void **out_blocks, size_t *blk_size)

Allocate memory from multi memory blocks allocator group.

Just as for *sys_mem_blocks_alloc()*, allocates multiple blocks of memory. Takes an opaque configuration pointer passed to the choice function, which is used by integration code to choose an allocator.

Parameters

- **group** – **[in]** Multi memory blocks allocator structure.
- **cfg** – **[in]** Opaque configuration parameter, as for sys_multi_mem_blocks_choice_fn_t
- **count** – **[in]** Number of blocks to allocate
- **out_blocks** – **[out]** Output array to be populated by pointers to the memory blocks. It must have at least `count` elements.
- **blk_size** – **[out]** If not NULL, output the block size of the chosen allocator.

Return values

- **0** – Successful
- **-EINVAL** – Invalid argument supplied, or no allocator chosen.
- **-ENOMEM** – Not enough blocks for allocation.

int **sys_multi_mem_blocks_free**(*sys_multi_mem_blocks_t* *group, size_t count, void **in_blocks)

Free memory allocated from multi memory blocks allocator group.

Free previous allocated memory blocks from *sys_multi_mem_blocks_alloc()*.

Note that all blocks in `in_blocks` must be from the same allocator.

Parameters

- **group** – **[in]** Multi memory blocks allocator structure.
- **count** – **[in]** Number of blocks to free.
- **in_blocks** – **[in]** Input array of pointers to the memory blocks.

Return values

- **0** – Successful
- **-EINVAL** – Invalid argument supplied, or no allocator chosen.
- **-EFAULT** – Invalid pointer(s) supplied.

3.4.5 Demand Paging

Demand paging provides a mechanism where data is only brought into physical memory as required by current execution context. The physical memory is conceptually divided in page-sized page frames as regions to hold data.

- When the processor tries to access data and the data page exists in one of the page frames, the execution continues without any interruptions.
- When the processor tries to access the data page that does not exist in any page frames, a page fault occurs. The paging code then brings in the corresponding data page from backing store into physical memory if there is a free page frame. If there is no more free page frames, the eviction algorithm is invoked to select a data page to be paged out, thus freeing up a page frame for new data to be paged in. If this data page has been modified after it is first paged in, the data will be written back into the backing store. If no modifications is done or after written back into backing store, the data page is now considered paged out and the corresponding page frame is now free. The paging code then invokes the backing store to page in the data page corresponding to the location of the requested data. The backing store copies that data page into the free page frame. Now the data page is in physical memory and execution can continue.

There are functions where paging in and out can be invoked manually using *k_mem_page_in()* and *k_mem_page_out()*. *k_mem_page_in()* can be used to page in data pages in anticipation that they are required in the near future. This is used to minimize number of page faults as these data pages are already in physical memory, and thus minimizing latency. *k_mem_page_out()* can be used to page out data pages where they are not going to be accessed for a considerable amount of time. This frees up page frames so that the next page in can be executed faster as the paging code does not need to invoke the eviction algorithm.

Terminology

Data Page
A data page is a page-sized region of data. It may exist in a page frame, or be paged out to some backing store. Its location can always be looked up in the CPU's page tables (or equivalent) by virtual address. The data type will always be `void *` or in some cases `uint8_t *` when doing pointer arithmetic.

Page Frame
A page frame is a page-sized physical memory region in RAM. It is a container where a data page may be placed. It is always referred to by physical address. Zephyr has a convention of using `uintptr_t` for physical addresses. For every page frame, a `struct k_mem_page_frame` is instantiated to store metadata. Flags for each page frame:

- `K_MEM_PAGE_FRAME_FREE` indicates a page frame is unused and on the list of free page frames. When this flag is set, none of the other flags are meaningful and they must not be modified.
- `K_MEM_PAGE_FRAME_PINNED` indicates a page frame is pinned in memory and should never be paged out.
- `K_MEM_PAGE_FRAME_RESERVED` indicates a physical page reserved by hardware and should not be used at all.
- `K_MEM_PAGE_FRAME_MAPPED` is set when a physical page is mapped to virtual memory address.
- `K_MEM_PAGE_FRAME_BUSY` indicates a page frame is currently involved in a page-in/out operation.
- `K_MEM_PAGE_FRAME_BACKED` indicates a page frame has a clean copy in the backing store.

K_MEM_SCRATCH_PAGE
: The virtual address of a special page provided to the backing store to: * Copy a data page from k_MEM_SCRATCH_PAGE to the specified location; or, * Copy a data page from the provided location to K_MEM_SCRATCH_PAGE. This is used as an intermediate page for page in/out operations. This scratch needs to be mapped read/write for backing store code to access. However the data page itself may only be mapped as read-only in virtual address space. If this page is provided as-is to backing store, the data page must be re-mapped as read/write which has security implications as the data page is no longer read-only to other parts of the application.

Paging Statistics

Paging statistics can be obtained via various function calls when CONFIG_DEMAND_PAGING_TIMING_HISTOGRAM_NUM_BINS is enabled:

- Overall statistics via *k_mem_paging_stats_get()*
- Per-thread statistics via *k_mem_paging_thread_stats_get()* if CONFIG_DEMAND_PAGING_THREAD_STATS is enabled
- Execution time histogram can be obtained when CONFIG_DEMAND_PAGING_TIMING_HISTOGRAM is enabled, and CONFIG_DEMAND_PAGING_TIMING_HISTOGRAM_NUM_BINS is defined. Note that the timing is highly dependent on the architecture, SoC or board. It is highly recommended that k_mem_paging_eviction_histogram_bounds[] and k_mem_paging_backing_store_histogram_bounds[] be defined for a particular application.
  - Execution time histogram of eviction algorithm via *k_mem_paging_histogram_eviction_get()*
  - Execution time histogram of backing store doing page-in via *k_mem_paging_histogram_backing_store_page_in_get()*
  - Execution time histogram of backing store doing page-out via *k_mem_paging_histogram_backing_store_page_out_get()*

Eviction Algorithm

The eviction algorithm is used to determine which data page and its corresponding page frame can be paged out to free up a page frame for the next page in operation. There are two functions which are called from the kernel paging code:

- *k_mem_paging_eviction_init()* is called to initialize the eviction algorithm. This is called at POST_KERNEL.
- *k_mem_paging_eviction_select()* is called to select a data page to evict. A function argument dirty is written to signal the caller whether the selected data page has been modified since it is first paged in. If the dirty bit is returned as set, the paging code signals to the backing store to write the data page back into storage (thus updating its content). The function returns a pointer to the page frame corresponding to the selected data page.

Currently, a NRU (Not-Recently-Used) eviction algorithm has been implemented as a sample. This is a very simple algorithm which ranks each data page on whether they have been accessed and modified. The selection is based on this ranking.

To implement a new eviction algorithm, the two functions mentioned above must be implemented.

Backing Store

Backing store is responsible for paging in/out data page between their corresponding page frames and storage. These are the functions which must be implemented:

- *k_mem_paging_backing_store_init()* is called to initialized the backing store at POST_KERNEL.
- *k_mem_paging_backing_store_location_get()* is called to reserve a backing store location so a data page can be paged out. This location token is passed to *k_mem_paging_backing_store_page_out()* to perform actual page out operation.
- *k_mem_paging_backing_store_location_free()* is called to free a backing store location (the location token) which can then be used for subsequent page out operation.
- *k_mem_paging_backing_store_page_in()* copies a data page from the backing store location associated with the provided location token to the page pointed by K_MEM_SCRATCH_PAGE.
- *k_mem_paging_backing_store_page_out()* copies a data page from K_MEM_SCRATCH_PAGE to the backing store location associated with the provided location token.
- *k_mem_paging_backing_store_page_finalize()* is invoked after *k_mem_paging_backing_store_page_in()* so that the page frame struct may be updated for internal accounting. This can be a no-op.

To implement a new backing store, the functions mentioned above must be implemented. *k_mem_paging_backing_store_page_finalize()* can be an empty function if so desired.

API Reference

group mem-demand-paging

Functions

int **k_mem_page_out**(void *addr, size_t size)

Evict a page-aligned virtual memory region to the backing store.

Useful if it is known that a memory region will not be used for some time. All the data pages within the specified region will be evicted to the backing store if they weren't already, with their associated page frames marked as available for mappings or page-ins.

None of the associated page frames mapped to the provided region should be pinned.

Note that there are no guarantees how long these pages will be evicted, they could take page faults immediately.

If CONFIG_DEMAND_PAGING_ALLOW_IRQ is enabled, this function may not be called by ISRs as the backing store may be in-use.

Parameters

- **addr** – Base page-aligned virtual address
- **size** – Page-aligned data region size

Return values

- **0** – Success
- **-ENOMEM** – Insufficient space in backing store to satisfy request. The region may be partially paged out.

void **k_mem_page_in**(void *addr, size_t size)

Load a virtual data region into memory.

After the function completes, all the page frames associated with this function will be paged in. However, they are not guaranteed to stay there. This is useful if the region is known to be used soon.

If CONFIG_DEMAND_PAGING_ALLOW_IRQ is enabled, this function may not be called by ISRs as the backing store may be in-use.

Parameters

- **addr** – Base page-aligned virtual address
- **size** – Page-aligned data region size

void **k_mem_pin**(void *addr, size_t size)

Pin an aligned virtual data region, paging in as necessary.

After the function completes, all the page frames associated with this region will be resident in memory and pinned such that they stay that way. This is a stronger version of z_mem_page_in().

If CONFIG_DEMAND_PAGING_ALLOW_IRQ is enabled, this function may not be called by ISRs as the backing store may be in-use.

Parameters

- **addr** – Base page-aligned virtual address
- **size** – Page-aligned data region size

void **k_mem_unpin**(void *addr, size_t size)

Un-pin an aligned virtual data region.

After the function completes, all the page frames associated with this region will be no longer marked as pinned. This does not evict the region, follow this with z_mem_page_out() if you need that.

Parameters

- **addr** – Base page-aligned virtual address
- **size** – Page-aligned data region size

void **k_mem_paging_stats_get**(struct *k_mem_paging_stats_t* *stats)

Get the paging statistics since system startup.

This populates the paging statistics struct being passed in as argument.

Parameters

- **stats** – **[inout]** Paging statistics struct to be filled.

void **k_mem_paging_thread_stats_get**(struct *k_thread* *thread, struct *k_mem_paging_stats_t* *stats)

Get the paging statistics since system startup for a thread.

This populates the paging statistics struct being passed in as argument for a particular thread.

Parameters

- **thread** – **[in]** Thread
- **stats** – **[inout]** Paging statistics struct to be filled.

void **k_mem_paging_histogram_eviction_get**(struct *k_mem_paging_histogram_t* *hist)

Get the eviction timing histogram.

This populates the timing histogram struct being passed in as argument.

Parameters

- **hist** – **[inout]** Timing histogram struct to be filled.

void **k_mem_paging_histogram_backing_store_page_in_get**(struct *k_mem_paging_histogram_t* *hist)

Get the backing store page-in timing histogram.

This populates the timing histogram struct being passed in as argument.

Parameters

- **hist** – **[inout]** Timing histogram struct to be filled.

void **k_mem_paging_histogram_backing_store_page_out_get**(struct *k_mem_paging_histogram_t* *hist)

Get the backing store page-out timing histogram.

This populates the timing histogram struct being passed in as argument.

Parameters

- **hist** – **[inout]** Timing histogram struct to be filled.

struct **k_mem_paging_stats_t**

#include <demand_paging.h> Paging Statistics.

Public Members

unsigned long **cnt**

Number of page faults.

unsigned long **irq_locked**

Number of page faults with IRQ locked.

unsigned long **irq_unlocked**

Number of page faults with IRQ unlocked.

unsigned long **in_isr**

Number of page faults while in ISR.

unsigned long **clean**

Number of clean pages selected for eviction.

unsigned long **dirty**

Number of dirty pages selected for eviction.

struct **k_mem_paging_histogram_t**

#include <demand_paging.h> Paging Statistics Histograms.

Eviction Algorithm APIs

group `mem-demand-paging-eviction`

Eviction algorithm APIs.

Functions

void **k_mem_paging_eviction_add**(struct k_mem_page_frame *pf)

Submit a page frame for eviction candidate tracking.

The kernel will invoke this to tell the eviction algorithm the provided page frame may be considered as a potential eviction candidate.

This function will never be called before the initial *k_mem_paging_eviction_init()*.

This function is invoked with interrupts locked.

Parameters

- `pf` – **[in]** The page frame to add

void **k_mem_paging_eviction_remove**(struct k_mem_page_frame *pf)

Remove a page frame from potential eviction candidates.

The kernel will invoke this to tell the eviction algorithm the provided page frame may no longer be considered as a potential eviction candidate.

This function will only be called with page frames that were submitted using *k_mem_paging_eviction_add()* beforehand.

This function is invoked with interrupts locked.

Parameters

- `pf` – **[in]** The page frame to remove

void **k_mem_paging_eviction_accessed**(uintptr_t phys)

Process a page frame as being newly accessed.

The architecture-specific memory fault handler will invoke this to tell the eviction algorithm the provided physical address belongs to a page frame being accessed and such page frame should become unlikely to be considered as the next eviction candidate.

This function is invoked with interrupts locked.

Parameters

- `phys` – **[in]** The physical address being accessed

struct k_mem_page_frame ***k_mem_paging_eviction_select**(bool *dirty)

Select a page frame for eviction.

The kernel will invoke this to choose a page frame to evict if there are no free page frames. It is not guaranteed that the returned page frame will actually be evicted. If it is then the kernel will call *k_mem_paging_eviction_remove()* with it.

This function will never be called before the initial *k_mem_paging_eviction_init()*.

This function is invoked with interrupts locked.

Parameters

- `dirty` – **[out]** Whether the page to evict is dirty

Returns

The page frame to evict

void k_mem_paging_eviction_init(void)

Initialization function.

Called at POST_KERNEL to perform any necessary initialization tasks for the eviction algorithm. *k_mem_paging_eviction_select()* is guaranteed to never be called until this has returned, and this will only be called once.

Backing Store APIs

group mem-demand-paging-backing-store

Backing store APIs.

Functions

int k_mem_paging_backing_store_location_get(struct k_mem_page_frame *pf, uintptr_t *location, bool page_fault)

Reserve or fetch a storage location for a data page loaded into a page frame.

The returned location token must be unique to the mapped virtual address. This location will be used in the backing store to page out data page contents for later retrieval. The location value must be page-aligned.

This function may be called multiple times on the same data page. If its page frame has its K_MEM_PAGE_FRAME_BACKED bit set, it is expected to return the previous backing store location for the data page containing a cached clean copy. This clean copy may be updated on page-out, or used to discard clean pages without needing to write out their contents.

If the backing store is full, some other backing store location which caches a loaded data page may be selected, in which case its associated page frame will have the K_MEM_PAGE_FRAME_BACKED bit cleared (as it is no longer cached).

k_mem_page_frame_to_virt(pf) will indicate the virtual address the page is currently mapped to. Large, sparse backing stores which can contain the entire address space may simply generate location tokens purely as a function of that virtual address with no other management necessary.

This function distinguishes whether it was called on behalf of a page fault. A free backing store location must always be reserved in order for page faults to succeed. If the page_fault parameter is not set, this function should return -ENOMEM even if one location is available.

This function is invoked with interrupts locked.

Parameters

- pf – Virtual address to obtain a storage location
- location – **[out]** storage location token
- page_fault – Whether this request was for a page fault

Returns
0 Success

Returns
-ENOMEM Backing store is full

void k_mem_paging_backing_store_location_free(uintptr_t location)

Free a backing store location.

Any stored data may be discarded, and the location token associated with this address may be re-used for some other data page.

This function is invoked with interrupts locked.

Parameters

- `location` – Location token to free

void `k_mem_paging_backing_store_page_out`(uintptr_t location)

Copy a data page from K_MEM_SCRATCH_PAGE to the specified location.

Immediately before this is called, K_MEM_SCRATCH_PAGE will be mapped read-write to the intended source page frame for the calling context.

Calls to this and *k_mem_paging_backing_store_page_in()* will always be serialized, but interrupts may be enabled.

Parameters

- `location` – Location token for the data page, for later retrieval

void `k_mem_paging_backing_store_page_in`(uintptr_t location)

Copy a data page from the provided location to K_MEM_SCRATCH_PAGE.

Immediately before this is called, K_MEM_SCRATCH_PAGE will be mapped read-write to the intended destination page frame for the calling context.

Calls to this and *k_mem_paging_backing_store_page_out()* will always be serialized, but interrupts may be enabled.

Parameters

- `location` – Location token for the data page

void `k_mem_paging_backing_store_page_finalize`(struct k_mem_page_frame *pf,
uintptr_t location)

Update internal accounting after a page-in.

This is invoked after *k_mem_paging_backing_store_page_in()* and interrupts have been* re-locked, making it safe to access the k_mem_page_frame data. The location value will be the same passed to *k_mem_paging_backing_store_page_in()*.

The primary use-case for this is to update custom fields for the backing store in the page frame, to reflect where the data should be evicted to if it is paged out again. This may be a no-op in some implementations.

If the backing store caches paged-in data pages, this is the appropriate time to set the K_MEM_PAGE_FRAME_BACKED bit. The kernel only skips paging out clean data pages if they are noted as clean in the page tables and the K_MEM_PAGE_FRAME_BACKED bit is set in their associated page frame.

Parameters

- `pf` – Page frame that was loaded in
- `location` – Location of where the loaded data page was retrieved

void `k_mem_paging_backing_store_init`(void)

Backing store initialization function.

The implementation may expect to receive page in/out calls as soon as this returns, but not before that. Called at POST_KERNEL.

This function is expected to do two things:

- Initialize any internal data structures and accounting for the backing store.
- If the backing store already contains all or some loaded kernel data pages at boot time, K_MEM_PAGE_FRAME_BACKED should be appropriately set for their associated page frames, and any internal accounting set up appropriately.

3.4.6 Virtual Memory

Virtual memory (VM) in Zephyr provides developers with the ability to fine tune access to memory. To utilize virtual memory, the platform must support Memory Management Unit (MMU) and it must be enabled in the build. Due to the target of Zephyr mainly being embedded systems, virtual memory support in Zephyr differs a bit from that in traditional operating systems:

Mapping of Kernel Image
Default is to do 1:1 mapping for the kernel image (including code and data) between physical and virtual memory address spaces, if demand paging is not enabled. Deviation from this requires careful manipulation of linker script.

Secondary Storage
Basic virtual memory support does not utilize secondary storage to extend usable memory. The maximum usable memory is the same as the physical memory.

- *Demand Paging* enables utilizing secondary storage as a backing store for virtual memory, thus allowing larger usable memory than the available physical memory. Note that demand paging needs to be explicitly enabled.
- Although the virtual memory space can be larger than physical memory space, without enabling demand paging, all virtually mapped memory must be backed by physical memory.

Kconfigs

Required These are the Kconfigs that need to be enabled or defined for kernel to support virtual memory.

- `CONFIG_MMU`: must be enabled for virtual memory support in kernel.
- `CONFIG_MMU_PAGE_SIZE`: size of a memory page. Default is 4KB.
- `CONFIG_KERNEL_VM_BASE`: base address of virtual address space.
- `CONFIG_KERNEL_VM_SIZE`: size of virtual address space. Default is 8MB.
- `CONFIG_KERNEL_VM_OFFSET`: kernel image starts at this offset from `CONFIG_KERNEL_VM_BASE`.

Optional

- `CONFIG_KERNEL_DIRECT_MAP`: permits 1:1 mappings between virtual and physical addresses, instead of kernel choosing addresses within the virtual address space. This is useful for mapping device MMIO regions for more precise access control.

Memory Map Overview

This is an overview of the memory map of the virtual memory address space. Note that the `Z_*` macros, which are used in code, may have different meanings depending on architecture and Kconfigs, which will be explained below.

```
+--------------+ <- K_MEM_VIRT_RAM_START
| Undefined VM | <- architecture specific reserved area
+--------------+ <- K_MEM_KERNEL_VIRT_START
| Mapping for  |
| main kernel  |
| image        |
|              |
|              |
+--------------+ <- K_MEM_VM_FREE_START
```

(continues on next page)

(continued from previous page)

```
|              |
| Unused,      |
| Available VM |
|              |
|..............| <- grows downward as more mappings are made
| Mapping      |
+--------------+
| Mapping      |
+--------------+
| ...          |
+--------------+
| Mapping      |
+--------------+ <- memory mappings start here
| Reserved     | <- special purpose virtual page(s) of size K_MEM_VM_RESERVED
+--------------+ <- K_MEM_VIRT_RAM_END
```

- `K_MEM_VIRT_RAM_START` is the beginning of the virtual memory address space. This needs to be page aligned. Currently, it is the same as `CONFIG_KERNEL_VM_BASE`.
- `K_MEM_VIRT_RAM_SIZE` is the size of the virtual memory address space. This needs to be page aligned. Currently, it is the same as `CONFIG_KERNEL_VM_SIZE`.
- `K_MEM_VIRT_RAM_END` is simply (`K_MEM_VIRT_RAM_START` + `K_MEM_VIRT_RAM_SIZE`).
- `K_MEM_KERNEL_VIRT_START` is the same as `z_mapped_start` specified in the linker script. This is the virtual address of the beginning of the kernel image at boot time.
- `K_MEM_KERNEL_VIRT_END` is the same as `z_mapped_end` specified in the linker script. This is the virtual address of the end of the kernel image at boot time.
- `K_MEM_VM_FREE_START` is the beginning of the virtual address space where addresses can be allocated for memory mapping. This depends on whether `CONFIG_ARCH_MAPS_ALL_RAM` is enabled.
  - If it is enabled, which means all physical memory are mapped in virtual memory address space, and it is the same as (`CONFIG_SRAM_BASE_ADDRESS` + `CONFIG_SRAM_SIZE`).
  - If it is disabled, `K_MEM_VM_FREE_START` is the same `K_MEM_KERNEL_VIRT_END` which is the end of the kernel image.
- `K_MEM_VM_RESERVED` is an area reserved to support kernel functions. For example, some addresses are reserved to support demand paging.

Virtual Memory Mappings

Setting up Mappings at Boot In general, most supported architectures set up the memory mappings at boot as following:

- `.text` section is read-only and executable. It is accessible in both kernel and user modes.
- `.rodata` section is read-only and non-executable. It is accessible in both kernel and user modes.
- Other kernel sections, such as `.data`, `.bss` and `.noinit`, are read-write and non-executable. They are only accessible in kernel mode.
  - Stacks for user mode threads are automatically granted read-write access to their corresponding user mode threads during thread creation.
  - Global variables, by default, are not accessible to user mode threads. Refer to *Memory Domains and Partitions* on how to use global variables in user mode threads, and on how to share data between user mode threads.

Caching modes for these mappings are architecture specific. They can be none, write-back, or write-through.

Note that SoCs have their own additional mappings required to boot where these mappings are defined under their own SoC configurations. These mappings usually include device MMIO regions needed to setup the hardware.

Mapping Anonymous Memory The unused physical memory can be mapped in virtual address space on demand. This is conceptually similar to memory allocation from heap, but these mappings must be aligned on page size and have finer access control.

- *k_mem_map()* can be used to map unused physical memory:
  - The requested size must be multiple of page size.
  - The address returned is inside the virtual address space between K_MEM_VM_FREE_START and K_MEM_VIRT_RAM_END.
  - The mapped region is not guaranteed to be physically contiguous in memory.
  - Guard pages immediately before and after the mapped virtual region are automatically allocated to catch access issue due to buffer underrun or overrun.
- The mapped region can be unmapped (i.e. freed) via *k_mem_unmap()*:
  - Caution must be exercised to give the pass the same region size to both *k_mem_map()* and *k_mem_unmap()*. The unmapping function does not check if it is a valid mapped region before unmapping.

API Reference

group kernel_memory_management

Kernel Memory Management.

Caching mode definitions.

These are mutually exclusive.

K_MEM_CACHE_NONE

No caching.

Most drivers want this.

K_MEM_CACHE_WT

Write-through caching.

Used by certain drivers.

K_MEM_CACHE_WB

Full write-back caching.

Any RAM mapped wants this.

K_MEM_CACHE_MASK

Reserved bits for cache modes in k_map() flags argument.

Region permission attributes.

Default is read-only, no user, no exec

K_MEM_PERM_RW
: Region will have read/write access (and not read-only)

K_MEM_PERM_EXEC
: Region will be executable (normally forbidden)

K_MEM_PERM_USER
: Region will be accessible to user mode (normally supervisor-only)

Region mapping behaviour attributes

K_MEM_DIRECT_MAP
: Region will be mapped to 1:1 virtual and physical address.

k_mem_map() control flags

K_MEM_MAP_UNINIT
: The mapped region is not guaranteed to be zeroed.

  This may improve performance. The associated page frames may contain indeterminate data, zeroes, or even sensitive information.

  This may not be used with K_MEM_PERM_USER as there are no circumstances where this is safe.

K_MEM_MAP_LOCK
: Region will be pinned in memory and never paged.

  Such memory is guaranteed to never produce a page fault due to page-outs or copy-on-write once the mapping call has returned. Physical page frames will be pre-fetched as necessary and pinned.

Functions

size_t **k_mem_free_get**(void)
: Return the amount of free memory available.

  The returned value will reflect how many free RAM page frames are available. If demand paging is enabled, it may still be possible to allocate more.

  The information reported by this function may go stale immediately if concurrent memory mappings or page-ins take place.

  Returns
  Free physical RAM, in bytes

static inline void *k_mem_map(size_t size, uint32_t flags)

Map anonymous memory into Zephyr's address space.

This function effectively increases the data space available to Zephyr. The kernel will choose a base virtual address and return it to the caller. The memory will have access permissions for all contexts set per the provided flags argument.

If user thread access control needs to be managed in any way, do not enable K_MEM_PERM_USER flags here; instead manage the region's permissions with memory domain APIs after the mapping has been established. Setting K_MEM_PERM_USER here will allow all user threads to access this memory which is usually undesirable.

Unless K_MEM_MAP_UNINIT is used, the returned memory will be zeroed.

The mapped region is not guaranteed to be physically contiguous in memory. Physically contiguous buffers should be allocated statically and pinned at build time.

Pages mapped in this way have write-back cache settings.

The returned virtual memory pointer will be page-aligned. The size parameter, and any base address for re-mapping purposes must be page- aligned.

Note that the allocation includes two guard pages immediately before and after the requested region. The total size of the allocation will be the requested size plus the size of these two guard pages.

Many K_MEM_MAP_* flags have been implemented to alter the behavior of this function, with details in the documentation for these flags.

Parameters

- `size` – Size of the memory mapping. This must be page-aligned.
- `flags` – K_MEM_PERM_*, K_MEM_MAP_* control flags.

Returns

The mapped memory location, or NULL if insufficient virtual address space, insufficient physical memory to establish the mapping, or insufficient memory for paging structures.

static inline void k_mem_unmap(void *addr, size_t size)

Un-map mapped memory.

This removes a memory mapping for the provided page-aligned region. Associated page frames will be free and the kernel may re-use the associated virtual address region. Any paged out data pages may be discarded.

Calling this function on a region which was not mapped to begin with is undefined behavior.

Parameters

- `addr` – Page-aligned memory region base virtual address
- `size` – Page-aligned memory region size

size_t k_mem_region_align(uintptr_t *aligned_addr, size_t *aligned_size, uintptr_t addr, size_t size, size_t align)

Given an arbitrary region, provide a aligned region that covers it.

The returned region will have both its base address and size aligned to the provided alignment value.

Parameters

- `aligned_addr` – **[out]** Aligned address
- `aligned_size` – **[out]** Aligned region size

- `addr` – **[in]** Region base address
- `size` – **[in]** Region size
- `align` – **[in]** What to align the address and size to

Return values
`offset` – between aligned_addr and addr

3.5 Data Structures

Zephyr provides a library of common general purpose data structures used within the kernel, but useful by application code in general. These include list and balanced tree structures for storing ordered data, and a ring buffer for managing "byte stream" data in a clean way.

Note that in general, the collections are implemented as "intrusive" data structures. The "node" data is the only struct used by the library code, and it does not store a pointer or other metadata to indicate what user data is "owned" by that node. Instead, the expectation is that the node will be itself embedded within a user-defined struct. Macros are provided to retrieve a user struct address from the embedded node pointer in a clean way. The purpose behind this design is to allow the collections to be used in contexts where dynamic allocation is disallowed (i.e. there is no need to allocate node objects because the memory is provided by the user).

Note also that these libraries are uniformly unsynchronized; access to them is not threadsafe by default. These are data structures, not synchronization primitives. The expectation is that any locking needed will be provided by the user.

3.5.1 Single-linked List

Zephyr provides a *sys_slist_t* type for storing simple singly-linked list data (i.e. data where each list element stores a pointer to the next element, but not the previous one). This supports constant-time access to the first (head) and last (tail) elements of the list, insertion before the head and after the tail of the list and constant time removal of the head. Removal of subsequent nodes requires access to the "previous" pointer and thus can only be performed in linear time by searching the list.

The *sys_slist_t* struct may be instantiated by the user in any accessible memory. It should be initialized with either *sys_slist_init()* or by static assignment from SYS_SLIST_STATIC_INIT before use. Its interior fields are opaque and should not be accessed by user code.

The end nodes of a list may be retrieved with *sys_slist_peek_head()* and *sys_slist_peek_tail()*, which will return NULL if the list is empty, otherwise a pointer to a *sys_snode_t* struct.

The *sys_snode_t* struct represents the data to be inserted. In general, it is expected to be allocated/controlled by the user, usually embedded within a struct which is to be added to the list. The container struct pointer may be retrieved from a list node using *SYS_SLIST_CONTAINER*, passing it the struct name of the containing struct and the field name of the node. Internally, the *sys_snode_t* struct contains only a next pointer, which may be accessed with *sys_slist_peek_next()*.

Lists may be modified by adding a single node at the head or tail with *sys_slist_prepend()* and *sys_slist_append()*. They may also have a node added to an interior point with *sys_slist_insert()*, which inserts a new node after an existing one. Similarly *sys_slist_remove()* will remove a node given a pointer to its predecessor. These operations are all constant time.

Convenience routines exist for more complicated modifications to a list. *sys_slist_merge_slist()* will append an entire list to an existing one. *sys_slist_append_list()* will append a bounded subset of an existing list in constant

time. And *sys_slist_find_and_remove()* will search a list (in linear time) for a given node and remove it if present.

Finally the slist implementation provides a set of "for each" macros that allows for iterating over a list in a natural way without needing to manually traverse the next pointers. *SYS_SLIST_FOR_EACH_NODE* will enumerate every node in a list given a local variable to store the node pointer. *SYS_SLIST_FOR_EACH_NODE_SAFE* behaves similarly, but has a more complicated implementation that requires an extra scratch variable for storage and allows the user to delete the iterated node during the iteration. Each of those macros also exists in a "container" variant (*SYS_SLIST_FOR_EACH_CONTAINER* and *SYS_SLIST_FOR_EACH_CONTAINER_SAFE*) which assigns a local variable of a type that matches the user's container struct and not the node struct, performing the required offsets internally. And *SYS_SLIST_ITERATE_FROM_NODE* exists to allow for enumerating a node and all its successors only, without inspecting the earlier part of the list.

Single-linked List Internals

The slist code is designed to be minimal and conventional. Internally, a *sys_slist_t* struct is nothing more than a pair of "head" and "tail" pointer fields. And a *sys_snode_t* stores only a single "next" pointer.

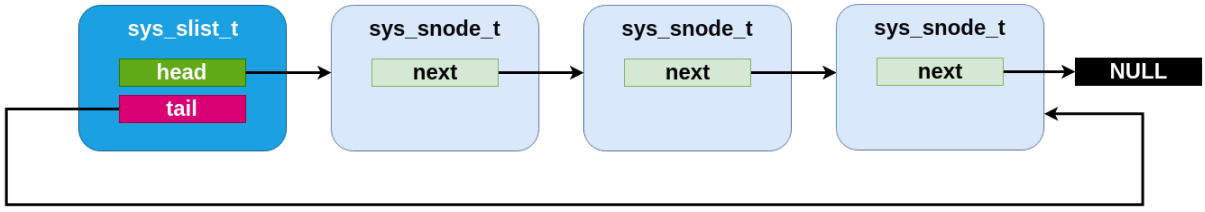


Fig. 4: An slist containing three elements.

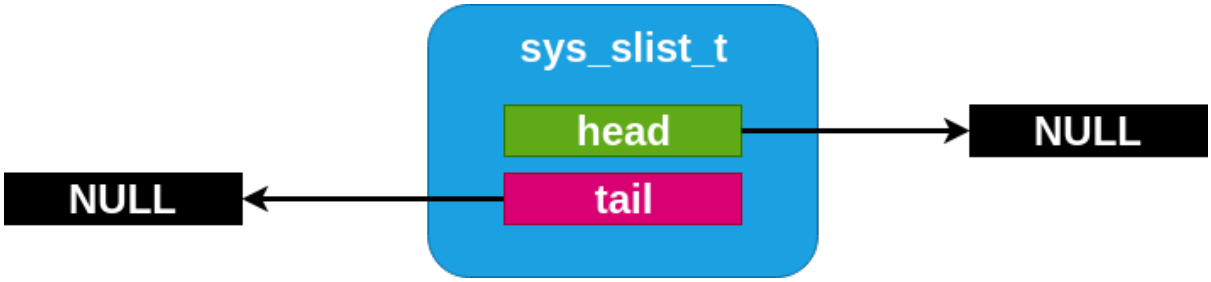


Fig. 5: An empty slist

The specific implementation of the list code, however, is done with an internal "Z_GENLIST" template API which allows for extracting those fields from arbitrary structures and emits an arbitrarily named set of functions. This allows for implementing more complicated single-linked list variants using the same basic primitives. The genlist implementor is responsible for a custom implementation of the primitive operations only: an "init" step for each struct, and a "get" and "set" primitives for each of head, tail and next pointers on their relevant structs. These inline functions are passed as parameters to the genlist macro expansion.

Only one such variant, sflist, exists in Zephyr at the moment.

Flagged List

The *sys_sflist_t* is implemented using the described genlist template API. With the exception of symbol naming ("sflist" instead of "slist") and the additional API described next, it operates in all ways identically to the slist API.

It adds the ability to associate exactly two bits of user defined "flags" with each list node. These can be accessed and modified with *sys_sfnode_flags_get()* and *sys_sfnode_flags_set()*. Internally, the flags are stored unioned with the bottom bits of the next pointer and incur no SRAM storage overhead when compared with the simpler slist code.

Single-linked List API Reference

group `single-linked-list_apis`

Single-linked list implementation.

Single-linked list implementation using inline macros/functions. This API is not thread safe, and thus if a list is used across threads, calls to functions must be protected with synchronization primitives.

Defines

`SYS_SLIST_FOR_EACH_NODE`(__sl, __sn)

Provide the primitive to iterate on a list Note: the loop is unsafe and thus __sn should not be removed.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_SLIST_FOR_EACH_NODE(l, n) {
    <user code>
}
```

This and other SYS_SLIST_*() macros are not thread safe.

Parameters

- **__sl** – A pointer on a sys_slist_t to iterate on
- **__sn** – A sys_snode_t pointer to peek each node of the list

`SYS_SLIST_ITERATE_FROM_NODE`(__sl, __sn)

Provide the primitive to iterate on a list, from a node in the list Note: the loop is unsafe and thus __sn should not be removed.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_SLIST_ITERATE_FROM_NODE(l, n) {
    <user code>
}
```

Like *SYS_SLIST_FOR_EACH_NODE()*, but __dn already contains a node in the list where to start searching for the next entry from. If NULL, it starts from the head.

This and other SYS_SLIST_*() macros are not thread safe.

Parameters

- **__sl** – A pointer on a sys_slist_t to iterate on
- **__sn** – A sys_snode_t pointer to peek each node of the list it contains the starting node, or NULL to start from the head

`SYS_SLIST_FOR_EACH_NODE_SAFE`(__sl, __sn, __sns)

Provide the primitive to safely iterate on a list Note: __sn can be removed, it will not break the loop.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_SLIST_FOR_EACH_NODE_SAFE(l, n, s) {
    <user code>
}
```

This and other SYS_SLIST_*() macros are not thread safe.

Parameters

- **__sl** – A pointer on a sys_slist_t to iterate on
- **__sn** – A sys_snode_t pointer to peek each node of the list
- **__sns** – A sys_snode_t pointer for the loop to run safely

SYS_SLIST_CONTAINER(__ln, __cn, __n)

Provide the primitive to resolve the container of a list node Note: it is safe to use with NULL pointer nodes.

Parameters

- **__ln** – A pointer on a sys_node_t to get its container
- **__cn** – Container struct type pointer
- **__n** – The field name of sys_node_t within the container struct

SYS_SLIST_PEEK_HEAD_CONTAINER(__sl, __cn, __n)

Provide the primitive to peek container of the list head.

Parameters

- **__sl** – A pointer on a sys_slist_t to peek
- **__cn** – Container struct type pointer
- **__n** – The field name of sys_node_t within the container struct

SYS_SLIST_PEEK_TAIL_CONTAINER(__sl, __cn, __n)

Provide the primitive to peek container of the list tail.

Parameters

- **__sl** – A pointer on a sys_slist_t to peek
- **__cn** – Container struct type pointer
- **__n** – The field name of sys_node_t within the container struct

SYS_SLIST_PEEK_NEXT_CONTAINER(__cn, __n)

Provide the primitive to peek the next container.

Parameters

- **__cn** – Container struct type pointer
- **__n** – The field name of sys_node_t within the container struct

SYS_SLIST_FOR_EACH_CONTAINER(__sl, __cn, __n)

Provide the primitive to iterate on a list under a container Note: the loop is unsafe and thus __cn should not be detached.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_SLIST_FOR_EACH_CONTAINER(l, c, n) {
    <user code>
}
```

Parameters

- **__sl** – A pointer on a sys_slist_t to iterate on

- **__cn** – A pointer to peek each entry of the list
- **__n** – The field name of sys_node_t within the container struct

SYS_SLIST_FOR_EACH_CONTAINER_SAFE(__sl, __cn, __cns, __n)

Provide the primitive to safely iterate on a list under a container Note: __cn can be detached, it will not break the loop.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_SLIST_FOR_EACH_NODE_SAFE(l, c, cn, n) {
    <user code>
}
```

Parameters

- **__sl** – A pointer on a sys_slist_t to iterate on
- **__cn** – A pointer to peek each entry of the list
- **__cns** – A pointer for the loop to run safely
- **__n** – The field name of sys_node_t within the container struct

SYS_SLIST_STATIC_INIT(ptr_to_list)

Statically initialize a single-linked list.

Parameters

- **ptr_to_list** – A pointer on the list to initialize

Typedefs

typedef struct _snode **sys_snode_t**

Single-linked list node structure.

typedef struct _slist **sys_slist_t**

Single-linked list structure.

Functions

static inline void **sys_slist_init**(*sys_slist_t* *list)

Initialize a list.

Parameters

- **list** – A pointer on the list to initialize

static inline *sys_snode_t* ***sys_slist_peek_head**(*sys_slist_t* *list)

Peek the first node from the list.

Parameters

- **list** – A point on the list to peek the first node from

Returns

A pointer on the first node of the list (or NULL if none)

static inline *sys_snode_t* ***sys_slist_peek_tail**(*sys_slist_t* *list)

Peek the last node from the list.

Parameters

- **list** – A point on the list to peek the last node from

Returns
A pointer on the last node of the list (or NULL if none)

static inline bool **sys_slist_is_empty**(*sys_slist_t* *list)

Test if the given list is empty.

Parameters

- **list** – A pointer on the list to test

Returns
a boolean, true if it's empty, false otherwise

static inline *sys_snode_t* ***sys_slist_peek_next_no_check**(*sys_snode_t* *node)

Peek the next node from current node, node is not NULL.

Faster then *sys_slist_peek_next()* if node is known not to be NULL.

Parameters

- **node** – A pointer on the node where to peek the next node

Returns
a pointer on the next node (or NULL if none)

static inline *sys_snode_t* ***sys_slist_peek_next**(*sys_snode_t* *node)

Peek the next node from current node.

Parameters

- **node** – A pointer on the node where to peek the next node

Returns
a pointer on the next node (or NULL if none)

static inline void **sys_slist_prepend**(*sys_slist_t* *list, *sys_snode_t* *node)

Prepend a node to the given list.

This and other sys_slist_*() functions are not thread safe.

Parameters

- **list** – A pointer on the list to affect
- **node** – A pointer on the node to prepend

static inline void **sys_slist_append**(*sys_slist_t* *list, *sys_snode_t* *node)

Append a node to the given list.

This and other sys_slist_*() functions are not thread safe.

Parameters

- **list** – A pointer on the list to affect
- **node** – A pointer on the node to append

static inline void **sys_slist_append_list**(*sys_slist_t* *list, void *head, void *tail)

Append a list to the given list.

Append a singly-linked, NULL-terminated list consisting of nodes containing the pointer to the next node as the first element of a node, to *list*. This and other sys_slist_*() functions are not thread safe.

FIXME: Why are the element parameters void *?

Parameters

- **list** – A pointer on the list to affect
- **head** – A pointer to the first element of the list to append
- **tail** – A pointer to the last element of the list to append

static inline void **sys_slist_merge_slist**(*sys_slist_t* *list, *sys_slist_t* *list_to_append)

merge two slists, appending the second one to the first

When the operation is completed, the appending list is empty. This and other sys_slist_*() functions are not thread safe.

Parameters

- **list** – A pointer on the list to affect
- **list_to_append** – A pointer to the list to append.

static inline void **sys_slist_insert**(*sys_slist_t* *list, *sys_snode_t* *prev, *sys_snode_t* *node)

Insert a node to the given list.

This and other sys_slist_*() functions are not thread safe.

Parameters

- **list** – A pointer on the list to affect
- **prev** – A pointer on the previous node
- **node** – A pointer on the node to insert

static inline *sys_snode_t* ***sys_slist_get_not_empty**(*sys_slist_t* *list)

Fetch and remove the first node of the given list.

List must be known to be non-empty. This and other sys_slist_*() functions are not thread safe.

Parameters

- **list** – A pointer on the list to affect

Returns

A pointer to the first node of the list

static inline *sys_snode_t* ***sys_slist_get**(*sys_slist_t* *list)

Fetch and remove the first node of the given list.

This and other sys_slist_*() functions are not thread safe.

Parameters

- **list** – A pointer on the list to affect

Returns

A pointer to the first node of the list (or NULL if empty)

static inline void **sys_slist_remove**(*sys_slist_t* *list, *sys_snode_t* *prev_node, *sys_snode_t* *node)

Remove a node.

This and other sys_slist_*() functions are not thread safe.

Parameters

- **list** – A pointer on the list to affect
- **prev_node** – A pointer on the previous node (can be NULL, which means the node is the list's head)

- **node** – A pointer on the node to remove

static inline bool **sys_slist_find_and_remove**(*sys_slist_t* *list, *sys_snode_t* *node)

Find and remove a node from a list.

This and other sys_slist_*() functions are not thread safe.

Parameters

- **list** – A pointer on the list to affect
- **node** – A pointer on the node to remove from the list

Returns

true if node was removed

static inline bool **sys_slist_find**(*sys_slist_t* *list, *sys_snode_t* *node, *sys_snode_t* **prev)

Find if a node is already linked in a singly linked list.

This and other sys_slist_*() functions are not thread safe.

Parameters

- **list** – A pointer to the list to check
- **node** – A pointer to the node to search in the list
- **prev** – **[out]** A pointer to the previous node

Returns

true if node was found in the list, false otherwise

static inline size_t **sys_slist_len**(*sys_slist_t* *list)

Compute the size of the given list in O(n) time.

Parameters

- **list** – A pointer on the list

Returns

an integer equal to the size of the list, or 0 if empty

Flagged List API Reference

group **flagged-single-linked-list_apis**

Flagged single-linked list implementation.

Similar to *Single-linked list* with the added ability to define user "flags" bits for each node. They can be accessed and modified using the *sys_sfnode_flags_get()* and *sys_sfnode_flags_set()* APIs.

Flagged single-linked list implementation using inline macros/functions. This API is not thread safe, and thus if a list is used across threads, calls to functions must be protected with synchronization primitives.

Defines

SYS_SFLIST_FOR_EACH_NODE(__sl, __sn)

Provide the primitive to iterate on a list Note: the loop is unsafe and thus __sn should not be removed.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_SFLIST_FOR_EACH_NODE(l, n) {
    <user code>
}
```

This and other SYS_SFLIST_*() macros are not thread safe.

Parameters

- **__sl** – A pointer on a sys_sflist_t to iterate on
- **__sn** – A sys_sfnode_t pointer to peek each node of the list

SYS_SFLIST_ITERATE_FROM_NODE(__sl, __sn)

Provide the primitive to iterate on a list, from a node in the list Note: the loop is unsafe and thus __sn should not be removed.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_SFLIST_ITERATE_FROM_NODE(l, n) {
    <user code>
}
```

Like *SYS_SFLIST_FOR_EACH_NODE()*, but __dn already contains a node in the list where to start searching for the next entry from. If NULL, it starts from the head.

This and other SYS_SFLIST_*() macros are not thread safe.

Parameters

- **__sl** – A pointer on a sys_sflist_t to iterate on
- **__sn** – A sys_sfnode_t pointer to peek each node of the list it contains the starting node, or NULL to start from the head

SYS_SFLIST_FOR_EACH_NODE_SAFE(__sl, __sn, __sns)

Provide the primitive to safely iterate on a list Note: __sn can be removed, it will not break the loop.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_SFLIST_FOR_EACH_NODE_SAFE(l, n, s) {
    <user code>
}
```

This and other SYS_SFLIST_*() macros are not thread safe.

Parameters

- **__sl** – A pointer on a sys_sflist_t to iterate on
- **__sn** – A sys_sfnode_t pointer to peek each node of the list
- **__sns** – A sys_sfnode_t pointer for the loop to run safely

SYS_SFLIST_CONTAINER(__ln, __cn, __n)

Provide the primitive to resolve the container of a list node Note: it is safe to use with NULL pointer nodes.

Parameters

- **__ln** – A pointer on a sys_sfnode_t to get its container
- **__cn** – Container struct type pointer
- **__n** – The field name of sys_sfnode_t within the container struct

SYS_SFLIST_PEEK_HEAD_CONTAINER(__sl, __cn, __n)

Provide the primitive to peek container of the list head.

Parameters

- **__sl** – A pointer on a sys_sflist_t to peek
- **__cn** – Container struct type pointer
- **__n** – The field name of sys_sfnode_t within the container struct

SYS_SFLIST_PEEK_TAIL_CONTAINER(__sl, __cn, __n)

Provide the primitive to peek container of the list tail.

Parameters

- **__sl** – A pointer on a sys_sflist_t to peek
- **__cn** – Container struct type pointer
- **__n** – The field name of sys_sfnode_t within the container struct

SYS_SFLIST_PEEK_NEXT_CONTAINER(__cn, __n)

Provide the primitive to peek the next container.

Parameters

- **__cn** – Container struct type pointer
- **__n** – The field name of sys_sfnode_t within the container struct

SYS_SFLIST_FOR_EACH_CONTAINER(__sl, __cn, __n)

Provide the primitive to iterate on a list under a container Note: the loop is unsafe and thus __cn should not be detached.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_SFLIST_FOR_EACH_CONTAINER(l, c, n) {
    <user code>
}
```

Parameters

- **__sl** – A pointer on a sys_sflist_t to iterate on
- **__cn** – A pointer to peek each entry of the list
- **__n** – The field name of sys_sfnode_t within the container struct

SYS_SFLIST_FOR_EACH_CONTAINER_SAFE(__sl, __cn, __cns, __n)

Provide the primitive to safely iterate on a list under a container Note: __cn can be detached, it will not break the loop.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_SFLIST_FOR_EACH_NODE_SAFE(l, c, cn, n) {
    <user code>
}
```

Parameters

- **__sl** – A pointer on a sys_sflist_t to iterate on
- **__cn** – A pointer to peek each entry of the list
- **__cns** – A pointer for the loop to run safely
- **__n** – The field name of sys_sfnode_t within the container struct

SYS_SFLIST_STATIC_INIT(ptr_to_list)

Statically initialize a flagged single-linked list.

Parameters

- **ptr_to_list** – A pointer on the list to initialize

SYS_SFLIST_FLAGS_MASK

Typedefs

typedef struct _sfnode **sys_sfnode_t**

Flagged single-linked list node structure.

typedef struct _sflist **sys_sflist_t**

Flagged single-linked list structure.

Functions

static inline void **sys_sflist_init**(*sys_sflist_t* *list)

Initialize a list.

Parameters

- **list** – A pointer on the list to initialize

static inline uint8_t **sys_sfnode_flags_get**(*sys_sfnode_t* *node)

Fetch flags value for a particular sfnode.

Parameters

- **node** – A pointer to the node to fetch flags from

Returns

The value of flags, which will be between 0 and 3 on 32-bit architectures, or between 0 and 7 on 64-bit architectures

static inline *sys_sfnode_t* ***sys_sflist_peek_head**(*sys_sflist_t* *list)

Peek the first node from the list.

Parameters

- **list** – A point on the list to peek the first node from

Returns

A pointer on the first node of the list (or NULL if none)

static inline *sys_sfnode_t* ***sys_sflist_peek_tail**(*sys_sflist_t* *list)

Peek the last node from the list.

Parameters

- **list** – A point on the list to peek the last node from

Returns

A pointer on the last node of the list (or NULL if none)

static inline void **sys_sfnode_init**(*sys_sfnode_t* *node, uint8_t flags)

Initialize an sflist node.

Set an initial flags value for this slist node, which can be a value between 0 and 3 on 32-bit architectures, or between 0 and 7 on 64-bit architectures. These flags will persist even if the node is moved around within a list, removed, or transplanted to a different slist.

This is ever so slightly faster than *sys_sfnode_flags_set()* and should only be used on a node that hasn't been added to any list.

Parameters

- **node** – A pointer to the node to set the flags on
- **flags** – The flags value to set

static inline void **sys_sfnode_flags_set**(*sys_sfnode_t* *node, uint8_t flags)

Set flags value for an sflist node.

Set a flags value for this slist node, which can be a value between 0 and 3 on 32-bit architectures, or between 0 and 7 on 64-bit architectures. These flags will persist even if the node is moved around within a list, removed, or transplanted to a different slist.

Parameters

- **node** – A pointer to the node to set the flags on
- **flags** – The flags value to set

static inline bool **sys_sflist_is_empty**(*sys_sflist_t* *list)

Test if the given list is empty.

Parameters

- **list** – A pointer on the list to test

Returns
a boolean, true if it's empty, false otherwise

static inline *sys_sfnode_t* ***sys_sflist_peek_next_no_check**(*sys_sfnode_t* *node)

Peek the next node from current node, node is not NULL.

Faster then *sys_sflist_peek_next()* if node is known not to be NULL.

Parameters

- **node** – A pointer on the node where to peek the next node

Returns
a pointer on the next node (or NULL if none)

static inline *sys_sfnode_t* ***sys_sflist_peek_next**(*sys_sfnode_t* *node)

Peek the next node from current node.

Parameters

- **node** – A pointer on the node where to peek the next node

Returns
a pointer on the next node (or NULL if none)

static inline void **sys_sflist_prepend**(*sys_sflist_t* *list, *sys_sfnode_t* *node)

Prepend a node to the given list.

This and other sys_sflist_*() functions are not thread safe.

Parameters

- **list** – A pointer on the list to affect

- **node** – A pointer on the node to prepend

static inline void **sys_sflist_append**(*sys_sflist_t* *list, *sys_sfnode_t* *node)

Append a node to the given list.

This and other sys_sflist_*() functions are not thread safe.

Parameters

- **list** – A pointer on the list to affect
- **node** – A pointer on the node to append

static inline void **sys_sflist_append_list**(*sys_sflist_t* *list, void *head, void *tail)

Append a list to the given list.

Append a singly-linked, NULL-terminated list consisting of nodes containing the pointer to the next node as the first element of a node, to *list*. This and other sys_sflist_*() functions are not thread safe.

FIXME: Why are the element parameters void *?

Parameters

- **list** – A pointer on the list to affect
- **head** – A pointer to the first element of the list to append
- **tail** – A pointer to the last element of the list to append

static inline void **sys_sflist_merge_sflist**(*sys_sflist_t* *list, *sys_sflist_t* *list_to_append)

merge two sflists, appending the second one to the first

When the operation is completed, the appending list is empty. This and other sys_sflist_*() functions are not thread safe.

Parameters

- **list** – A pointer on the list to affect
- **list_to_append** – A pointer to the list to append.

static inline void **sys_sflist_insert**(*sys_sflist_t* *list, *sys_sfnode_t* *prev, *sys_sfnode_t* *node)

Insert a node to the given list.

This and other sys_sflist_*() functions are not thread safe.

Parameters

- **list** – A pointer on the list to affect
- **prev** – A pointer on the previous node
- **node** – A pointer on the node to insert

static inline *sys_sfnode_t* ***sys_sflist_get_not_empty**(*sys_sflist_t* *list)

Fetch and remove the first node of the given list.

List must be known to be non-empty. This and other sys_sflist_*() functions are not thread safe.

Parameters

- **list** – A pointer on the list to affect

Returns

A pointer to the first node of the list

static inline *sys_sfnode_t* ***sys_sflist_get**(*sys_sflist_t* *list)

Fetch and remove the first node of the given list.

This and other sys_sflist_*() functions are not thread safe.

Parameters

- **list** – A pointer on the list to affect

Returns
A pointer to the first node of the list (or NULL if empty)

static inline void **sys_sflist_remove**(*sys_sflist_t* *list, *sys_sfnode_t* *prev_node, *sys_sfnode_t* *node)

Remove a node.

This and other sys_sflist_*() functions are not thread safe.

Parameters

- **list** – A pointer on the list to affect
- **prev_node** – A pointer on the previous node (can be NULL, which means the node is the list's head)
- **node** – A pointer on the node to remove

static inline bool **sys_sflist_find_and_remove**(*sys_sflist_t* *list, *sys_sfnode_t* *node)

Find and remove a node from a list.

This and other sys_sflist_*() functions are not thread safe.

Parameters

- **list** – A pointer on the list to affect
- **node** – A pointer on the node to remove from the list

Returns
true if node was removed

static inline size_t **sys_sflist_len**(*sys_sflist_t* *list)

Compute the size of the given list in O(n) time.

Parameters

- **list** – A pointer on the list

Returns
an integer equal to the size of the list, or 0 if empty

3.5.2 Double-linked List

Similar to the single-linked list in many respects, Zephyr includes a double-linked implementation. This provides the same algorithmic behavior for all the existing slist operations, but also allows for constant-time removal and insertion (at all points: before or after the head, tail or any internal node). To do this, the list stores two pointers per node, and thus has somewhat higher runtime code and memory space needs.

A *sys_dlist_t* struct may be instantiated by the user in any accessible memory. It must be initialized with *sys_dlist_init()* or *SYS_DLIST_STATIC_INIT* before use. The *sys_dnode_t* struct is expected to be provided by the user for any nodes added to the list (typically embedded within the struct to be tracked, as described above). It must be initialized in zeroed/bss memory or with *sys_dnode_init()* before use.

Primitive operations may retrieve the head/tail of a list and the next/prev pointers of a node with *sys_dlist_peek_head()*, *sys_dlist_peek_tail()*, *sys_dlist_peek_next()* and

sys_dlist_peek_prev(). These can all return NULL where appropriate (i.e. for empty lists, or nodes at the endpoints of the list).

A dlist can be modified in constant time by removing a node with *sys_dlist_remove()*, by adding a node to the head or tail of a list with *sys_dlist_prepend()* and *sys_dlist_append()*, or by inserting a node before an existing node with *sys_dlist_insert()*.

As for slist, each node in a dlist can be processed in a natural code block style using *SYS_DLIST_FOR_EACH_NODE*. This macro also exists in a "FROM_NODE" form which allows for iterating from a known starting point, a "SAFE" variant that allows for removing the node being inspected within the code block, a "CONTAINER" style that provides the pointer to a containing struct instead of the raw node, and a "CONTAINER_SAFE" variant that provides both properties.

Convenience utilities provided by dlist include *sys_dlist_insert_at()*, which inserts a node that linearly searches through a list to find the right insertion point, which is provided by the user as a C callback function pointer, and *sys_dnode_is_linked()*, which will affirmatively return whether or not a node is currently linked into a dlist or not (via an implementation that has zero overhead vs. the normal list processing).

Double-linked List Internals

Internally, the dlist implementation is minimal: the *sys_dlist_t* struct contains "head" and "tail" pointer fields, the *sys_dnode_t* contains "prev" and "next" pointers, and no other data is stored. But in practice the two structs are internally identical, and the list struct is inserted as a node into the list itself. This allows for a very clean symmetry of operations:

- An empty list has backpointers to itself in the list struct, which can be trivially detected.
- The head and tail of the list can be detected by comparing the prev/next pointers of a node vs. the list struct address.
- An insertion or deletion never needs to check for the special case of inserting at the head or tail. There are never any NULL pointers within the list to be avoided. Exactly the same operations are run, without tests or branches, for all list modification primitives.

Effectively, a dlist of N nodes can be thought of as a "ring" of "N+1" nodes, where one node represents the list tracking struct.

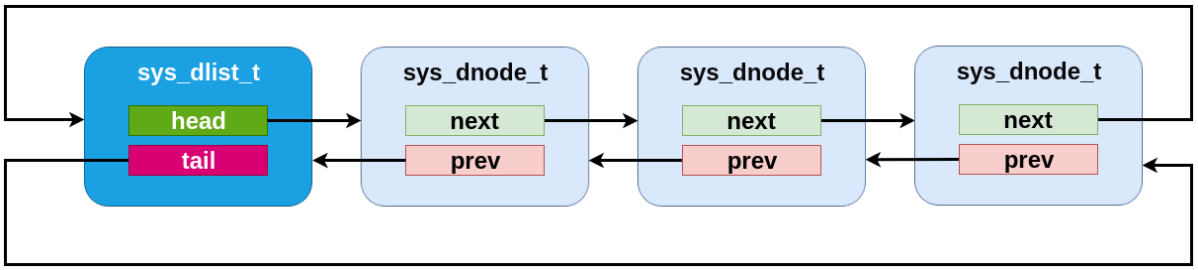


Fig. 6: A dlist containing three elements. Note that the list struct appears as a fourth "element" in the list.

Doubly-linked List API Reference

group `doubly-linked-list_apis`

Doubly-linked list implementation.

Doubly-linked list implementation using inline macros/functions. This API is not thread safe, and thus if a list is used across threads, calls to functions must be protected with synchronization primitives.

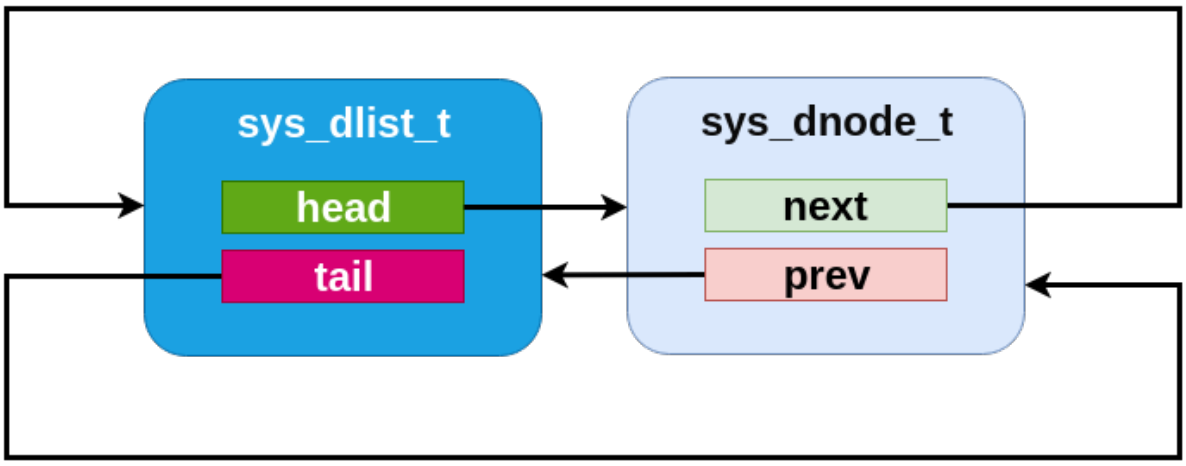


Fig. 7: An dlist containing just one element.

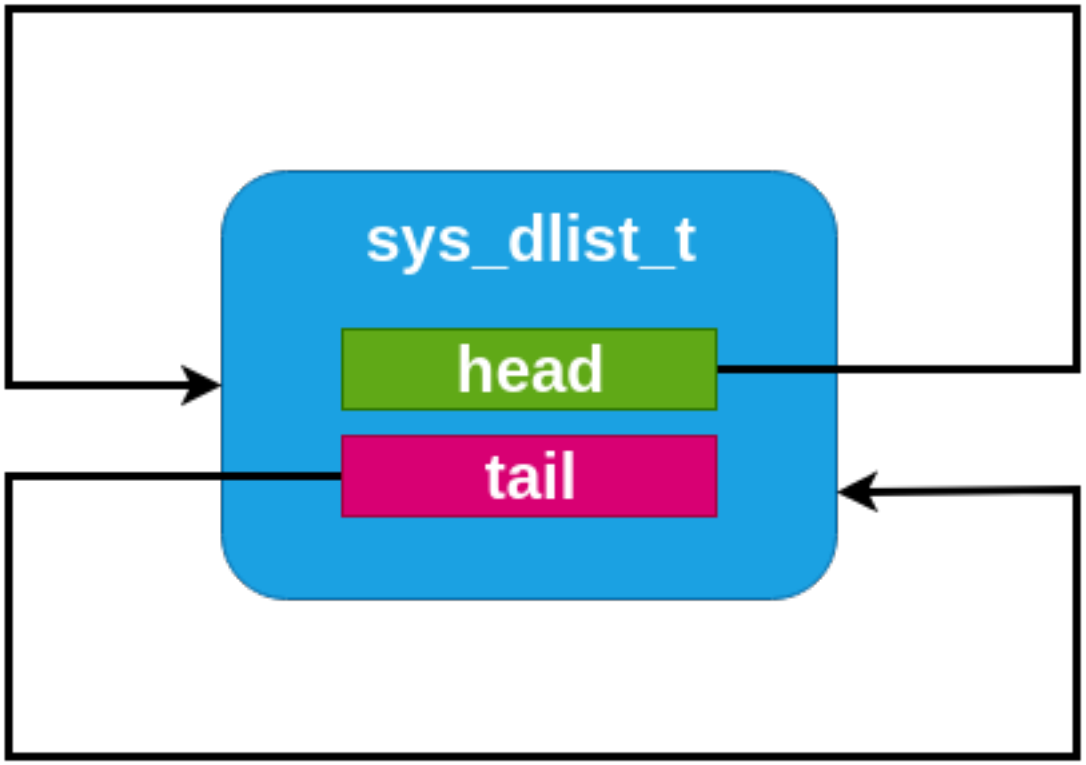


Fig. 8: An empty dlist.

The lists are expected to be initialized such that both the head and tail pointers point to the list itself. Initializing the lists in such a fashion simplifies the adding and removing of nodes to/from the list.

Defines

SYS_DLIST_FOR_EACH_NODE(__dl, __dn)

Provide the primitive to iterate on a list Note: the loop is unsafe and thus __dn should not be removed.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_DLIST_FOR_EACH_NODE(l, n) {
    <user code>
}
```

This and other SYS_DLIST_*() macros are not thread safe.

Parameters

- **__dl** – A pointer on a sys_dlist_t to iterate on
- **__dn** – A sys_dnode_t pointer to peek each node of the list

SYS_DLIST_ITERATE_FROM_NODE(__dl, __dn)

Provide the primitive to iterate on a list, from a node in the list Note: the loop is unsafe and thus __dn should not be removed.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_DLIST_ITERATE_FROM_NODE(l, n) {
    <user code>
}
```

Like *SYS_DLIST_FOR_EACH_NODE()*, but __dn already contains a node in the list where to start searching for the next entry from. If NULL, it starts from the head.

This and other SYS_DLIST_*() macros are not thread safe.

Parameters

- **__dl** – A pointer on a sys_dlist_t to iterate on
- **__dn** – A sys_dnode_t pointer to peek each node of the list; it contains the starting node, or NULL to start from the head

SYS_DLIST_FOR_EACH_NODE_SAFE(__dl, __dn, __dns)

Provide the primitive to safely iterate on a list Note: __dn can be removed, it will not break the loop.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_DLIST_FOR_EACH_NODE_SAFE(l, n, s) {
    <user code>
}
```

This and other SYS_DLIST_*() macros are not thread safe.

Parameters

- **__dl** – A pointer on a sys_dlist_t to iterate on
- **__dn** – A sys_dnode_t pointer to peek each node of the list
- **__dns** – A sys_dnode_t pointer for the loop to run safely

SYS_DLIST_CONTAINER(__dn, __cn, __n)

Provide the primitive to resolve the container of a list node Note: it is safe to use with NULL pointer nodes.

Parameters

- **__dn** – A pointer on a sys_dnode_t to get its container
- **__cn** – Container struct type pointer
- **__n** – The field name of sys_dnode_t within the container struct

SYS_DLIST_PEEK_HEAD_CONTAINER(__dl, __cn, __n)

Provide the primitive to peek container of the list head.

Parameters

- **__dl** – A pointer on a sys_dlist_t to peek
- **__cn** – Container struct type pointer
- **__n** – The field name of sys_dnode_t within the container struct

SYS_DLIST_PEEK_NEXT_CONTAINER(__dl, __cn, __n)

Provide the primitive to peek the next container.

Parameters

- **__dl** – A pointer on a sys_dlist_t to peek
- **__cn** – Container struct type pointer
- **__n** – The field name of sys_dnode_t within the container struct

SYS_DLIST_FOR_EACH_CONTAINER(__dl, __cn, __n)

Provide the primitive to iterate on a list under a container Note: the loop is unsafe and thus __cn should not be detached.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_DLIST_FOR_EACH_CONTAINER(l, c, n) {
    <user code>
}
```

Parameters

- **__dl** – A pointer on a sys_dlist_t to iterate on
- **__cn** – A container struct type pointer to peek each entry of the list
- **__n** – The field name of sys_dnode_t within the container struct

SYS_DLIST_FOR_EACH_CONTAINER_SAFE(__dl, __cn, __cns, __n)

Provide the primitive to safely iterate on a list under a container Note: __cn can be detached, it will not break the loop.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_DLIST_FOR_EACH_CONTAINER_SAFE(l, c, cn, n) {
    <user code>
}
```

Parameters

- **__dl** – A pointer on a sys_dlist_t to iterate on
- **__cn** – A container struct type pointer to peek each entry of the list

- **__cns** – A container struct type pointer for the loop to run safely
- **__n** – The field name of sys_dnode_t within the container struct

SYS_DLIST_STATIC_INIT(ptr_to_list)
Static initializer for a doubly-linked list.

Typedefs

typedef struct _dnode **sys_dlist_t**
Doubly-linked list structure.

typedef struct _dnode **sys_dnode_t**
Doubly-linked list node structure.

Functions

static inline void **sys_dlist_init**(*sys_dlist_t* *list)
initialize list to its empty state

Parameters

- **list** – the doubly-linked list

static inline void **sys_dnode_init**(*sys_dnode_t* *node)
initialize node to its state when not in a list

Parameters

- **node** – the node

static inline bool **sys_dnode_is_linked**(const *sys_dnode_t* *node)
check if a node is a member of any list

Parameters

- **node** – the node

Returns
true if node is linked into a list, false if it is not

static inline bool **sys_dlist_is_head**(*sys_dlist_t* *list, *sys_dnode_t* *node)
check if a node is the list's head

Parameters

- **list** – the doubly-linked list to operate on
- **node** – the node to check

Returns
true if node is the head, false otherwise

static inline bool **sys_dlist_is_tail**(*sys_dlist_t* *list, *sys_dnode_t* *node)
check if a node is the list's tail

Parameters

- **list** – the doubly-linked list to operate on
- **node** – the node to check

Returns
true if node is the tail, false otherwise

static inline bool **sys_dlist_is_empty**(*sys_dlist_t* *list)
check if the list is empty

Parameters
- **list** – the doubly-linked list to operate on

Returns
true if empty, false otherwise

static inline bool **sys_dlist_has_multiple_nodes**(*sys_dlist_t* *list)
check if more than one node present

This and other sys_dlist_*() functions are not thread safe.

Parameters
- **list** – the doubly-linked list to operate on

Returns
true if multiple nodes, false otherwise

static inline *sys_dnode_t* ***sys_dlist_peek_head**(*sys_dlist_t* *list)
get a reference to the head item in the list

Parameters
- **list** – the doubly-linked list to operate on

Returns
a pointer to the head element, NULL if list is empty

static inline *sys_dnode_t* ***sys_dlist_peek_head_not_empty**(*sys_dlist_t* *list)
get a reference to the head item in the list

The list must be known to be non-empty.

Parameters
- **list** – the doubly-linked list to operate on

Returns
a pointer to the head element

static inline *sys_dnode_t* ***sys_dlist_peek_next_no_check**(*sys_dlist_t* *list, *sys_dnode_t* *node)
get a reference to the next item in the list, node is not NULL

Faster than *sys_dlist_peek_next()* if node is known not to be NULL.

Parameters
- **list** – the doubly-linked list to operate on
- **node** – the node from which to get the next element in the list

Returns
a pointer to the next element from a node, NULL if node is the tail

static inline *sys_dnode_t* ***sys_dlist_peek_next**(*sys_dlist_t* *list, *sys_dnode_t* *node)
get a reference to the next item in the list

Parameters
- **list** – the doubly-linked list to operate on
- **node** – the node from which to get the next element in the list

Returns
a pointer to the next element from a node, NULL if node is the tail or NULL (when node comes from reading the head of an empty list).

static inline *sys_dnode_t* ***sys_dlist_peek_prev_no_check**(*sys_dlist_t* *list, *sys_dnode_t* *node)

get a reference to the previous item in the list, node is not NULL

Faster than *sys_dlist_peek_prev()* if node is known not to be NULL.

Parameters

- **list** – the doubly-linked list to operate on
- **node** – the node from which to get the previous element in the list

Returns
a pointer to the previous element from a node, NULL if node is the tail

static inline *sys_dnode_t* ***sys_dlist_peek_prev**(*sys_dlist_t* *list, *sys_dnode_t* *node)

get a reference to the previous item in the list

Parameters

- **list** – the doubly-linked list to operate on
- **node** – the node from which to get the previous element in the list

Returns
a pointer to the previous element from a node, NULL if node is the tail or NULL (when node comes from reading the head of an empty list).

static inline *sys_dnode_t* ***sys_dlist_peek_tail**(*sys_dlist_t* *list)

get a reference to the tail item in the list

Parameters

- **list** – the doubly-linked list to operate on

Returns
a pointer to the tail element, NULL if list is empty

static inline void **sys_dlist_append**(*sys_dlist_t* *list, *sys_dnode_t* *node)

add node to tail of list

This and other sys_dlist_*() functions are not thread safe.

Parameters

- **list** – the doubly-linked list to operate on
- **node** – the element to append

static inline void **sys_dlist_prepend**(*sys_dlist_t* *list, *sys_dnode_t* *node)

add node to head of list

This and other sys_dlist_*() functions are not thread safe.

Parameters

- **list** – the doubly-linked list to operate on
- **node** – the element to append

static inline void **sys_dlist_insert**(*sys_dnode_t* *successor, *sys_dnode_t* *node)

Insert a node into a list.

Insert a node before a specified node in a dlist.

Parameters

- **successor** – the position before which “node” will be inserted
- **node** – the element to insert

static inline void **sys_dlist_insert_at**(*sys_dlist_t* *list, *sys_dnode_t* *node, int (*cond)(*sys_dnode_t* *node, void *data), void *data)

insert node at position

Insert a node in a location depending on a external condition. The cond() function checks if the node is to be inserted *before* the current node against which it is checked. This and other sys_dlist_*() functions are not thread safe.

Parameters

- **list** – the doubly-linked list to operate on
- **node** – the element to insert
- **cond** – a function that determines if the current node is the correct insert point
- **data** – parameter to cond()

static inline void **sys_dlist_remove**(*sys_dnode_t* *node)

remove a specific node from a list

The list is implicit from the node. The node must be part of a list. This and other sys_dlist_*() functions are not thread safe.

Parameters

- **node** – the node to remove

static inline *sys_dnode_t* ***sys_dlist_get**(*sys_dlist_t* *list)

get the first node in a list

This and other sys_dlist_*() functions are not thread safe.

Parameters

- **list** – the doubly-linked list to operate on

Returns
the first node in the list, NULL if list is empty

static inline size_t **sys_dlist_len**(*sys_dlist_t* *list)

Compute the size of the given list in O(n) time.

Parameters

- **list** – A pointer on the list

Returns
an integer equal to the size of the list, or 0 if empty

3.5.3 Multi Producer Single Consumer Packet Buffer

A *Multi Producer Single Consumer Packet Buffer (MPSC_PBUF)* is a circular buffer, whose contents are stored in first-in-first-out order. Variable size packets are stored in the buffer. Packet buffer works under assumption that there is a single context that consumes the data. However, it is possible that another context may interfere to flush the data and never come back (panic case). Packet is produced in two steps: first requested amount of data is allocated, producer fills the data and commits it. Consuming a packet is also performed in two steps: consumer claims the packet, gets pointer to it and length and later on packet is freed. This approach reduces memory copying.

A *MPSC Packet Buffer* has the following key properties:

- Allocate, commit scheme used for packet producing.
- Claim, free scheme used for packet consuming.
- Allocator ensures that contiguous memory of requested length is allocated.
- Following policies can be applied when requested space cannot be allocated:
  - **Overwrite** - oldest entries are dropped until requested amount of memory can be allocated. For each dropped packet user callback is called.
  - **No overwrite** - When requested amount of space cannot be allocated, allocation fails.
- Dedicated, optimized API for storing short packets.
- Allocation with timeout.

Internals

Each packet in the buffer contains `MPSC_PBUF` specific header which is used for internal management. Header consists of 2 bit flags. In order to optimize memory usage, header can be added on top of the user header using `MPSC_PBUF_HDR` and remaining bits in the first word can be application specific. Header consists of following flags:

- valid - bit set to one when packet contains valid user packet
- busy - bit set when packet is being consumed (claimed but not free)

Header state:

valid	busy	description
0	0	space is free
1	0	valid packet
1	1	claimed valid packet
0	1	internal skip packet

Packet buffer space contains free space, valid user packets and internal skip packets. Internal skip packets indicates padding, e.g. at the end of the buffer.

Allocation Using pairs for read and write indexes, available space is determined. If space can be allocated, temporary write index is moved and pointer to a space within buffer is returned. Packet header is reset. If allocation required wrapping of the write index, a skip packet is added to the end of buffer. If space cannot be allocated and overwrite is disabled then `NULL` pointer is returned or context blocks if allocation was with timeout.

Allocation with overwrite If overwrite is enabled, oldest packets are dropped until requested amount of space can be allocated. When packets are dropped `busy` flag is checked in the header to ensure that currently consumed packet is not overwritten. In that case, skip packet is added before busy packet and packets following the busy packet are dropped. When busy packet is being freed, such situation is detected and packet is converted to skip packet to avoid double processing.

Usage

Packet header definition Packet header details can be found in include/zephyr/sys/mpsc_packet.h. API functions can be found in include/zephyr/sys/mpsc_pbuf.h. Headers are split to avoid include spam when declaring the packet.

User header structure must start with internal header:

```
#include <zephyr/sys/mpsc_packet.h>

struct foo_header {
        MPSC_PBUF_HDR;
        uint32_t length: 32 - MPSC_PBUF_HDR_BITS;
};
```

Packet buffer configuration Configuration structure contains buffer details, configuration flags and callbacks. Following callbacks are used by the packet buffer:

- Drop notification - callback called whenever a packet is dropped due to overwrite.
- Get packet length - callback to determine packet length

Packet producing Standard, two step method:

```
foo_packet *packet = mpsc_pbuf_alloc(buffer, len, K_NO_WAIT);

fill_data(packet);

mpsc_pbuf_commit(buffer, packet);
```

Performance optimized storing of small packets:

- 32 bit word packet
- 32 bit word with pointer packet

Note that since packets are written by value, they should already contain `valid` bit set in the header.

```
mpsc_pbuf_put_word(buffer, data);
mpsc_pbuf_put_word_ext(buffer, data, ptr);
```

Packet consuming Two step method:

```
foo_packet *packet = mpsc_pbuf_claim(buffer);

process(packet);

mpsc_pbuf_free(buffer, packet);
```

3.5.4 Single Producer Single Consumer Packet Buffer

A *Single Producer Single Consumer Packet Buffer (SPSC_PBUF)* is a circular buffer, whose contents are stored in first-in-first-out order. Variable size packets are stored in the buffer. Packet buffer works under assumption that there is a single context that produces packets and a single context that consumes the data.

Implementation is focused on performance and memory footprint.

Packets are added to the buffer using `spsc_pbuf_write()` which copies a data into the buffer. If the buffer is full error is returned.

Packets are copied out of the buffer using `spsc_pbuf_read()`.

3.5.5 Balanced Red/Black Tree

For circumstances where sorted containers may become large at runtime, a list becomes problematic due to algorithmic costs of searching it. For these situations, Zephyr provides a balanced tree implementation which has runtimes on search and removal operations bounded at O(log2(N)) for a tree of size N. This is implemented using a conventional red/black tree as described by multiple academic sources.

The *rbtree* tracking struct for a rbtree may be initialized anywhere in user accessible memory. It should contain only zero bits before first use. No specific initialization API is needed or required.

Unlike a list, where position is explicit, the ordering of nodes within an rbtree must be provided as a predicate function by the user. A function of type *rb_lessthan_t()* should be assigned to the `lessthan_fn` field of the *rbtree* struct before any tree operations are attempted. This function should, as its name suggests, return a boolean True value if the first node argument is "less than" the second in the ordering desired by the tree. Note that "equal" is not allowed, nodes within a tree must have a single fixed order for the algorithm to work correctly.

As with the slist and dlist containers, nodes within an rbtree are represented as a *rbnode* structure which exists in user-managed memory, typically embedded within the data structure being tracked in the tree. Unlike the list code, the data within an rbnode is entirely opaque. It is not possible for the user to extract the binary tree topology and "manually" traverse the tree as it is for a list.

Nodes can be inserted into a tree with *rb_insert()* and removed with *rb_remove()*. Access to the "first" and "last" nodes within a tree (in the sense of the order defined by the comparison function) is provided by *rb_get_min()* and *rb_get_max()*. There is also a predicate, *rb_contains()*, which returns a boolean True if the provided node pointer exists as an element within the tree. As described above, all of these routines are guaranteed to have at most log time complexity in the size of the tree.

There are two mechanisms provided for enumerating all elements in an rbtree. The first, *rb_walk()*, is a simple callback implementation where the caller specifies a C function pointer and an untyped argument to be passed to it, and the tree code calls that function for each node in order. This has the advantage of a very simple implementation, at the cost of a somewhat more cumbersome API for the user (not unlike ISO C's `bsearch()` routine). It is a recursive implementation, however, and is thus not always available in environments that forbid the use of unbounded stack techniques like recursion.

There is also a *RB_FOR_EACH* iterator provided, which, like the similar APIs for the lists, works to iterate over a list in a more natural way, using a nested code block instead of a callback. It is also nonrecursive, though it requires log-sized space on the stack by default (however, this can be configured to use a fixed/maximally size buffer instead where needed to avoid the dynamic allocation). As with the lists, this is also available in a *RB_FOR_EACH_CONTAINER* variant which enumerates using a pointer to a container field and not the raw node pointer.

Tree Internals

As described, the Zephyr rbtree implementation is a conventional red/black tree as described pervasively in academic sources. Low level details about the algorithm are out of scope for this document, as they match existing conventions. This discussion will be limited to details notable or specific to the Zephyr implementation.

The core invariant guaranteed by the tree is that the path from the root of the tree to any leaf is no more than twice as long as the path to any other leaf. This is achieved by associating one bit of "color" with each node, either red or black, and enforcing a rule that no red child can be a child of another red child (i.e. that the number of black nodes on any path to the root must be the same, and that no more than that number of "extra" red nodes may be present). This rule is enforced by a set of rotation rules used to "fix" trees following modification.

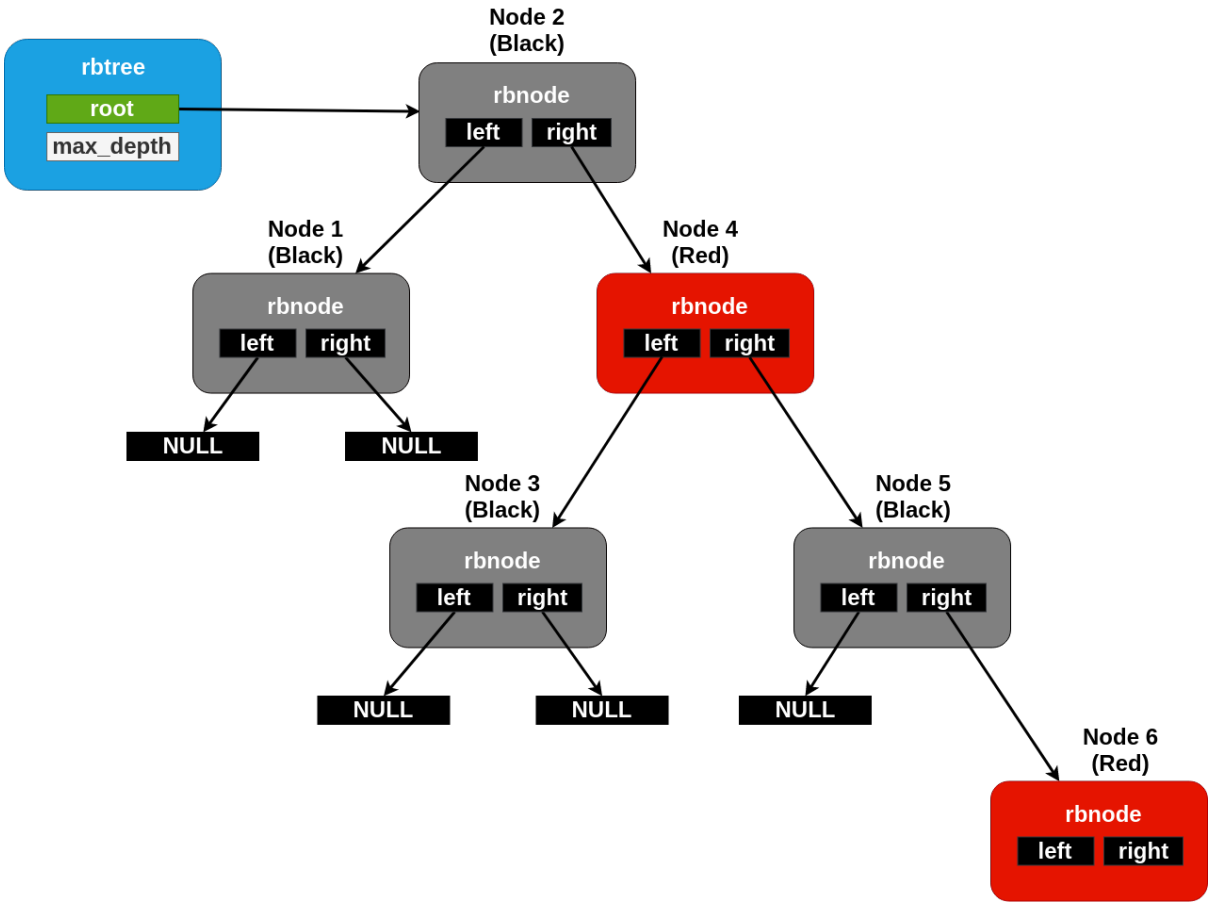


Fig. 9: A maximally unbalanced rbtree with a black height of two. No more nodes can be added underneath the rightmost node without rebalancing.

These rotations are conceptually implemented on top of a primitive that "swaps" the position of one node with another in the list. Typical implementations effect this by simply swapping the nodes internal "data" pointers, but because the Zephyr *rbnode* is intrusive, that cannot work. Zephyr must include somewhat more elaborate code to handle the edge cases (for example, one swapped node can be the root, or the two may already be parent/child).

The *rbnode* struct for a Zephyr rbtree contains only two pointers, representing the "left", and "right" children of a node within the binary tree. Traversal of a tree for rebalancing following modification, however, routinely requires the ability to iterate "upwards" from a node as well. It is very common for red/black trees in the industry to store a third "parent" pointer for this purpose. Zephyr avoids this requirement by building a "stack" of node pointers locally as it traverses downward through the tree and updating it appropriately as modifications are made. So a Zephyr rbtree can be implemented with no more runtime storage overhead than a dlist.

These properties, of a balanced tree data structure that works with only two pointers of data per node and that works without any need for a memory allocation API, are quite rare in the industry and are somewhat unique to Zephyr.

Red/Black Tree API Reference

group `rbtree_apis`

Balanced Red/Black Tree implementation.

This implements an intrusive balanced tree that guarantees O(log2(N)) runtime for all operations and amortized O(1) behavior for creation and destruction of whole trees. The algorithms and naming are conventional per existing academic and didactic implementations, c.f.:

https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

The implementation is size-optimized to prioritize runtime memory usage. The data structure is intrusive, which is to say the *rbnode* handle is intended to be placed in a separate struct, in the same way as with other such structures (e.g. Zephyr's *Doubly-linked list*), and requires no data pointer to be stored in the node. The color bit is unioned with a pointer (fairly common for such libraries). Most notably, there is no "parent" pointer stored in the node, the upper structure of the tree being generated dynamically via a stack as the tree is recursed. So the overall memory overhead of a node is just two pointers, identical with a doubly-linked list.

Defines

`RB_FOR_EACH`(tree, node)

Walk a tree in-order without recursing.

While *rb_walk()* is very simple, recursing on the C stack can be clumsy for some purposes and on some architectures wastes significant memory in stack frames. This macro implements a non-recursive "foreach" loop that can iterate directly on the tree, at a moderate cost in code size.

Note that the resulting loop is not safe against modifications to the tree. Changes to the tree structure during the loop will produce incorrect results, as nodes may be skipped or duplicated. Unlike linked lists, no _SAFE variant exists.

Note also that the macro expands its arguments multiple times, so they should not be expressions with side effects.

Parameters

- `tree` – A pointer to a struct rbtree to walk

- **node** – The symbol name of a local struct rbnode* variable to use as the iterator

RB_FOR_EACH_CONTAINER(tree, node, field)

Loop over rbtree with implicit container field logic.

As for *RB_FOR_EACH()*, but "node" can have an arbitrary type containing a struct rbnode.

Parameters

- **tree** – A pointer to a struct rbtree to walk
- **node** – The symbol name of a local iterator
- **field** – The field name of a struct rbnode inside node

Typedefs

typedef bool (*rb_lessthan_t)(struct *rbnode* *a, struct *rbnode* *b)

Red/black tree comparison predicate.

Compares the two nodes and returns true if node A is strictly less than B according to the tree's sorting criteria, false otherwise.

Note that during insert, the new node being inserted will always be "A", where "B" is the existing node within the tree against which it is being compared. This trait can be used (with care!) to implement "most/least recently added" semantics between nodes which would otherwise compare as equal.

typedef void (*rb_visit_t)(struct *rbnode* *node, void *cookie)

Prototype for node visitor callback.

Param node
Node being visited

Param cookie
User-specified data

Functions

void rb_insert(struct *rbtree* *tree, struct *rbnode* *node)

Insert node into tree.

void rb_remove(struct *rbtree* *tree, struct *rbnode* *node)

Remove node from tree.

static inline struct *rbnode* *rb_get_min(struct *rbtree* *tree)

Returns the lowest-sorted member of the tree.

static inline struct *rbnode* *rb_get_max(struct *rbtree* *tree)

Returns the highest-sorted member of the tree.

bool rb_contains(struct *rbtree* *tree, struct *rbnode* *node)

Returns true if the given node is part of the tree.

Note that this does not internally dereference the node pointer (though the tree's lessthan callback might!), it just tests it for equality with items in the tree. So it's feasible to use this to implement a "set" construct by simply testing the pointer value itself.

static inline void **rb_walk**(struct *rbtree* *tree, *rb_visit_t* visit_fn, void *cookie)

Walk/enumerate a rbtree.

Very simple recursive enumeration. Low code size, but requiring a separate function can be clumsy for the user and there is no way to break out of the loop early. See RB_FOR_EACH for an iterative implementation.

struct **rbnode**

#include <*rb.h*> Balanced red/black tree node structure.

struct **rbtree**

#include <*rb.h*> Balanced red/black tree structure.

Public Members

struct *rbnode* ***root**

Root node of the tree.

rb_lessthan_t **lessthan_fn**

Comparison function for nodes in the tree.

3.5.6 Ring Buffers

A *ring buffer* is a circular buffer, whose contents are stored in first-in-first-out order.

For circumstances where an application needs to implement asynchronous "streaming" copying of data, Zephyr provides a `struct ring_buf` abstraction to manage copies of such data in and out of a shared buffer of memory.

Two content data modes are supported:

- **Byte mode**: raw bytes can be enqueued and dequeued.
- **Data item mode**: Multiple 32-bit word data items with metadata can be enqueued and dequeued from the ring buffer in chunks of up to 1020 bytes. Each data item also has two associated metadata values: a type identifier and a 16-bit integer value, both of which are application-specific.

While the underlying data structure is the same, it is not legal to mix these two modes on a single ring buffer instance. A ring buffer initialized with a byte count must be used only with the "bytes" API, one initialized with a word count must use the "items" calls.

- *Concepts*
  - *Byte mode*
  - *Data item mode*
  - *Concurrency*
  - *Internal Operation*
- *Implementation*
  - *Defining a Ring Buffer*
  - *Enqueuing Data*

  - *Retrieving Data*
- *Configuration Options*
- *API Reference*

Concepts

Any number of ring buffers can be defined (limited only by available RAM). Each ring buffer is referenced by its memory address.

A ring buffer has the following key properties:

- A **data buffer** of bytes or 32-bit words. The data buffer contains the raw bytes or 32-bit words that have been added to the ring buffer but not yet removed.
- A **data buffer size**, measured in bytes or 32-bit words. This governs the maximum amount of data (including possible metadata values) the ring buffer can hold.

A ring buffer must be initialized before it can be used. This sets its data buffer to empty.

A `struct ring_buf` may be placed anywhere in user-accessible memory, and must be initialized with *ring_buf_init()* or *ring_buf_item_init()* before use. This must be provided a region of user-controlled memory for use as the buffer itself. Note carefully that the units of the size of the buffer passed change (either bytes or words) depending on how the ring buffer will be used later. Macros for combining these steps in a single static declaration exist for convenience. *RING_BUF_DECLARE* will declare and statically initialize a ring buffer with a specified byte count, where *RING_BUF_ITEM_DECLARE* will declare and statically initialize a buffer with a given count of 32 bit words. *RING_BUF_ITEM_SIZEOF* will compute the size in 32-bit words corresponding to a type or an expression. Note: rounds up if the size is not a multiple of 32 bits.

"Bytes" data may be copied into the ring buffer using *ring_buf_put()*, passing a data pointer and byte count. These bytes will be copied into the buffer in order, as many as will fit in the allocated buffer. The total number of bytes copied (which may be fewer than provided) will be returned. Likewise *ring_buf_get()* will copy bytes out of the ring buffer in the order that they were written, into a user-provided buffer, returning the number of bytes that were transferred.

To avoid multiply-copied-data situations, a "claim" API exists for byte mode. *ring_buf_put_claim()* takes a byte size value from the user and returns a pointer to memory internal to the ring buffer that can be used to receive those bytes, along with a size of the contiguous internal region (which may be smaller than requested). The user can then copy data into that region at a later time without assembling all the bytes in a single region first. When complete, *ring_buf_put_finish()* can be used to signal the buffer that the transfer is complete, passing the number of bytes actually transferred. At this point a new transfer can be initiated. Similarly, *ring_buf_get_claim()* returns a pointer to internal ring buffer data from which the user can read without making a verbatim copy, and *ring_buf_get_finish()* signals the buffer with how many bytes have been consumed and allows for a new transfer to begin.

"Items" mode works similarly to bytes mode, except that all transfers are in units of 32 bit words and all memory is assumed to be aligned on 32 bit boundaries. The write and read operations are *ring_buf_item_put()* and *ring_buf_item_get()*, and work otherwise identically to the bytes mode APIs. There no "claim" API provided for items mode. One important difference is that unlike *ring_buf_put()*, *ring_buf_item_put()* will not do a partial transfer; it will return an error in the case where the provided data does not fit in its entirety.

The user can manage the capacity of a ring buffer without modifying it using either *ring_buf_space_get()* or *ring_buf_item_space_get()* which returns the number of free bytes or free 32-bit item words respectively, or by testing the *ring_buf_is_empty()* predicate.

Finally, a *ring_buf_reset()* call exists to immediately empty a ring buffer, discarding the tracking of any bytes or items already written to the buffer. It does not modify the memory contents of the buffer itself, however.

Byte mode A **byte mode** ring buffer instance is declared using *RING_BUF_DECLARE()* and accessed using: *ring_buf_put_claim()*, *ring_buf_put_finish()*, *ring_buf_get_claim()*, *ring_buf_get_finish()*, *ring_buf_put()* and *ring_buf_get()*.

Data can be copied into the ring buffer (see *ring_buf_put()*) or ring buffer memory can be used directly by the user. In the latter case, the operation is split into three stages:

1. allocating the buffer (*ring_buf_put_claim()*) when user requests the destination location where data can be written.
2. writing the data by the user (e.g. buffer written by DMA).
3. indicating the amount of data written to the provided buffer (*ring_buf_put_finish()*). The amount can be less than or equal to the allocated amount.

Data can be retrieved from a ring buffer through copying (see *ring_buf_get()*) or accessed directly by address. In the latter case, the operation is split into three stages:

1. retrieving source location with valid data written to a ring buffer (see *ring_buf_get_claim()*).
2. processing data
3. freeing processed data (see *ring_buf_get_finish()*). The amount freed can be less than or equal or to the retrieved amount.

Data item mode A **data item mode** ring buffer instance is declared using *RING_BUF_ITEM_DECLARE()* and accessed using *ring_buf_item_put()* and *ring_buf_item_get()*.

A ring buffer **data item** is an array of 32-bit words from 0 to 1020 bytes in length. When a data item is **enqueued** (*ring_buf_item_put()*) its contents are copied to the data buffer, along with its associated metadata values (which occupy one additional 32-bit word). If the ring buffer has insufficient space to hold the new data item the enqueue operation fails.

A data item is **dequeued** (*ring_buf_item_get()*) from a ring buffer by removing the oldest enqueued item. The contents of the dequeued data item, as well as its two metadata values, are copied to areas supplied by the retriever. If the ring buffer is empty, or if the data array supplied by the retriever is not large enough to hold the data item's data, the dequeue operation fails.

Concurrency The ring buffer APIs do not provide any concurrency control. Depending on usage (particularly with respect to number of concurrent readers/writers) applications may need to protect the ring buffer with mutexes and/or use semaphores to notify consumers that there is data to read.

For the trivial case of one producer and one consumer, concurrency control shouldn't be needed.

Internal Operation Data streamed through a ring buffer is always written to the next byte within the buffer, wrapping around to the first element after reaching the end, thus the "ring" structure. Internally, the `struct ring_buf` contains its own buffer pointer and its size, and also a set of "head" and "tail" indices representing where the next read and write operations may occur.

This boundary is invisible to the user using the normal put/get APIs, but becomes a barrier to the "claim" API, because obviously no contiguous region can be returned that crosses the end of the buffer. This can be surprising to application code, and produce performance artifacts when transfers need to happen close to the end of the buffer, as the number of calls to claim/finish needs to double for such transfers.

Implementation

Defining a Ring Buffer A ring buffer is defined using a variable of type *ring_buf*. It must then be initialized by calling *ring_buf_init()* or *ring_buf_item_init()*.

The following code defines and initializes an empty **data item mode** ring buffer (which is part of a larger data structure). The ring buffer's data buffer is capable of holding 64 words of data and metadata information.

```
#define MY_RING_BUF_WORDS 64

struct my_struct {
    struct ring_buf rb;
    uint32_t buffer[MY_RING_BUF_WORDS];
    ...
};
struct my_struct ms;

void init_my_struct {
    ring_buf_item_init(&ms.rb, MY_RING_BUF_WORDS, ms.buffer);
    ...
}
```

Alternatively, a ring buffer can be defined and initialized at compile time using one of two macros at file scope. Each macro defines both the ring buffer itself and its data buffer.

The following code defines a **data item mode** ring buffer:

```
#define MY_RING_BUF_WORDS 93
RING_BUF_ITEM_DECLARE(my_ring_buf, MY_RING_BUF_WORDS);
```

The following code defines a ring buffer intended to be used for raw bytes:

```
#define MY_RING_BUF_BYTES 93
RING_BUF_DECLARE(my_ring_buf, MY_RING_BUF_BYTES);
```

Enqueuing Data Bytes are copied to a **byte mode** ring buffer by calling *ring_buf_put()*.

```
uint8_t my_data[MY_RING_BUF_BYTES];
uint32_t ret;

ret = ring_buf_put(&ring_buf, my_data, MY_RING_BUF_BYTES);
if (ret != MY_RING_BUF_BYTES) {
    /* not enough room, partial copy. */
    ...
}
```

Data can be added to a **byte mode** ring buffer by directly accessing the ring buffer's memory. For example:

```
uint32_t size;
uint32_t rx_size;
uint8_t *data;
int err;

/* Allocate buffer within a ring buffer memory. */
size = ring_buf_put_claim(&ring_buf, &data, MY_RING_BUF_BYTES);

/* Work directly on a ring buffer memory. */
rx_size = uart_rx(data, size);
```

(continues on next page)

(continued from previous page)

```
/* Indicate amount of valid data. rx_size can be equal or less than size. */
err = ring_buf_put_finish(&ring_buf, rx_size);
if (err != 0) {
    /* This shouldn't happen unless rx_size > size */
    ...
}
```

A data item is added to a ring buffer by calling *ring_buf_item_put()*.

```
uint32_t data[MY_DATA_WORDS];
int ret;

ret = ring_buf_item_put(&ring_buf, TYPE_FOO, 0, data, MY_DATA_WORDS);
if (ret == -EMSGSIZE) {
    /* not enough room for the data item */
    ...
}
```

If the data item requires only the type or application-specific integer value (i.e. it has no data array), a size of 0 and data pointer of NULL can be specified.

```
int ret;

ret = ring_buf_item_put(&ring_buf, TYPE_BAR, 17, NULL, 0);
if (ret == -EMSGSIZE) {
    /* not enough room for the data item */
    ...
}
```

Retrieving Data Data bytes are copied out from a **byte mode** ring buffer by calling *ring_buf_get()*. For example:

```
uint8_t my_data[MY_DATA_BYTES];
size_t  ret;

ret = ring_buf_get(&ring_buf, my_data, sizeof(my_data));
if (ret != sizeof(my_data)) {
    /* Fewer bytes copied. */
} else {
    /* Requested amount of bytes retrieved. */
    ...
}
```

Data can be retrieved from a **byte mode** ring buffer by direct operations on the ring buffer's memory. For example:

```
uint32_t size;
uint32_t proc_size;
uint8_t *data;
int err;

/* Get buffer within a ring buffer memory. */
size = ring_buf_get_claim(&ring_buf, &data, MY_RING_BUF_BYTES);

/* Work directly on a ring buffer memory. */
proc_size = process(data, size);

/* Indicate amount of data that can be freed. proc_size can be equal or less
 * than size.
```

(continues on next page)

(continued from previous page)

```
 */
err = ring_buf_get_finish(&ring_buf, proc_size);
if (err != 0) {
    /* proc_size exceeds amount of valid data in a ring buffer. */
    ...
}
```

A data item is removed from a ring buffer by calling *ring_buf_item_get()*.

```
uint32_t my_data[MY_DATA_WORDS];
uint16_t my_type;
uint8_t  my_value;
uint8_t  my_size;
int ret;

my_size = MY_DATA_WORDS;
ret = ring_buf_item_get(&ring_buf, &my_type, &my_value, my_data, &my_size);
if (ret == -EMSGSIZE) {
    printk("Buffer is too small, need %d uint32_t\n", my_size);
} else if (ret == -EAGAIN) {
    printk("Ring buffer is empty\n");
} else {
    printk("Got item of type %u value &u of size %u dwords\n",
           my_type, my_value, my_size);
    ...
}
```

Configuration Options

Related configuration options:

- CONFIG_RING_BUFFER: Enable ring buffer.

API Reference

The following ring buffer APIs are provided by include/zephyr/sys/ring_buffer.h:

group **ring_buffer_apis**

Simple ring buffer implementation.

Defines

RING_BUF_DECLARE(name, size8)

Define and initialize a ring buffer for byte data.

This macro establishes a ring buffer of an arbitrary size. The basic storage unit is a byte.

The ring buffer can be accessed outside the module where it is defined using:

```
extern struct ring_buf <name>;
```

Parameters

- **name** – Name of the ring buffer.
- **size8** – Size of ring buffer (in bytes).

RING_BUF_ITEM_DECLARE(name, size32)

Define and initialize an “item based” ring buffer.

This macro establishes an “item based” ring buffer. Each data item is an array of 32-bit words (from zero to 1020 bytes in length), coupled with a 16-bit type identifier and an 8-bit integer value.

The ring buffer can be accessed outside the module where it is defined using:

```
extern struct ring_buf <name>;
```

Parameters

- **name** – Name of the ring buffer.
- **size32** – Size of ring buffer (in 32-bit words).

RING_BUF_ITEM_DECLARE_SIZE(name, size32)

Define and initialize an “item based” ring buffer.

This exists for backward compatibility reasons. *RING_BUF_ITEM_DECLARE* should be used instead.

Parameters

- **name** – Name of the ring buffer.
- **size32** – Size of ring buffer (in 32-bit words).

RING_BUF_ITEM_DECLARE_POW2(name, pow)

Define and initialize a power-of-2 sized “item based” ring buffer.

This macro establishes an “item based” ring buffer by specifying its size using a power of 2. This exists mainly for backward compatibility reasons. *RING_BUF_ITEM_DECLARE* should be used instead.

Parameters

- **name** – Name of the ring buffer.
- **pow** – Ring buffer size exponent.

RING_BUF_ITEM_SIZEOF(expr)

Compute the ring buffer size in 32-bit needed to store an element.

The argument can be a type or an expression. Note: rounds up if the size is not a multiple of 32 bits.

Parameters

- **expr** – Expression or type to compute the size of

Functions

static inline void **ring_buf_internal_reset**(struct *ring_buf* *buf, int32_t value)

Function to force *ring_buf* internal states to given value.

Any value other than 0 makes sense only in validation testing context.

static inline void **ring_buf_init**(struct *ring_buf* *buf, uint32_t size, uint8_t *data)

Initialize a ring buffer for byte data.

This routine initializes a ring buffer, prior to its first use. It is only used for ring buffers not defined using RING_BUF_DECLARE.

Parameters

- **buf** – Address of ring buffer.
- **size** – Ring buffer size (in bytes).
- **data** – Ring buffer data area (uint8_t data[size]).

static inline void **ring_buf_item_init**(struct *ring_buf* *buf, uint32_t size, uint32_t *data)

Initialize an “item based” ring buffer.

This routine initializes a ring buffer, prior to its first use. It is only used for ring buffers not defined using RING_BUF_ITEM_DECLARE.

Each data item is an array of 32-bit words (from zero to 1020 bytes in length), coupled with a 16-bit type identifier and an 8-bit integer value.

Each data item is an array of 32-bit words (from zero to 1020 bytes in length), coupled with a 16-bit type identifier and an 8-bit integer value.

Parameters

- **buf** – Address of ring buffer.
- **size** – Ring buffer size (in 32-bit words)
- **data** – Ring buffer data area (uint32_t data[size]).

static inline bool **ring_buf_is_empty**(struct *ring_buf* *buf)

Determine if a ring buffer is empty.

Parameters

- **buf** – Address of ring buffer.

Returns

true if the ring buffer is empty, or false if not.

static inline void **ring_buf_reset**(struct *ring_buf* *buf)

Reset ring buffer state.

Parameters

- **buf** – Address of ring buffer.

static inline uint32_t **ring_buf_space_get**(struct *ring_buf* *buf)

Determine free space in a ring buffer.

Parameters

- **buf** – Address of ring buffer.

Returns

Ring buffer free space (in bytes).

static inline uint32_t **ring_buf_item_space_get**(struct *ring_buf* *buf)

Determine free space in an “item based” ring buffer.

Parameters

- **buf** – Address of ring buffer.

Returns

Ring buffer free space (in 32-bit words).

static inline uint32_t **ring_buf_capacity_get**(struct *ring_buf* *buf)

Return ring buffer capacity.

Parameters

- **buf** – Address of ring buffer.

Returns
Ring buffer capacity (in bytes).

static inline uint32_t **ring_buf_size_get**(struct *ring_buf* *buf)

Determine used space in a ring buffer.

Parameters

- **buf** – Address of ring buffer.

Returns
Ring buffer space used (in bytes).

uint32_t **ring_buf_put_claim**(struct *ring_buf* *buf, uint8_t **data, uint32_t size)

Allocate buffer for writing data to a ring buffer.

With this routine, memory copying can be reduced since internal ring buffer can be used directly by the user. Once data is written to allocated area number of bytes written must be confirmed (see *ring_buf_put_finish*).

Warning

Use cases involving multiple writers to the ring buffer must prevent concurrent write operations, either by preventing all writers from being preempted or by using a mutex to govern writes to the ring buffer.

Warning

Ring buffer instance should not mix byte access and item access (calls prefixed with ring_buf_item_).

Parameters

- **buf** – **[in]** Address of ring buffer.
- **data** – **[out]** Pointer to the address. It is set to a location within ring buffer.
- **size** – **[in]** Requested allocation size (in bytes).

Returns
Size of allocated buffer which can be smaller than requested if there is not enough free space or buffer wraps.

int **ring_buf_put_finish**(struct *ring_buf* *buf, uint32_t size)

Indicate number of bytes written to allocated buffers.

The number of bytes must be equal to or lower than the sum corresponding to all preceding *ring_buf_put_claim* invocations (or even 0). Surplus bytes will be returned to the available free buffer space.

Warning

Use cases involving multiple writers to the ring buffer must prevent concurrent write operations, either by preventing all writers from being preempted or by using a mutex to govern writes to the ring buffer.

Warning

Ring buffer instance should not mix byte access and item access (calls prefixed with ring_buf_item_).

Parameters

- `buf` – Address of ring buffer.
- `size` – Number of valid bytes in the allocated buffers.

Return values

- `0` – Successful operation.
- `-EINVAL` – Provided *size* exceeds free space in the ring buffer.

uint32_t **ring_buf_put**(struct *ring_buf* *buf, const uint8_t *data, uint32_t size)

Write (copy) data to a ring buffer.

This routine writes data to a ring buffer *buf*.

Warning

Use cases involving multiple writers to the ring buffer must prevent concurrent write operations, either by preventing all writers from being preempted or by using a mutex to govern writes to the ring buffer.

Warning

Ring buffer instance should not mix byte access and item access (calls prefixed with ring_buf_item_).

Parameters

- `buf` – Address of ring buffer.
- `data` – Address of data.
- `size` – Data size (in bytes).

Return values

`Number` – of bytes written.

uint32_t **ring_buf_get_claim**(struct *ring_buf* *buf, uint8_t **data, uint32_t size)

Get address of a valid data in a ring buffer.

With this routine, memory copying can be reduced since internal ring buffer can be used directly by the user. Once data is processed it must be freed using *ring_buf_get_finish*.

Warning

Use cases involving multiple reads of the ring buffer must prevent concurrent read operations, either by preventing all readers from being preempted or by using a mutex to govern reads to the ring buffer.

Warning

Ring buffer instance should not mix byte access and item access (calls prefixed with ring_buf_item_).

Parameters

- `buf` – **[in]** Address of ring buffer.
- `data` – **[out]** Pointer to the address. It is set to a location within ring buffer.
- `size` – **[in]** Requested size (in bytes).

Returns

Number of valid bytes in the provided buffer which can be smaller than requested if there is not enough free space or buffer wraps.

int `ring_buf_get_finish`(struct *ring_buf* *buf, uint32_t size)

Indicate number of bytes read from claimed buffer.

The number of bytes must be equal or lower than the sum corresponding to all preceding *ring_buf_get_claim* invocations (or even 0). Surplus bytes will remain available in the buffer.

Warning

Use cases involving multiple reads of the ring buffer must prevent concurrent read operations, either by preventing all readers from being preempted or by using a mutex to govern reads to the ring buffer.

Warning

Ring buffer instance should not mix byte access and item mode (calls prefixed with ring_buf_item_).

Parameters

- `buf` – Address of ring buffer.
- `size` – Number of bytes that can be freed.

Return values

- `0` – Successful operation.
- `-EINVAL` – Provided *size* exceeds valid bytes in the ring buffer.

uint32_t `ring_buf_get`(struct *ring_buf* *buf, uint8_t *data, uint32_t size)

Read data from a ring buffer.

This routine reads data from a ring buffer *buf*.

Warning

Use cases involving multiple reads of the ring buffer must prevent concurrent read operations, either by preventing all readers from being preempted or by using a mutex to govern reads to the ring buffer.

Warning

Ring buffer instance should not mix byte access and item mode (calls prefixed with ring_buf_item_).

Parameters

- `buf` – Address of ring buffer.
- `data` – Address of the output buffer. Can be NULL to discard data.
- `size` – Data size (in bytes).

Return values

`Number` – of bytes written to the output buffer.

uint32_t `ring_buf_peek`(struct *ring_buf* *buf, uint8_t *data, uint32_t size)

Peek at data from a ring buffer.

This routine reads data from a ring buffer *buf* without removal.

Warning

Use cases involving multiple reads of the ring buffer must prevent concurrent read operations, either by preventing all readers from being preempted or by using a mutex to govern reads to the ring buffer.

Warning

Ring buffer instance should not mix byte access and item mode (calls prefixed with ring_buf_item_).

Warning

Multiple calls to peek will result in the same data being 'peeked' multiple times. To remove data, use either *ring_buf_get* or *ring_buf_get_claim* followed by *ring_buf_get_finish* with a non-zero `size`.

Parameters

- `buf` – Address of ring buffer.
- `data` – Address of the output buffer. Cannot be NULL.
- `size` – Data size (in bytes).

Return values

`Number` – of bytes written to the output buffer.

int `ring_buf_item_put`(struct *ring_buf* *buf, uint16_t type, uint8_t value, uint32_t *data, uint8_t size32)

Write a data item to a ring buffer.

This routine writes a data item to ring buffer *buf*. The data item is an array of 32-bit words (from zero to 1020 bytes in length), coupled with a 16-bit type identifier and an 8-bit integer value.

Warning

Use cases involving multiple writers to the ring buffer must prevent concurrent write operations, either by preventing all writers from being preempted or by using a mutex to govern writes to the ring buffer.

Parameters

- `buf` – Address of ring buffer.
- `type` – Data item's type identifier (application specific).
- `value` – Data item's integer value (application specific).
- `data` – Address of data item.
- `size32` – Data item size (number of 32-bit words).

Return values

- `0` – Data item was written.
- `-EMSGSIZE` – Ring buffer has insufficient free space.

int `ring_buf_item_get`(struct *ring_buf* *buf, uint16_t *type, uint8_t *value, uint32_t *data, uint8_t *size32)

Read a data item from a ring buffer.

This routine reads a data item from ring buffer *buf*. The data item is an array of 32-bit words (up to 1020 bytes in length), coupled with a 16-bit type identifier and an 8-bit integer value.

Warning

Use cases involving multiple reads of the ring buffer must prevent concurrent read operations, either by preventing all readers from being preempted or by using a mutex to govern reads to the ring buffer.

Parameters

- `buf` – Address of ring buffer.
- `type` – Area to store the data item's type identifier.
- `value` – Area to store the data item's integer value.
- `data` – Area to store the data item. Can be NULL to discard data.
- `size32` – Size of the data item storage area (number of 32-bit chunks).

Return values

- `0` – Data item was fetched; *size32* now contains the number of 32-bit words read into data area *data*.
- `-EAGAIN` – Ring buffer is empty.
- `-EMSGSIZE` – Data area *data* is too small; *size32* now contains the number of 32-bit words needed.

struct `ring_buf`

#include <ring_buffer.h> A structure to represent a ring buffer.

3.5.7 Multi Producer Single Consumer Lock Free Queue

A *Multi Producer Single Consumer Lock Free Queue (MPSC)* is an lockfree intrusive queue based on atomic pointer swaps as described by Dmitry Vyukov at 1024cores.

API Reference

group `mpsc_lockfree`

Multiple Producer Single Consumer (MPSC) Lockfree Queue API.

Defines

`mpsc_ptr_get`(ptr)

`mpsc_ptr_set`(ptr, val)

`mpsc_ptr_set_get`(ptr, val)

`MPSC_INIT`(symbol)

Static initializer for a mpsc queue.

Since the queue is

Parameters

- `symbol` – name of the queue

Typedefs

typedef atomic_ptr_t `mpsc_ptr_t`

Functions

static inline void `mpsc_init`(struct *mpsc* *q)

Initialize queue.

Parameters

- `q` – Queue to initialize or reset

ALWAYS_INLINE static void `mpsc_push`(struct *mpsc* *q, struct *mpsc_node* *n)

Push a node.

Parameters

- `q` – Queue to push the node to
- `n` – Node to push into the queue

static inline struct *mpsc_node* *`mpsc_pop`(struct *mpsc* *q)

Pop a node off of the list.

Return values

- `NULL` – When no node is available
- `node` – When node is available

struct `mpsc_node`

#include <mpsc_lockfree.h> Queue member.

struct `mpsc`

#include <mpsc_lockfree.h> MPSC Queue.

3.5.8 Single Producer Single Consumer Lock Free Queue

A *Single Producer Single Consumer Lock Free Queue (SPSC)* is a lock free atomic ring buffer based queue.

API Reference

group `spsc_lockfree`

Single Producer Single Consumer (SPSC) Lockfree Queue API.

Defines

`SPSC_INITIALIZER`(sz, buf)

Statically initialize an spsc.

Parameters

- `sz` – Size of the spsc, must be power of 2 (ex: 2, 4, 8)
- `buf` – Buffer pointer

`SPSC_DECLARE`(name, type)

Declare an anonymous struct type for an spsc.

Parameters

- `name` – Name of the spsc symbol to be provided
- `type` – Type stored in the spsc

`SPSC_DEFINE`(name, type, sz)

Define an spsc with a fixed size.

Parameters

- `name` – Name of the spsc symbol to be provided
- `type` – Type stored in the spsc
- `sz` – Size of the spsc, must be power of 2 (ex: 2, 4, 8)

`spsc_size`(spsc)

Size of the SPSC queue.

Parameters

- `spsc` – SPSC reference

`spsc_reset`(spsc)

Initialize/reset a spsc such that its empty.

Note that this is not safe to do while being used in a producer/consumer situation with multiple calling contexts (isrs/threads).

Parameters

- **spsc** – SPSC to initialize/reset

`spsc_acquire`(spsc)

Acquire an element to produce from the SPSC.

Parameters

- **spsc** – SPSC to acquire an element from for producing

Returns

A pointer to the acquired element or null if the spsc is full

`spsc_produce`(spsc)

Produce one previously acquired element to the SPSC.

This makes one element available to the consumer immediately

Parameters

- **spsc** – SPSC to produce the previously acquired element or do nothing

`spsc_produce_all`(spsc)

Produce all previously acquired elements to the SPSC.

This makes all previous acquired elements available to the consumer immediately

Parameters

- **spsc** – SPSC to produce all previously acquired elements or do nothing

`spsc_drop_all`(spsc)

Drop all previously acquired elements.

This makes all previous acquired elements available to be acquired again

Parameters

- **spsc** – SPSC to drop all previously acquired elements or do nothing

`spsc_consume`(spsc)

Consume an element from the spsc.

Parameters

- **spsc** – Spsc to consume from

Returns

Pointer to element or null if no consumable elements left

`spsc_release`(spsc)

Release a consumed element.

Parameters

- **spsc** – SPSC to release consumed element or do nothing

`spsc_release_all`(spsc)

Release all consumed elements.

Parameters

- **spsc** – SPSC to release consumed elements or do nothing

`spsc_acquirable`(spsc)

Count of acquirable in spsc.

Parameters

- **spsc** – SPSC to get item count for

spsc_consumable(spsc)
Count of consumables in spsc.

Parameters

- **spsc** – SPSC to get item count for

spsc_peek(spsc)
Peek at the first available item in queue.

Parameters

- **spsc** – Spsc to peek into

Returns
Pointer to element or null if no consumable elements left

spsc_next(spsc, item)
Peek at the next item in the queue from a given one.

Parameters

- **spsc** – SPSC to peek at
- **item** – Pointer to an item in the queue

Returns
Pointer to element or null if none left

spsc_prev(spsc, item)
Get the previous item in the queue from a given one.

Parameters

- **spsc** – SPSC to peek at
- **item** – Pointer to an item in the queue

Returns
Pointer to element or null if none left

struct spsc
Common SPSC attributes.

Warning

Not to be manipulated without the macros!

3.6 Executing Time Functions

The timing functions can be used to obtain execution time of a section of code to aid in analysis and optimization.

Please note that the timing functions may use a different timer than the default kernel timer, where the timer being used is specified by architecture, SoC or board configuration.

3.6.1 Configuration

To allow using the timing functions, CONFIG_TIMING_FUNCTIONS needs to be enabled.

3.6.2 Usage

To gather timing information:

1. Call *timing_init()* to initialize the timer.
2. Call *timing_start()* to signal the start of gathering of timing information. This usually starts the timer.
3. Call *timing_counter_get()* to mark the start of code execution.
4. Call *timing_counter_get()* to mark the end of code execution.
5. Call *timing_cycles_get()* to get the number of timer cycles between start and end of code execution.
6. Call *timing_cycles_to_ns()* with total number of cycles to convert number of cycles to nanoseconds.
7. Repeat from step 3 to gather timing information for other blocks of code.
8. Call *timing_stop()* to signal the end of gathering of timing information. This usually stops the timer.

Example

This shows an example on how to use the timing functions:

```
#include <zephyr/timing/timing.h>

void gather_timing(void)
{
    timing_t start_time, end_time;
    uint64_t total_cycles;
    uint64_t total_ns;

    timing_init();
    timing_start();

    start_time = timing_counter_get();

    code_execution_to_be_measured();

    end_time = timing_counter_get();

    total_cycles = timing_cycles_get(&start_time, &end_time);
    total_ns = timing_cycles_to_ns(total_cycles);

    timing_stop();
}
```

3.6.3 API documentation

group **timing_api**

Timing Measurement APIs.

The timing measurement APIs can be used to obtain execution time of a section of code to aid in analysis and optimization.

Please note that the timing functions may use a different timer than the default kernel timer, where the timer being used is specified by architecture, SoC or board configuration.

Functions

void **timing_init**(void)

Initialize the timing subsystem.

Perform the necessary steps to initialize the timing subsystem.

void **timing_start**(void)

Signal the start of the timing information gathering.

Signal to the timing subsystem that timing information will be gathered from this point forward.

void **timing_stop**(void)

Signal the end of the timing information gathering.

Signal to the timing subsystem that timing information is no longer being gathered from this point forward.

static inline timing_t **timing_counter_get**(void)

Return timing counter.

Returns
Timing counter.

static inline uint64_t **timing_cycles_get**(volatile timing_t *const start, volatile timing_t *const end)

Get number of cycles between `start` and `end`.

For some architectures or SoCs, the raw numbers from counter need to be scaled to obtain actual number of cycles.

Parameters

- **start** – Pointer to counter at start of a measured execution.
- **end** – Pointer to counter at stop of a measured execution.

Returns
Number of cycles between start and end.

static inline uint64_t **timing_freq_get**(void)

Get frequency of counter used (in Hz).

Returns
Frequency of counter used for timing in Hz.

static inline uint64_t **timing_cycles_to_ns**(uint64_t cycles)

Convert number of `cycles` into nanoseconds.

Parameters

- **cycles** – Number of cycles

Returns
Converted time value

static inline uint64_t **timing_cycles_to_ns_avg**(uint64_t cycles, uint32_t count)

Convert number of `cycles` into nanoseconds with averaging.

Parameters

- **cycles** – Number of cycles
- **count** – Times of accumulated cycles to average over

Returns
Converted time value

static inline uint32_t **timing_freq_get_mhz**(void)

Get frequency of counter used (in MHz).

Returns

Frequency of counter used for timing in MHz.

group **timing_api_arch**

Arch specific Timing Measurement APIs.

Implements the necessary bits to support timing measurement using architecture specific timing measurement mechanism.

Functions

void **arch_timing_init**(void)

Initialize the timing subsystem.

Perform the necessary steps to initialize the timing subsystem.

See also

timing_init()

void **arch_timing_start**(void)

Signal the start of the timing information gathering.

Signal to the timing subsystem that timing information will be gathered from this point forward.

See also

timing_start()

Note

Any call to *arch_timing_counter_get()* must be done between calls to *arch_timing_start()* and *arch_timing_stop()*, and on the same CPU core.

void **arch_timing_stop**(void)

Signal the end of the timing information gathering.

Signal to the timing subsystem that timing information is no longer being gathered from this point forward.

See also

timing_stop()

Note

Any call to *arch_timing_counter_get()* must be done between calls to *arch_timing_start()* and *arch_timing_stop()*, and on the same CPU core.

timing_t **arch_timing_counter_get**(void)

Return timing counter.

See also

timing_counter_get()

Note

Any call to *arch_timing_counter_get()* must be done between calls to *arch_timing_start()* and *arch_timing_stop()*, and on the same CPU core.

Note

Not all architectures have a timing counter with 64 bit precision. It is possible to see this value "go backwards" due to internal rollover. Timing code must be prepared to address the rollover (with platform-dependent code, e.g. by casting to a uint32_t before subtraction) or by using *arch_timing_cycles_get()* which is required to understand the distinction.

Returns

Timing counter.

uint64_t **arch_timing_cycles_get**(volatile timing_t *const start, volatile timing_t *const end)

Get number of cycles between `start` and `end`.

See also

timing_cycles_get()

Note

For some architectures, the raw numbers from counter need to be scaled to obtain actual number of cycles, or may roll over internally. This function computes a positive-definite interval between two returned cycle values.

Parameters

- **start** – Pointer to counter at start of a measured execution.
- **end** – Pointer to counter at stop of a measured execution.

Returns
Number of cycles between start and end.

uint64_t **arch_timing_freq_get**(void)
Get frequency of counter used (in Hz).

See also

timing_freq_get()

Returns
Frequency of counter used for timing in Hz.

uint64_t **arch_timing_cycles_to_ns**(uint64_t cycles)
Convert number of `cycles` into nanoseconds.

See also

timing_cycles_to_ns()

Parameters

- **cycles** – Number of cycles

Returns
Converted time value

uint64_t **arch_timing_cycles_to_ns_avg**(uint64_t cycles, uint32_t count)
Convert number of `cycles` into nanoseconds with averaging.

See also

timing_cycles_to_ns_avg()

Parameters

- **cycles** – Number of cycles
- **count** – Times of accumulated cycles to average over

Returns
Converted time value

uint32_t **arch_timing_freq_get_mhz**(void)
Get frequency of counter used (in MHz).

See also

timing_freq_get_mhz()

Returns
Frequency of counter used for timing in MHz.

group **timing_api_soc**
SoC specific Timing Measurement APIs.

Implements the necessary bits to support timing measurement using SoC specific timing measurement mechanism.

Functions

void **soc_timing_init**(void)
Initialize the timing subsystem on SoC.

Perform the necessary steps to initialize the timing subsystem.

See also

timing_init()

void **soc_timing_start**(void)
Signal the start of the timing information gathering.

Signal to the timing subsystem that timing information will be gathered from this point forward.

See also

timing_start()

void **soc_timing_stop**(void)
Signal the end of the timing information gathering.

Signal to the timing subsystem that timing information is no longer being gathered from this point forward.

See also

timing_stop()

timing_t **soc_timing_counter_get**(void)

Return timing counter.

See also

timing_counter_get()

Note

Not all SoCs have timing counters with 64 bit precision. It is possible to see this value "go backwards" due to internal rollover. Timing code must be prepared to address the rollover (with SoC dependent code, e.g. by casting to a uint32_t before subtraction) or by using *soc_timing_cycles_get()* which is required to understand the distinction.

Returns

Timing counter.

uint64_t **soc_timing_cycles_get**(volatile timing_t *const start, volatile timing_t *const end)

Get number of cycles between `start` and `end`.

See also

timing_cycles_get()

Note

The raw numbers from counter need to be scaled to obtain actual number of cycles, or may roll over internally. This function computes a positive-definite interval between two returned cycle values.

Parameters

- **start** – Pointer to counter at start of a measured execution.
- **end** – Pointer to counter at stop of a measured execution.

Returns

Number of cycles between start and end.

uint64_t **soc_timing_freq_get**(void)

Get frequency of counter used (in Hz).

See also

timing_freq_get()

Returns
Frequency of counter used for timing in Hz.

uint64_t **soc_timing_cycles_to_ns**(uint64_t cycles)
Convert number of `cycles` into nanoseconds.

See also

timing_cycles_to_ns()

Parameters

- **cycles** – Number of cycles

Returns
Converted time value

uint64_t **soc_timing_cycles_to_ns_avg**(uint64_t cycles, uint32_t count)
Convert number of `cycles` into nanoseconds with averaging.

See also

timing_cycles_to_ns_avg()

Parameters

- **cycles** – Number of cycles
- **count** – Times of accumulated cycles to average over

Returns
Converted time value

uint32_t **soc_timing_freq_get_mhz**(void)
Get frequency of counter used (in MHz).

See also

timing_freq_get_mhz()

Returns
Frequency of counter used for timing in MHz.

group `timing_api_board`

Board specific Timing Measurement APIs.

Implements the necessary bits to support timing measurement using board specific timing measurement mechanism.

Functions

void `board_timing_init`(void)

Initialize the timing subsystem.

Perform the necessary steps to initialize the timing subsystem.

See also

timing_init()

void `board_timing_start`(void)

Signal the start of the timing information gathering.

Signal to the timing subsystem that timing information will be gathered from this point forward.

See also

timing_start()

void `board_timing_stop`(void)

Signal the end of the timing information gathering.

Signal to the timing subsystem that timing information is no longer being gathered from this point forward.

See also

timing_stop()

timing_t `board_timing_counter_get`(void)

Return timing counter.

See also

timing_counter_get()

Note

Not all timing counters have 64 bit precision. It is possible to see this value "go backwards" due to internal rollover. Timing code must be prepared to address the rollover (with board dependent code, e.g. by casting to a uint32_t before subtraction) or by using *board_timing_cycles_get()* which is required to understand the distinction.

Returns
Timing counter.

uint64_t **board_timing_cycles_get**(volatile timing_t *const start, volatile timing_t *const end)

Get number of cycles between `start` and `end`.

See also

timing_cycles_get()

Note

The raw numbers from counter need to be scaled to obtain actual number of cycles, or may roll over internally. This function computes a positive-definite interval between two returned cycle values.

Parameters

- **start** – Pointer to counter at start of a measured execution.
- **end** – Pointer to counter at stop of a measured execution.

Returns
Number of cycles between start and end.

uint64_t **board_timing_freq_get**(void)

Get frequency of counter used (in Hz).

See also

timing_freq_get()

Returns
Frequency of counter used for timing in Hz.

uint64_t **board_timing_cycles_to_ns**(uint64_t cycles)

Convert number of `cycles` into nanoseconds.

See also

timing_cycles_to_ns()

Parameters

- **cycles** – Number of cycles

Returns
Converted time value

uint64_t **board_timing_cycles_to_ns_avg**(uint64_t cycles, uint32_t count)
Convert number of cycles into nanoseconds with averaging.

See also

timing_cycles_to_ns_avg()

Parameters

- **cycles** – Number of cycles
- **count** – Times of accumulated cycles to average over

Returns
Converted time value

uint32_t **board_timing_freq_get_mhz**(void)
Get frequency of counter used (in MHz).

See also

timing_freq_get_mhz()

Returns
Frequency of counter used for timing in MHz.

3.7 Object Cores

Object cores are a kernel debugging tool that can be used to both identify and perform operations on registered objects.

- *Object Core Concepts*
- *Object Core Statistics Concepts*
- *Implementation*
  - *Defining a New Object Type*
  - *Initializing a New Object Core*

  - *Walking a List of Object Cores*
  - *Object Core Statistics Querying*
- *Configuration Options*
- *API Reference*

3.7.1 Object Core Concepts

Each instance of an object embeds an object core field named *obj_core*. Objects of the same type are linked together via their respective object cores to form a singly linked list. Each object core also links to the their respective object type. Each object type contains a singly linked list linking together all the object cores of that type. Object types are also linked together via a singly linked list. Together, this can allow debugging tools to traverse all the objects in the system.

Object cores have been integrated into following kernel objects:

- *Condition Variables*
- *Events*
- *FIFOs* and *LIFOs*
- *Mailboxes*
- *Memory Slabs*
- *Message Queues*
- *Mutexes*
- *Pipes*
- *Semaphores*
- *Threads*
- *Timers*
- *System Memory Blocks*

Developers are free to integrate them if desired into other objects within their projects.

3.7.2 Object Core Statistics Concepts

A variety of kernel objects allow for the gathering and reporting of statistics. Object cores provide a uniform means to retrieve that information via object core statistics. When enabled, the object type contains a pointer to a statistics descriptor that defines the various operations that have been enabled for interfacing with the object's statistics. Additionally, the object core contains a pointer to the "raw" statistical information associated with that object. Raw data is the raw, unmanipulated data associated with the statistics. Queried data may be "raw", but it may also have been manipulated in some way by calculation (such as determining an average).

The following table indicates both what objects have been integrated into the object core statistics as well as the structures used for both "raw" and "queried" data.

Object	Raw Data Type	Query Data Type
struct mem_slab	struct mem_slab_info	struct sys_memory_stats
struct sys_mem_blocks	struct sys_mem_blocks_info	struct sys_memory_stats
struct k_thread	struct k_cycle_stats	struct k_thread_runtime_stats
struct _cpu	struct k_cycle_stats	struct k_thread_runtime_stats
struct z_kernel	struct k_cycle_stats[num CPUs]	struct k_thread_runtime_stats

3.7.3 Implementation

Defining a New Object Type

An object type is defined using a global variable of type *k_obj_type*. It must be initialized before any objects of that type are initialized. The following code shows how a new object type can be initialized for use with object cores and object core statistics.

```
/* Unique object type ID */

#define K_OBJ_TYPE_MY_NEW_TYPE  K_OBJ_TYPE_ID_GEN("UNIQ")
struct k_obj_type  my_obj_type;

struct my_obj_type_raw_info {
    ...
};

struct my_obj_type_query_stats {
    ...
};

struct my_new_obj {
    ...
    struct k_obj_core obj_core;
    struct my_obj_type_raw_info  info;
};

struct k_obj_core_stats_desc my_obj_type_stats_desc = {
    .raw_size = sizeof(struct my_obj_type_raw_stats),
    .query_size = sizeof(struct my_obj_type_query_stats),
    .raw = my_obj_type_stats_raw,
    .query = my_obj_type_stats_query,
    .reset = my_obj_type_stats_reset,
    .disable = NULL,    /* Stats gathering is always on */
    .enable = NULL,     /* Stats gathering is always on */
};

void my_obj_type_init(void)
{
    z_obj_type_init(&my_obj_type, K_OBJ_TYPE_MY_NEW_TYPE,
                    offsetof(struct my_new_obj, obj_core);
    k_obj_type_stats_init(&my_obj_type, &my_obj_type_stats_desc);
}
```

Initializing a New Object Core

Kernel objects that have already been integrated into the object core framework automatically have their object cores initialized when the object is initialized. However, developers that wish

to add their own objects into the framework need to both initialize the object core and link it. The following code builds on the example above and initializes the object core.

```
void my_new_obj_init(struct my_new_obj *new_obj)
{
    ...
    k_obj_core_init(K_OBJ_CORE(new_obj), &my_obj_type);
    k_obj_core_link(K_OBJ_CORE(new_obj));
    k_obj_core_stats_register(K_OBJ_CORE(new_obj), &new_obj->raw_stats,
                              sizeof(struct my_obj_type_raw_info));
}
```

Walking a List of Object Cores

Two routines exist for walking the list of object cores linked to an object type. These are *k_obj_type_walk_locked()* and *k_obj_type_walk_unlocked()*. The following code builds upon the example above and prints the addresses of all the objects of that new object type.

```
int walk_op(struct k_obj_core *obj_core, void *data)
{
    uint8_t *ptr;

    ptr = obj_core;
    ptr -= obj_core->type->obj_core_offset;

    printk("%p\n", ptr);

    return 0;
}

void print_object_addresses(void)
{
    struct k_obj_type *obj_type;

    /* Find the object type */

    obj_type = k_obj_type_find(K_OBJ_TYPE_MY_NEW_TYPE);

    /* Walk the list of objects */

    k_obj_type_walk_unlocked(obj_type, walk_op, NULL);
}
```

Object Core Statistics Querying

The following code builds on the examples above and shows how an object integrated into the object core statistics framework can both retrieve queried data and reset the stats associated with the object.

```
struct my_new_obj my_obj;

...

void my_func(void)
{
    struct my_obj_type_query_stats  my_stats;
    int  status;
```

(continues on next page)

(continued from previous page)

```
    my_obj_type_init(&my_obj);

    ...

    status = k_obj_core_stats_query(K_OBJ_CORE(&my_obj),
                                    &my_stats, sizeof(my_stats));
    if (status != 0) {
        /* Failed to get stats */
        ...
    } else {
        k_obj_core_stats_reset(K_OBJ_CORE(&my_obj));
    }

    ...
}
```

3.7.4 Configuration Options

Related configuration options:

- CONFIG_OBJ_CORE
- CONFIG_OBJ_CORE_CONDVAR
- CONFIG_OBJ_CORE_EVENT
- CONFIG_OBJ_CORE_FIFO
- CONFIG_OBJ_CORE_LIFO
- CONFIG_OBJ_CORE_MAILBOX
- CONFIG_OBJ_CORE_MEM_SLAB
- CONFIG_OBJ_CORE_MSGQ
- CONFIG_OBJ_CORE_MUTEX
- CONFIG_OBJ_CORE_PIPE
- CONFIG_OBJ_CORE_SEM
- CONFIG_OBJ_CORE_STACK
- CONFIG_OBJ_CORE_THREAD
- CONFIG_OBJ_CORE_TIMER
- CONFIG_OBJ_CORE_SYS_MEM_BLOCKS
- CONFIG_OBJ_CORE_STATS
- CONFIG_OBJ_CORE_STATS_MEM_SLAB
- CONFIG_OBJ_CORE_STATS_THREAD
- CONFIG_OBJ_CORE_STATS_SYSTEM
- CONFIG_OBJ_CORE_STATS_SYS_MEM_BLOCKS

3.7.5 API Reference

group `obj_core_apis`

Defines

K_OBJ_CORE(kobj)
: Convert kernel object pointer into its object core pointer.

K_OBJ_TYPE_ID_GEN(s)
: Generate new object type IDs based on a 4 letter string.

K_OBJ_TYPE_CONDVAR_ID
: Condition variable object type.

K_OBJ_TYPE_CPU_ID
: CPU object type.

K_OBJ_TYPE_EVENT_ID
: Event object type.

K_OBJ_TYPE_FIFO_ID
: FIFO object type.

K_OBJ_TYPE_KERNEL_ID
: Kernel object type.

K_OBJ_TYPE_LIFO_ID
: LIFO object type.

K_OBJ_TYPE_MEM_BLOCK_ID
: Memory block object type.

K_OBJ_TYPE_MBOX_ID
: Mailbox object type.

K_OBJ_TYPE_MEM_SLAB_ID
: Memory slab object type.

K_OBJ_TYPE_MSGQ_ID
: Message queue object type.

K_OBJ_TYPE_MUTEX_ID
: Mutex object type.

K_OBJ_TYPE_PIPE_ID
: Pipe object type.

K_OBJ_TYPE_SEM_ID
: Semaphore object type.

K_OBJ_TYPE_STACK_ID
: Stack object type.

`K_OBJ_TYPE_THREAD_ID`
: Thread object type.

`K_OBJ_TYPE_TIMER_ID`
: Timer object type.

Functions

struct *k_obj_type* *`k_obj_type_find`(uint32_t type_id)

Find a specific object type by ID.

Given an object type ID, this function searches for the object type that is associated with the specified type ID *type_id*.

Parameters

- `type_id` – Type ID associated with object type

Return values

- `NULL` – if object type not found
- `Pointer` – to object type if found

int `k_obj_type_walk_locked`(struct *k_obj_type* *type, int (*func)(struct *k_obj_core**, void*), void *data)

Walk the object type's list of object cores.

This function takes a global spinlock and walks the object type's list of object cores and invokes the callback function on each element while holding that lock. Although this will ensure that the list is not modified, one can expect a significant penalty in terms of performance and latency.

The callback function shall either return non-zero to stop further walking, or it shall return 0 to continue walking.

Parameters

- `type` – Pointer to the object type
- `func` – Callback to invoke on each object core of the object type
- `data` – Custom data passed to the callback

Return values

`non-zero` – if walk is terminated by the callback; otherwise 0

int `k_obj_type_walk_unlocked`(struct *k_obj_type* *type, int (*func)(struct *k_obj_core**, void*), void *data)

Walk the object type's list of object cores.

This function is similar to *k_obj_type_walk_locked()* except that it walks the list without obtaining the global spinlock. No synchronization is provided here. Mutation of the list of objects while this function is in progress must be prevented at the application layer, otherwise undefined/unreliable behavior, corruption and/or crashes may result.

The callback function shall either return non-zero to stop further walking, or it shall return 0 to continue walking.

Parameters

- `type` – Pointer to the object type
- `func` – Callback to invoke on each object core of the object type

- **data** – Custom data passed to the callback

Return values

non-zero – if walk is terminated by the callback; otherwise 0

void **k_obj_core_init**(struct *k_obj_core* *obj_core, struct *k_obj_type* *type)

Initialize the core of the kernel object.

Initializing the kernel object core associates it with the specified kernel object type.

Parameters

- **obj_core** – Pointer to the kernel object to initialize
- **type** – Pointer to the kernel object type

void **k_obj_core_link**(struct *k_obj_core* *obj_core)

Link the kernel object to the kernel object type list.

A kernel object can be optionally linked into the kernel object type's list of objects. A kernel object must have been initialized before it can be linked. Linked kernel objects can be traversed and have information extracted from them by system tools.

Parameters

- **obj_core** – Pointer to the kernel object

void **k_obj_core_init_and_link**(struct *k_obj_core* *obj_core, struct *k_obj_type* *type)

Automatically link the kernel object after initializing it.

A useful wrapper to both initialize the core of the kernel object and automatically link it into the kernel object type's list of objects.

Parameters

- **obj_core** – Pointer to the kernel object to initialize
- **type** – Pointer to the kernel object type

void **k_obj_core_unlink**(struct *k_obj_core* *obj_core)

Unlink the kernel object from the kernel object type list.

Kernel objects can be unlinked from their respective kernel object type lists. If on a list, it must be done at the end of the kernel object's life cycle.

Parameters

- **obj_core** – Pointer to the kernel object

struct **k_obj_core_stats_desc**

#include <obj_core.h> Object core statistics descriptor.

Public Members

size_t **raw_size**

Internal representation stats buffer size.

size_t **query_size**

Stats buffer size used for reporting.

int (***raw**)(struct *k_obj_core* *obj_core, void *stats)

Function pointer to retrieve internal representation of stats.

int (*query)(struct *k_obj_core* *obj_core, void *stats)
Function pointer to retrieve reported statistics.

int (*reset)(struct *k_obj_core* *obj_core)
Function pointer to reset object's statistics.

int (*disable)(struct *k_obj_core* *obj_core)
Function pointer to disable object's statistics gathering.

int (*enable)(struct *k_obj_core* *obj_core)
Function pointer to enable object's statistics gathering.

struct k_obj_type
#include <*obj_core.h*> Object type structure.

Public Members

sys_snode_t node
Node within list of object types.

sys_slist_t list
List of objects of this object type.

uint32_t id
Unique type ID.

size_t obj_core_offset
Offset to obj_core field.

struct k_obj_core
#include <*obj_core.h*> Object core structure.

Public Members

sys_snode_t node
Object node within object type's list.

struct *k_obj_type* *type
Object type to which object belongs.

group obj_core_stats_apis

Functions

int `k_obj_core_stats_register`(struct *k_obj_core* *obj_core, void *stats, size_t stats_len)

Register kernel object for gathering statistics.

Before a kernel object can gather statistics, it must be registered to do so. Registering will also automatically enable the kernel object to gather its statistics.

Parameters

- `obj_core` – Pointer to kernel object core
- `stats` – Pointer to raw kernel statistics
- `stats_len` – Size of raw kernel statistics buffer

Return values

- `0` – on success
- `-errno` – on failure

int `k_obj_core_stats_deregister`(struct *k_obj_core* *obj_core)

Deregister kernel object from gathering statistics.

Deregistering a kernel object core from gathering statistics prevents it from gathering any more statistics. It is expected to be invoked at the end of a kernel object's life cycle.

Parameters

- `obj_core` – Pointer to kernel object core

Return values

- `0` – on success
- `-errno` – on failure

int `k_obj_core_stats_raw`(struct *k_obj_core* *obj_core, void *stats, size_t stats_len)

Retrieve the raw statistics associated with the kernel object.

This function copies the raw statistics associated with the kernel object core specified by *obj_core* into the buffer *stats*. Note that the size of the buffer (*stats_len*) must match the size specified by the kernel object type's statistics descriptor.

Parameters

- `obj_core` – Pointer to kernel object core
- `stats` – Pointer to memory buffer into which to copy raw stats
- `stats_len` – Length of the memory buffer

Return values

- `0` – on success
- `-errno` – on failure

int `k_obj_core_stats_query`(struct *k_obj_core* *obj_core, void *stats, size_t stats_len)

Retrieve the statistics associated with the kernel object.

This function copies the statistics associated with the kernel object core specified by *obj_core* into the buffer *stats*. Unlike the raw statistics this may report calculated values such as averages. Note that the size of the buffer (*stats_len*) must match the size specified by the kernel object type's statistics descriptor.

Parameters

- `obj_core` – Pointer to kernel object core
- `stats` – Pointer to memory buffer into which to copy the queried stats
- `stats_len` – Length of the memory buffer

Return values

- `0` – on success
- `-errno` – on failure

int **k_obj_core_stats_reset**(struct *k_obj_core* *obj_core)

Reset the stats associated with the kernel object.

This function resets the statistics associated with the kernel object core specified by *obj_core*.

Parameters

- `obj_core` – Pointer to kernel object core

Return values

- `0` – on success
- `-errno` – on failure

int **k_obj_core_stats_disable**(struct *k_obj_core* *obj_core)

Stop gathering the stats associated with the kernel object.

This function temporarily stops the gathering of statistics associated with the kernel object core specified by *obj_core*. The gathering of statistics can be resumed by invoking :c:func :k_obj_core_stats_enable.

Parameters

- `obj_core` – Pointer to kernel object core

Return values

- `0` – on success
- `-errno` – on failure

int **k_obj_core_stats_enable**(struct *k_obj_core* *obj_core)

Reset the stats associated with the kernel object.

This function resumes the gathering of statistics associated with the kernel object core specified by *obj_core*.

Parameters

- `obj_core` – Pointer to kernel object core

Return values

- `0` – on success
- `-errno` – on failure

3.8 Time Utilities

3.8.1 Overview

Uptime in Zephyr is based on the a tick counter. With the default `CONFIG_TICKLESS_KERNEL` this counter advances at a nominally constant rate from zero at the instant the system started. The POSIX equivalent to this counter is something like `CLOCK_MONOTONIC` or, in Linux, `CLOCK_MONOTONIC_RAW`. *`k_uptime_get()`* provides a millisecond representation of this time.

Applications often need to correlate the Zephyr internal time with external time scales used in daily life, such as local time or Coordinated Universal Time. These systems interpret time

in different ways and may have discontinuities due to leap seconds and local time offsets like daylight saving time.

Because of these discontinuities, as well as significant inaccuracies in the clocks underlying the cycle counter, the offset between time estimated from the Zephyr clock and the actual time in a "real" civil time scale is not constant and can vary widely over the runtime of a Zephyr application.

The time utilities API supports:

- *converting between time representations*
- *synchronizing and aligning time scales*

For terminology and concepts that support these functions see *Concepts Underlying Time Support in Zephyr*.

3.8.2 Time Utility APIs

Representation Transformation

Time scale instants can be represented in multiple ways including:

- Seconds since an epoch. POSIX representations of time in this form include `time_t` and `struct timespec`, which are generally interpreted as a representation of "UNIX Time".
- Calendar time as a year, month, day, hour, minutes, and seconds relative to an epoch. POSIX representations of time in this form include `struct tm`.

Keep in mind that these are simply time representations that must be interpreted relative to a time scale which may be local time, UTC, or some other continuous or discontinuous scale.

Some necessary transformations are available in standard C library routines. For example, `time_t` measuring seconds since the POSIX EPOCH is converted to `struct tm` representing calendar time with gmtime(). Sub-second timestamps like `struct timespec` can also use this to produce the calendar time representation and deal with sub-second offsets separately.

The inverse transformation is not standardized: APIs like `mktime()` expect information about time zones. Zephyr provides this transformation with *`timeutil_timegm()`* and *`timeutil_timegm64()`*.

group **timeutil_repr_apis**

Functions

int64_t **timeutil_timegm64**(const struct *tm* *tm)

Convert broken-down time to a POSIX epoch offset in seconds.

See also

http://man7.org/linux/man-pages/man3/timegm.3.html

Parameters

- **tm** – pointer to broken down time.

Returns

the corresponding time in the POSIX epoch time scale.

time_t **timeutil_timegm**(const struct *tm* *tm)

Convert broken-down time to a POSIX epoch offset in seconds.

See also

http://man7.org/linux/man-pages/man3/timegm.3.html

Parameters

- **tm** – pointer to broken down time.

Returns

the corresponding time in the POSIX epoch time scale. If the time cannot be represented then (time_t)-1 is returned and `errno` is set to `ERANGE`.

Time Scale Synchronization

There are several factors that affect synchronizing time scales:

- The rate of discrete instant representation change. For example Zephyr uptime is tracked in ticks which advance at events that nominally occur at `CONFIG_SYS_CLOCK_TICKS_PER_SEC` Hertz, while an external time source may provide data in whole or fractional seconds (e.g. microseconds).
- The absolute offset required to align the two scales at a single instant.
- The relative error between observable instants in each scale, required to align multiple instants consistently. For example a reference clock that's conditioned by a 1-pulse-per-second GPS signal will be much more accurate than a Zephyr system clock driven by a RC oscillator with a +/- 250 ppm error.

Synchronization or alignment between time scales is done with a multi-step process:

- An instant in a time scale is represented by an (unsigned) 64-bit integer, assumed to advance at a fixed nominal rate.
- *timeutil_sync_config* records the nominal rates of a reference time scale/source (e.g. TAI) and a local time source (e.g. *k_uptime_ticks()*).
- *timeutil_sync_instant* records the representation of a single instant in both the reference and local time scales.
- *timeutil_sync_state* provides storage for an initial instant, a recently received second observation, and a skew that can adjust for relative errors in the actual rate of each time scale.
- *timeutil_sync_ref_from_local()* and *timeutil_sync_local_from_ref()* convert instants in one time scale to another taking into account skew that can be estimated from the two instances stored in the state structure by *timeutil_sync_estimate_skew()*.

group **timeutil_sync_apis**

Functions

int **timeutil_sync_state_update**(struct *timeutil_sync_state* *tsp, const struct *timeutil_sync_instant* *inst)

Record a new instant in the time synchronization state.

Note that this updates only the latest persisted instant. The skew is not adjusted automatically.

Parameters

- `tsp` – pointer to a *timeutil_sync_state* object.
- `inst` – the new instant to be recorded. This becomes the base instant if there is no base instant, otherwise the value must be strictly after the base instant in both the reference and local time scales.

Return values

- `0` – if installation succeeded in providing a new base
- `1` – if installation provided a new latest instant
- `-EINVAL` – if the new instant is not compatible with the base instant

int **timeutil_sync_state_set_skew**(struct *timeutil_sync_state* *tsp, float skew, const struct *timeutil_sync_instant* *base)

Update the state with a new skew and possibly base value.

Set the skew from a value retrieved from persistent storage, or calculated based on recent skew estimations including from *timeutil_sync_estimate_skew()*.

Optionally update the base timestamp. If the base is replaced the latest instant will be cleared until *timeutil_sync_state_update()* is invoked.

Parameters

- `tsp` – pointer to a time synchronization state.
- `skew` – the skew to be used. The value must be positive and shouldn't be too far away from 1.
- `base` – optional new base to be set. If provided this becomes the base timestamp that will be used along with skew to convert between reference and local timescale instants. Setting the base clears the captured latest value.

Returns
0 if skew was updated

Returns
-EINVAL if skew was not valid

float **timeutil_sync_estimate_skew**(const struct *timeutil_sync_state* *tsp)

Estimate the skew based on current state.

Using the base and latest syncpoints from the state determine the skew of the local clock relative to the reference clock. See *timeutil_sync_state::skew*.

Parameters

- `tsp` – pointer to a time synchronization state. The base and latest syncpoints must be present and the latest syncpoint must be after the base point in the local time scale.

Returns
the estimated skew, or zero if skew could not be estimated.

int **timeutil_sync_ref_from_local**(const struct *timeutil_sync_state* *tsp, uint64_t local, uint64_t *refp)

Interpolate a reference timescale instant from a local instant.

Parameters

- **tsp** – pointer to a time synchronization state. This must have a base and a skew installed.
- **local** – an instant measured in the local timescale. This may be before or after the base instant.
- **refp** – where the corresponding instant in the reference timescale should be stored. A negative interpolated reference time produces an error. If interpolation fails the referenced object is not modified.

Return values

- **0** – if interpolated using a skew of 1
- **1** – if interpolated using a skew not equal to 1
- **-EINVAL** –
  - the times synchronization state is not adequately initialized
  - refp is null
- **-ERANGE** – the interpolated reference time would be negative

int **timeutil_sync_local_from_ref**(const struct *timeutil_sync_state* *tsp, uint64_t ref, int64_t *localp)

Interpolate a local timescale instant from a reference instant.

Parameters

- **tsp** – pointer to a time synchronization state. This must have a base and a skew installed.
- **ref** – an instant measured in the reference timescale. This may be before or after the base instant.
- **localp** – where the corresponding instant in the local timescale should be stored. An interpolated value before local time 0 is provided without error. If interpolation fails the referenced object is not modified.

Return values

- **0** – if successful with a skew of 1
- **1** – if successful with a skew not equal to 1
- **-EINVAL** –
  - the time synchronization state is not adequately initialized
  - refp is null

int32_t **timeutil_sync_skew_to_ppb**(float skew)

Convert from a skew to an error in parts-per-billion.

A skew of 1.0 has zero error. A skew less than 1 has a positive error (clock is faster than it should be). A skew greater than one has a negative error (clock is slower than it should be).

Note that due to the limited precision of float compared with double the smallest error that can be represented is about 120 ppb. A “precise” time source may have error on the order of 2000 ppb.

A skew greater than 3.14748 may underflow the 32-bit representation; this represents a clock running at less than 1/3 its nominal rate.

Returns

skew error represented as parts-per-billion, or INT32_MIN if the skew cannot be represented in the return type.

struct **timeutil_sync_config**

#include <*timeutil.h*> Immutable state for synchronizing two clocks.

Values required to convert durations between two time scales.

> **Note**
>
> The accuracy of the translation and calculated skew between sources depends on the resolution of these frequencies. A reference frequency with microsecond or nanosecond resolution would produce the most accurate tracking when the local reference is the Zephyr tick counter. A reference source like an RTC chip with 1 Hz resolution requires a much larger interval between sampled instants to detect relative clock drift.

Public Members

uint32_t **ref_Hz**

The nominal instance counter rate in Hz.

This value is assumed to be precise, but may drift depending on the reference clock source.

The value must be positive.

uint32_t **local_Hz**

The nominal local counter rate in Hz.

This value is assumed to be inaccurate but reasonably stable. For a local clock driven by a crystal oscillator an error of 25 ppm is common; for an RC oscillator larger errors should be expected. The timeutil_sync infrastructure can calculate the skew between the local and reference clocks and apply it when converting between time scales.

The value must be positive.

struct **timeutil_sync_instant**

#include <*timeutil.h*> Representation of an instant in two time scales.

Capturing the same instant in two time scales provides a registration point that can be used to convert between those time scales.

Public Members

uint64_t **ref**

An instant in the reference time scale.

This must never be zero in an initialized *timeutil_sync_instant* object.

uint64_t **local**

The corresponding instance in the local time scale.

This may be zero in a valid *timeutil_sync_instant* object.

struct **timeutil_sync_state**

#include <timeutil.h> State required to convert instants between time scales.

This state in conjunction with functions that manipulate it capture the offset information necessary to convert between two timescales along with information that corrects for skew due to inaccuracies in clock rates.

State objects should be zero-initialized before use.

Public Members

const struct *timeutil_sync_config* ***cfg**

Pointer to reference and local rate information.

struct *timeutil_sync_instant* **base**

The base instant in both time scales.

struct *timeutil_sync_instant* **latest**

The most recent instant in both time scales.

This is captured here to provide data for skew calculation.

float **skew**

The scale factor used to correct for clock skew.

The nominal rate for the local counter is assumed to be inaccurate but stable, i.e. it will generally be some parts-per-million faster or slower than specified.

A duration in observed local clock ticks must be multiplied by this value to produce a duration in ticks of a clock operating at the nominal local rate.

A zero value indicates that the skew has not been initialized. If the value is zero when *base* is initialized the skew will be set to 1. Otherwise the skew is assigned through *timeutil_sync_state_set_skew()*.

3.8.3 Concepts Underlying Time Support in Zephyr

Terms from ISO/TC 154/WG 5 N0038 (ISO/WD 8601-1) and elsewhere:

- A *time axis* is a representation of time as an ordered sequence of instants.
- A *time scale* is a way of representing an instant relative to an origin that serves as the epoch.
- A time scale is *monotonic* (increasing) if the representation of successive time instants never decreases in value.
- A time scale is *continuous* if the representation has no abrupt changes in value, e.g. jumping forward or back when going between successive instants.
- Civil time generally refers to time scales that legally defined by civil authorities, like local governments, often to align local midnight to solar time.

Relevant Time Scales

International Atomic Time (TAI) is a time scale based on averaging clocks that count in SI seconds. TAI is a monotonic and continuous time scale.

Universal Time (UT) is a time scale based on Earth's rotation. UT is a discontinuous time scale as it requires occasional adjustments (leap seconds) to maintain alignment to changes in Earth's rotation. Thus the difference between TAI and UT varies over time. There are several variants of UT, with UTC being the most common.

UT times are independent of location. UT is the basis for Standard Time (or "local time") which is the time at a particular location. Standard time has a fixed offset from UT at any given instant, primarily influenced by longitude, but the offset may be adjusted ("daylight saving time") to align standard time to the local solar time. In a sense local time is "more discontinuous" than UT.

POSIX Time is a time scale that counts seconds since the "POSIX epoch" at 1970-01-01T00:00:00Z (i.e. the start of 1970 UTC). UNIX Time is an extension of POSIX time using negative values to represent times before the POSIX epoch. Both of these scales assume that every day has exactly 86400 seconds. In normal use instants in these scales correspond to times in the UTC scale, so they inherit the discontinuity.

The continuous analogue is UNIX Leap Time which is UNIX time plus all leap-second corrections added after the POSIX epoch (when TAI-UTC was 8 s).

Example of Time Scale Differences A positive leap second was introduced at the end of 2016, increasing the difference between TAI and UTC from 36 seconds to 37 seconds. There was no leap second introduced at the end of 1999, when the difference between TAI and UTC was only 32 seconds. The following table shows relevant civil and epoch times in several scales:

UTC Date	UNIX time	TAI Date	TAI-UTC	UNIX Leap Time
1970-01-01T00:00:00Z	0	1970-01-01T00:00:08	+8	0
1999-12-31T23:59:28Z	946684768	2000-01-01T00:00:00	+32	946684792
1999-12-31T23:59:59Z	946684799	2000-01-01T00:00:31	+32	946684823
2000-01-01T00:00:00Z	946684800	2000-01-01T00:00:32	+32	946684824
2016-12-31T23:59:59Z	1483228799	2017-01-01T00:00:35	+36	1483228827
2016-12-31T23:59:60Z	undefined	2017-01-01T00:00:36	+36	1483228828
2017-01-01T00:00:00Z	1483228800	2017-01-01T00:00:37	+37	1483228829

Functional Requirements The Zephyr tick counter has no concept of leap seconds or standard time offsets and is a continuous time scale. However it can be relatively inaccurate, with drifts as much as three minutes per hour (assuming an RC timer with 5% tolerance).

There are two stages required to support conversion between Zephyr time and common human time scales:

- Translation between the continuous but inaccurate Zephyr time scale and an accurate external stable time scale;
- Translation between the stable time scale and the (possibly discontinuous) civil time scale.

The API around *timeutil_sync_state_update()* supports the first step of converting between continuous time scales.

The second step requires external information including schedules of leap seconds and local time offset changes. This may be best provided by an external library, and is not currently part of the time utility APIs.

Selecting an External Source and Time Scale If an application requires civil time accuracy within several seconds then UTC could be used as the stable time source. However, if the external source adjusts to a leap second there will be a discontinuity: the elapsed time between two observations taken at 1 Hz is not equal to the numeric difference between their timestamps.

For precise activities a continuous scale that is independent of local and solar adjustments simplifies things considerably. Suitable continuous scales include:

- GPS time: epoch of 1980-01-06T00:00:00Z, continuous following TAI with an offset of TAI-GPS=19 s.
- Bluetooth Mesh time: epoch of 2000-01-01T00:00:00Z, continuous following TAI with an offset of -32.
- UNIX Leap Time: epoch of 1970-01-01T00:00:00Z, continuous following TAI with an offset of -8.

Because C and Zephyr library functions support conversion between integral and calendar time representations using the UNIX epoch, UNIX Leap Time is an ideal choice for the external time scale.

The mechanism used to populate synchronization points is not relevant: it may involve reading from a local high-precision RTC peripheral, exchanging packets over a network using a protocol like NTP or PTP, or processing NMEA messages received a GPS with or without a 1pps signal.

3.9 Utilities

This page contains reference documentation for <sys/util.h>, which provides miscellaneous utility functions and macros.

group sys-util

Since
2.4

Version
0.1.0

Defines

POINTER_TO_UINT(x)
Cast x, a pointer, to an unsigned integer.

UINT_TO_POINTER(x)
Cast x, an unsigned integer, to a void*.

POINTER_TO_INT(x)
Cast x, a pointer, to a signed integer.

INT_TO_POINTER(x)
Cast x, a signed integer, to a void*.

BITS_PER_LONG
Number of bits in a long int.

BITS_PER_LONG_LONG
Number of bits in a long long int.

GENMASK(h, l)
Create a contiguous bitmask starting at bit position l and ending at position h.

GENMASK64(h, l)
Create a contiguous 64-bit bitmask starting at bit position l and ending at position h.

LSB_GET(value)

Extract the Least Significant Bit from value.

FIELD_GET(mask, value)

Extract a bitfield element from value corresponding to the field mask mask.

FIELD_PREP(mask, value)

Prepare a bitfield element using value with mask representing its field position and width.

The result should be combined with other fields using a logical OR.

ZERO_OR_COMPILE_ERROR(cond)

0 if cond is true-ish; causes a compile error otherwise.

IS_ARRAY(array)

Zero if array has an array type, a compile error otherwise.

This macro is available only from C, not C++.

ARRAY_SIZE(array)

Number of elements in the given array.

In C++, due to language limitations, this will accept as array any type that implements operator[]. The results may not be particularly meaningful in this case.

In C, passing a pointer as array causes a compile error.

IS_ARRAY_ELEMENT(array, ptr)

Whether ptr is an element of array.

This macro can be seen as a slightly stricter version of *PART_OF_ARRAY* in that it also ensures that ptr is aligned to an array-element boundary of array.

In C, passing a pointer as array causes a compile error.

Parameters

- **array** – the array in question
- **ptr** – the pointer to check

Returns

1 if ptr is part of array, 0 otherwise

ARRAY_INDEX(array, ptr)

Index of ptr within array.

With CONFIG_ASSERT=y, this macro will trigger a runtime assertion when ptr does not fall into the range of array or when ptr is not aligned to an array-element boundary of array.

In C, passing a pointer as array causes a compile error.

Parameters

- **array** – the array in question
- **ptr** – pointer to an element of array

Returns

the array index of ptr within array, on success

PART_OF_ARRAY(array, ptr)

Check if a pointer ptr lies within array.

In C but not C++, this causes a compile error if array is not an array (e.g. if ptr and array are mixed up).

Parameters

- **array** – an array
- **ptr** – a pointer

Returns
1 if ptr is part of array, 0 otherwise

ARRAY_INDEX_FLOOR(array, ptr)

Array-index of ptr within array, rounded down.

This macro behaves much like *ARRAY_INDEX* with the notable difference that it accepts any ptr in the range of array rather than exclusively a ptr aligned to an array-element boundary of array.

With CONFIG_ASSERT=y, this macro will trigger a runtime assertion when ptr does not fall into the range of array.

In C, passing a pointer as array causes a compile error.

Parameters

- **array** – the array in question
- **ptr** – pointer to an element of array

Returns
the array index of ptr within array, on success

ARRAY_FOR_EACH(array, idx)

Iterate over members of an array using an index variable.

Parameters

- **array** – the array in question
- **idx** – name of array index variable

ARRAY_FOR_EACH_PTR(array, ptr)

Iterate over members of an array using a pointer.

Parameters

- **array** – the array in question
- **ptr** – pointer to an element of array

SAME_TYPE(a, b)

Validate if two entities have a compatible type.

Parameters

- **a** – the first entity to be compared
- **b** – the second entity to be compared

Returns
1 if the two elements are compatible, 0 if they are not

CONTAINER_OF_VALIDATE(ptr, type, field)

Validate CONTAINER_OF parameters, only applies to C mode.

CONTAINER_OF(ptr, type, field)

Get a pointer to a structure containing the element.

Example:

```
struct foo {
   int bar;
};

struct foo my_foo;
int *ptr = &my_foo.bar;

struct foo *container = CONTAINER_OF(ptr, struct foo, bar);
```

Above, `container` points at `my_foo`.

Parameters

- **ptr** – pointer to a structure element
- **type** – name of the type that `ptr` is an element of
- **field** – the name of the field within the struct `ptr` points to

Returns
a pointer to the structure that contains `ptr`

`SIZEOF_FIELD`(type, member)

Report the size of a struct field in bytes.

Parameters

- **type** – The structure containing the field of interest.
- **member** – The field to return the size of.

Returns
The field size.

`CONCAT`(...)

Concatenate input arguments.

Concatenate provided tokens into a combined token during the preprocessor pass. This can be used to, for ex., build an identifier out of multiple parts, where one of those parts may be, for ex, a number, another macro, or a macro argument.

Parameters

- `...` – Tokens to concatencate

Returns
Concatenated token.

`IS_ALIGNED`(ptr, align)

Check if `ptr` is aligned to `align` alignment.

`ROUND_UP`(x, align)

Value of `x` rounded up to the next multiple of `align`.

`ROUND_DOWN`(x, align)

Value of `x` rounded down to the previous multiple of `align`.

`WB_UP`(x)

Value of `x` rounded up to the next word boundary.

`WB_DN`(x)

Value of `x` rounded down to the previous word boundary.

`DIV_ROUND_UP`(n, d)

Divide and round up.

Example:

```
DIV_ROUND_UP(1, 2); // 1
DIV_ROUND_UP(3, 2); // 2
```

Parameters

- **n** – Numerator.
- **d** – Denominator.

Returns
The result of n / d, rounded up.

`DIV_ROUND_CLOSEST`(n, d)

Divide and round to the nearest integer.

Example:

```
DIV_ROUND_CLOSEST(5, 2); // 3
DIV_ROUND_CLOSEST(5, -2); // -3
DIV_ROUND_CLOSEST(5, 3); // 2
```

Parameters

- **n** – Numerator.
- **d** – Denominator.

Returns
The result of n / d, rounded to the nearest integer.

`ceiling_fraction`(numerator, divider)

Ceiling function applied to `numerator` / `divider` as a fraction.

Deprecated:
Use *DIV_ROUND_UP()* instead.

`MAX`(a, b)

Obtain the maximum of two values.

Note

Arguments are evaluated twice. Use Z_MAX for a GCC-only, single evaluation version

Parameters

- **a** – First value.
- **b** – Second value.

Returns
Maximum value of a and b.

`MIN`(a, b)

Obtain the minimum of two values.

> **Note**
>
> Arguments are evaluated twice. Use Z_MIN for a GCC-only, single evaluation version

Parameters

- `a` – First value.
- `b` – Second value.

Returns
Minimum value of a and b.

`CLAMP`(val, low, high)

Clamp a value to a given range.

> **Note**
>
> Arguments are evaluated multiple times. Use Z_CLAMP for a GCC-only, single evaluation version.

Parameters

- `val` – Value to be clamped.
- `low` – Lowest allowed value (inclusive).
- `high` – Highest allowed value (inclusive).

Returns
Clamped value.

`IN_RANGE`(val, min, max)

Checks if a value is within range.

> **Note**
>
> `val` is evaluated twice.

Parameters

- `val` – Value to be checked.
- `min` – Lower bound (inclusive).
- `max` – Upper bound (inclusive).

Return values

- `true` – If value is within range
- `false` – If the value is not within range

`LOG2`(x)

Compute log2(x)

> **Note**
>
> This macro expands its argument multiple times (to permit use in constant expressions), which must not have side effects.

Parameters

- x – An unsigned integral value to compute logarithm of (positive only)

Returns

log2(x) when 1 <= x <= max(x), -1 when x < 1

`LOG2CEIL(x)`

Compute ceil(log2(x))

> **Note**
>
> This macro expands its argument multiple times (to permit use in constant expressions), which must not have side effects.

Parameters

- x – An unsigned integral value

Returns

ceil(log2(x)) when 1 <= x <= max(type(x)), 0 when x < 1

`NHPOT(x)`

Compute next highest power of two.

Equivalent to 2^ceil(log2(x))

> **Note**
>
> This macro expands its argument multiple times (to permit use in constant expressions), which must not have side effects.

Parameters

- x – An unsigned integral value

Returns

2^ceil(log2(x)) or 0 if 2^ceil(log2(x)) would saturate 64-bits

`KB(x)`

Number of bytes in x kibibytes.

`MB(x)`

Number of bytes in x mebibytes.

`GB(x)`

Number of bytes in x gibibytes.

`KHZ(x)`

Number of Hz in x kHz.

`MHZ(x)`

Number of Hz in x MHz.

WAIT_FOR(expr, timeout, delay_stmt)

Wait for an expression to return true with a timeout.

Spin on an expression with a timeout and optional delay between iterations

Commonly needed when waiting on hardware to complete an asynchronous request to read/write/initialize/reset, but useful for any expression.

Parameters

- **expr** – Truth expression upon which to poll, e.g.: XYZREG & XYZREG_EN
- **timeout** – Timeout to wait for in microseconds, e.g.: 1000 (1ms)
- **delay_stmt** – Delay statement to perform each poll iteration e.g.: NULL, *k_yield()*, k_msleep(1) or k_busy_wait(1)

Return values

expr – As a boolean return, if false then it has timed out.

BIT(n)

Unsigned integer with bit position n set (signed in assembly language).

BIT64(_n)

64-bit unsigned integer with bit position _n set.

WRITE_BIT(var, bit, set)

Set or clear a bit depending on a boolean value.

The argument var is a variable whose value is written to as a side effect.

Parameters

- **var** – Variable to be altered
- **bit** – Bit number
- **set** – if 0, clears bit in var; any other value sets bit

BIT_MASK(n)

Bit mask with bits 0 through n-1 (inclusive) set, or 0 if n is 0.

BIT64_MASK(n)

64-bit bit mask with bits 0 through n-1 (inclusive) set, or 0 if n is 0.

IS_POWER_OF_TWO(x)

Check if a x is a power of two.

IS_SHIFTED_BIT_MASK(m, s)

Check if bits are set continuously from the specified bit.

The macro is not dependent on the bit-width.

Parameters

- **m** – Check whether the bits are set continuously or not.
- **s** – Specify the lowest bit for that is continuously set bits.

IS_BIT_MASK(m)

Check if bits are set continuously from the LSB.

Parameters

- **m** – Check whether the bits are set continuously from LSB.

IS_ENABLED(config_macro)

Check for macro definition in compiler-visible expressions.

This trick was pioneered in Linux as the config_enabled() macro. It has the effect of taking a macro value that may be defined to "1" or may not be defined at all and turning it into a literal expression that can be handled by the C compiler instead of just the preprocessor. It is often used with a CONFIG_FOO macro which may be defined to 1 via Kconfig, or left undefined.

That is, it works similarly to `#if defined(CONFIG_FOO)` except that its expansion is a C expression. Thus, much `#ifdef` usage can be replaced with equivalents like:

```
if (IS_ENABLED(CONFIG_FOO)) {
        do_something_with_foo
}
```

This is cleaner since the compiler can generate errors and warnings for `do_something_with_foo` even when CONFIG_FOO is undefined.

Note: Use of IS_ENABLED in a `#if` statement is discouraged as it doesn't provide any benefit vs plain `#if defined()`

Parameters

- **config_macro** – Macro to check

Returns

1 if `config_macro` is defined to 1, 0 otherwise (including if `config_macro` is not defined)

COND_CODE_1(_flag, _if_1_code, _else_code)

Insert code depending on whether `_flag` expands to 1 or not.

This relies on similar tricks as *IS_ENABLED()*, but as the result of `_flag` expansion, results in either `_if_1_code` or `_else_code` is expanded.

To prevent the preprocessor from treating commas as argument separators, the `_if_1_code` and `_else_code` expressions must be inside brackets/parentheses: `()`. These are stripped away during macro expansion.

Example:

```
COND_CODE_1(CONFIG_FLAG, (uint32_t x;), (there_is_no_flag();))
```

If CONFIG_FLAG is defined to 1, this expands to:

```
uint32_t x;
```

It expands to `there_is_no_flag();` otherwise.

This could be used as an alternative to:

```
#if defined(CONFIG_FLAG) && (CONFIG_FLAG == 1)
#define MAYBE_DECLARE(x) uint32_t x
#else
#define MAYBE_DECLARE(x) there_is_no_flag()
#endif

MAYBE_DECLARE(x);
```

However, the advantage of *COND_CODE_1()* is that code is resolved in place where it is used, while the `#if` method defines MAYBE_DECLARE on two lines and requires it to be invoked again on a separate line. This makes *COND_CODE_1()* more concise and also sometimes more useful when used within another macro's expansion.

Note

_flag can be the result of preprocessor expansion, e.g. an expression involving *NUM_VA_ARGS_LESS_1(...)*. However, _if_1_code is only expanded if _flag expands to the integer literal 1. Integer expressions that evaluate to 1, e.g. after doing some arithmetic, will not work.

Parameters

- **_flag** – evaluated flag
- **_if_1_code** – result if _flag expands to 1; must be in parentheses
- **_else_code** – result otherwise; must be in parentheses

COND_CODE_0(_flag, _if_0_code, _else_code)

Like *COND_CODE_1()* except tests if _flag is 0.

This is like *COND_CODE_1()*, except that it tests whether _flag expands to the integer literal 0. It expands to _if_0_code if so, and _else_code otherwise; both of these must be enclosed in parentheses.

See also

COND_CODE_1()

Parameters

- **_flag** – evaluated flag
- **_if_0_code** – result if _flag expands to 0; must be in parentheses
- **_else_code** – result otherwise; must be in parentheses

IF_ENABLED(_flag, _code)

Insert code if _flag is defined and equals 1.

Like *COND_CODE_1()*, this expands to _code if _flag is defined to 1; it expands to nothing otherwise.

Example:

```
IF_ENABLED(CONFIG_FLAG, (uint32_t foo;))
```

If CONFIG_FLAG is defined to 1, this expands to:

```
uint32_t foo;
```

and to nothing otherwise.

It can be considered as a more compact alternative to:

```
#if defined(CONFIG_FLAG) && (CONFIG_FLAG == 1)
uint32_t foo;
#endif
```

Parameters

- **_flag** – evaluated flag
- **_code** – result if _flag expands to 1; must be in parentheses

IF_DISABLED(_flag, _code)

Insert code if _flag is not defined as 1.

This expands to nothing if _flag is defined and equal to 1; it expands to _code otherwise.

Example:

```
IF_DISABLED(CONFIG_FLAG, (uint32_t foo;))
```

If CONFIG_FLAG isn't defined or different than 1, this expands to:

```
uint32_t foo;
```

and to nothing otherwise.

IF_DISABLED does the opposite of IF_ENABLED.

Parameters

- **_flag** – evaluated flag
- **_code** – result if _flag does not expand to 1; must be in parentheses

IS_EMPTY(...)

Check if a macro has a replacement expression.

If a is a macro defined to a nonempty value, this will return true, otherwise it will return false. It only works with defined macros, so an additional #ifdef test may be needed in some cases.

This macro may be used with *COND_CODE_1()* and *COND_CODE_0()* while processing __VA_ARGS__ to avoid processing empty arguments.

Example:

```
#define EMPTY
#define NON_EMPTY  1
#undef  UNDEFINED
IS_EMPTY(EMPTY)
IS_EMPTY(NON_EMPTY)
IS_EMPTY(UNDEFINED)
#if defined(EMPTY) && IS_EMPTY(EMPTY) == true
some_conditional_code
#endif
```

In above examples, the invocations of *IS_EMPTY(...)* return true, false, and true; some_conditional_code is included.

Parameters

- ... – macro to check for emptiness (may be __VA_ARGS__)

IS_EQ(a, b)

Like a == b, but does evaluation and short-circuiting at C preprocessor time.

This however only works for integer literal from 0 to 4095.

LIST_DROP_EMPTY(...)

Remove empty arguments from list.

During macro expansion, __VA_ARGS__ and other preprocessor generated lists may contain empty elements, e.g.:

```
#define LIST ,a,b,,d,
```

Using EMPTY to show each empty element, LIST contains:

```
EMPTY, a, b, EMPTY, d
```

When processing such lists, e.g. using *FOR_EACH()*, all empty elements will be processed, and may require filtering out. To make that process easier, it is enough to invoke LIST_DROP_EMPTY which will remove all empty elements.

Example:

```
LIST_DROP_EMPTY(LIST)
```

expands to:

```
a, b, d
```

Parameters

- ... – list to be processed

EMPTY

Macro with an empty expansion.

This trivial definition is provided for readability when a macro should expand to an empty result, which e.g. is sometimes needed to silence checkpatch.

Example:

```
#define LIST_ITEM(n) , item##n
```

The above would cause checkpatch to complain, but:

```
#define LIST_ITEM(n) EMPTY, item##n
```

would not.

IDENTITY(V)

Macro that expands to its argument.

This is useful in macros like *FOR_EACH()* when there is no transformation required on the list elements.

Parameters

- V – any value

GET_ARG_N(N, ...)

Get nth argument from argument list.

Parameters

- N – Argument index to fetch. Counter from 1.
- ... – Variable list of arguments from which one argument is returned.

Returns

Nth argument.

GET_ARGS_LESS_N(N, ...)

Strips n first arguments from the argument list.

Parameters

- N – Number of arguments to discard.
- ... – Variable list of arguments.

Returns
argument list without N first arguments.

`UTIL_OR`(a, b)

Like `a || b`, but does evaluation and short-circuiting at C preprocessor time.

This is not the same as the binary `||` operator; in particular, `a` should expand to an integer literal 0 or 1. However, `b` can be any value.

This can be useful when `b` is an expression that would cause a build error when `a` is 1.

`UTIL_AND`(a, b)

Like `a && b`, but does evaluation and short-circuiting at C preprocessor time.

This is not the same as the binary `&&`, however; in particular, `a` should expand to an integer literal 0 or 1. However, `b` can be any value.

This can be useful when `b` is an expression that would cause a build error when `a` is 0.

`UTIL_INC`(x)

UTIL_INC(x) for an integer literal x from 0 to 4095 expands to an integer literal whose value is x+1.

See also

UTIL_DEC(x)

`UTIL_DEC`(x)

UTIL_DEC(x) for an integer literal x from 0 to 4095 expands to an integer literal whose value is x-1.

See also

UTIL_INC(x)

`UTIL_X2`(y)

UTIL_X2(y) for an integer literal y from 0 to 4095 expands to an integer literal whose value is 2y.

`LISTIFY`(LEN, F, sep, ...)

Generates a sequence of code with configurable separator.

Example:

```
#define FOO(i, _) MY_PWM ## i
{ LISTIFY(PWM_COUNT, FOO, (,)) }
```

The above two lines expand to:

{ MY_PWM0 , MY_PWM1 }

Note

Calling LISTIFY with undefined arguments has undefined behavior.

Parameters

- **LEN** – The length of the sequence. Must be an integer literal less than 4095.
- **F** – A macro function that accepts at least two arguments: `F(i, ...)`. `F` is called repeatedly in the expansion. Its first argument `i` is the index in the sequence, and the variable list of arguments passed to LISTIFY are passed through to `F`.
- **sep** – Separator (e.g. comma or semicolon). Must be in parentheses; this is required to enable providing a comma as separator.

FOR_EACH(F, sep, ...)

Call a macro `F` on each provided argument with a given separator between each call.

Example:

```
#define F(x) int a##x
FOR_EACH(F, (;), 4, 5, 6);
```

This expands to:

```
int a4;
int a5;
int a6;
```

Parameters

- **F** – Macro to invoke
- **sep** – Separator (e.g. comma or semicolon). Must be in parentheses; this is required to enable providing a comma as separator.
- `...` – Variable argument list. The macro `F` is invoked as `F(element)` for each element in the list.

FOR_EACH_NONEMPTY_TERM(F, term, ...)

Like *FOR_EACH()*, but with a terminator instead of a separator, and drops empty elements from the argument list.

The `sep` argument to *FOR_EACH(F, (sep), a, b)* is a separator which is placed between calls to `F`, like this:

```
FOR_EACH(F, (sep), a, b) // F(a) sep F(b)
                         //              ^^^ no sep here!
```

By contrast, the `term` argument to *FOR_EACH_NONEMPTY_TERM(F, (term),a, b)* is added after each time `F` appears in the expansion:

```
FOR_EACH_NONEMPTY_TERM(F, (term), a, b) // F(a) term F(b) term
                                        //               ^^^^
```

Further, any empty elements are dropped:

```
FOR_EACH_NONEMPTY_TERM(F, (term), a, EMPTY, b) // F(a) term F(b) term
```

This is more convenient in some cases, because *FOR_EACH_NONEMPTY_TERM()* expands to nothing when given an empty argument list, and it's often cumbersome to write a macro `F` that does the right thing even when given an empty argument.

One example is when `__VA_ARGS__` may or may not be empty, and the results are embedded in a larger initializer:

```
#define SQUARE(x) ((x)*(x))

int my_array[] = {
        FOR_EACH_NONEMPTY_TERM(SQUARE, (,), FOO(...))
        FOR_EACH_NONEMPTY_TERM(SQUARE, (,), BAR(...))
        FOR_EACH_NONEMPTY_TERM(SQUARE, (,), BAZ(...))
};
```

This is more convenient than:

a. figuring out whether the FOO, BAR, and BAZ expansions are empty and adding a comma manually (or not) between *FOR_EACH()* calls

b. rewriting SQUARE so it reacts appropriately when "x" is empty (which would be necessary if e.g. FOO expands to nothing)

Parameters

- **F** – Macro to invoke on each nonempty element of the variable arguments
- **term** – Terminator (e.g. comma or semicolon) placed after each invocation of F. Must be in parentheses; this is required to enable providing a comma as separator.
- `...` – Variable argument list. The macro F is invoked as `F(element)` for each nonempty element in the list.

`FOR_EACH_IDX`(F, sep, ...)

Call macro F on each provided argument, with the argument's index as an additional parameter.

This is like *FOR_EACH()*, except F should be a macro which takes two arguments: `F(index, variable_arg)`.

Example:

```
#define F(idx, x) int a##idx = x
FOR_EACH_IDX(F, (;), 4, 5, 6);
```

This expands to:

```
int a0 = 4;
int a1 = 5;
int a2 = 6;
```

Parameters

- **F** – Macro to invoke
- **sep** – Separator (e.g. comma or semicolon). Must be in parentheses; this is required to enable providing a comma as separator.
- `...` – Variable argument list. The macro F is invoked as `F(index, element)` for each element in the list.

`FOR_EACH_FIXED_ARG`(F, sep, fixed_arg, ...)

Call macro F on each provided argument, with an additional fixed argument as a parameter.

This is like *FOR_EACH()*, except F should be a macro which takes two arguments: `F(variable_arg, fixed_arg)`.

Example:

```
static void func(int val, void *dev);
FOR_EACH_FIXED_ARG(func, (;), dev, 4, 5, 6);
```

This expands to:

```
func(4, dev);
func(5, dev);
func(6, dev);
```

Parameters

- **F** – Macro to invoke
- **sep** – Separator (e.g. comma or semicolon). Must be in parentheses; this is required to enable providing a comma as separator.
- **fixed_arg** – Fixed argument passed to `F` as the second macro parameter.
- `...` – Variable argument list. The macro `F` is invoked as `F(element, fixed_arg)` for each element in the list.

`FOR_EACH_IDX_FIXED_ARG`(F, sep, fixed_arg, ...)

Calls macro `F` for each variable argument with an index and fixed argument.

This is like the combination of *FOR_EACH_IDX()* with *FOR_EACH_FIXED_ARG()*.

Example:

```
#define F(idx, x, fixed_arg) int fixed_arg##idx = x
FOR_EACH_IDX_FIXED_ARG(F, (;), a, 4, 5, 6);
```

This expands to:

```
int a0 = 4;
int a1 = 5;
int a2 = 6;
```

Parameters

- **F** – Macro to invoke
- **sep** – Separator (e.g. comma or semicolon). Must be in parentheses; This is required to enable providing a comma as separator.
- **fixed_arg** – Fixed argument passed to `F` as the third macro parameter.
- `...` – Variable list of arguments. The macro `F` is invoked as `F(index, element, fixed_arg)` for each element in the list.

`REVERSE_ARGS`(...)

Reverse arguments order.

Parameters

- `...` – Variable argument list.

`NUM_VA_ARGS_LESS_1`(...)

Number of arguments in the variable arguments list minus one.

Note

Supports up to 64 arguments.

Parameters

- ... – List of arguments

Returns
Number of variadic arguments in the argument list, minus one

NUM_VA_ARGS(...)

Number of arguments in the variable arguments list.

> **Note**
>
> Supports up to 63 arguments.

Parameters

- ... – List of arguments

Returns
Number of variadic arguments in the argument list

MACRO_MAP_CAT(...)

Mapping macro that pastes results together.

This is similar to *FOR_EACH()* in that it invokes a macro repeatedly on each element of __VA_ARGS__. However, unlike *FOR_EACH()*, *MACRO_MAP_CAT()* pastes the results together into a single token.

For example, with this macro FOO:

```
#define FOO(x) item_##x##_
```

MACRO_MAP_CAT(FOO, a, b, c), expands to the token:

```
item_a_item_b_item_c_
```

Parameters

- ... – Macro to expand on each argument, followed by its arguments. (The macro should take exactly one argument.)

Returns
The results of expanding the macro on each argument, all pasted together

MACRO_MAP_CAT_N(N, ...)

Mapping macro that pastes a fixed number of results together.

Similar to *MACRO_MAP_CAT()*, but expects a fixed number of arguments. If more arguments are given than are expected, the rest are ignored.

Parameters

- N – Number of arguments to map
- ... – Macro to expand on each argument, followed by its arguments. (The macro should take exactly one argument.)

Returns
The results of expanding the macro on each argument, all pasted together

Functions

static inline bool **is_power_of_two**(unsigned int x)

Is x a power of two?

Parameters

- **x** – value to check

Returns
true if x is a power of two, false otherwise

ALWAYS_INLINE static bool **is_null_no_warn**(void *p)

Is p equal to NULL?

Some macros may need to check their arguments against NULL to support multiple use-cases, but NULL checks can generate warnings if such a macro is used in contexts where that particular argument can never be NULL.

The warnings can be triggered if: a) all macros are expanded (e.g. when using CONFIG_COMPILER_SAVE_TEMPS=y) or b) tracking of macro expansions are turned off (-ftrack-macro-expansion=0)

The warnings can be circumvented by using this inline function for doing the NULL check within the macro. The compiler is still able to optimize the NULL check out at a later stage.

Parameters

- **p** – Pointer to check

Returns
true if p is equal to NULL, false otherwise

static inline int64_t **arithmetic_shift_right**(int64_t value, uint8_t shift)

Arithmetic shift right.

Parameters

- **value** – value to shift
- **shift** – number of bits to shift

Returns
value shifted right by shift; opened bit positions are filled with the sign bit

static inline void **bytecpy**(void *dst, const void *src, size_t size)

byte by byte memcpy.

Copy size bytes of src into dest. This is guaranteed to be done byte by byte.

Parameters

- **dst** – Pointer to the destination memory.
- **src** – Pointer to the source of the data.
- **size** – The number of bytes to copy.

static inline void **byteswp**(void *a, void *b, size_t size)

byte by byte swap.

Swap *size* bytes between memory regions *a* and *b*. This is guaranteed to be done byte by byte.

Parameters

- **a** – Pointer to the first memory region.

- **b** – Pointer to the second memory region.
- **size** – The number of bytes to swap.

int **char2hex**(char c, uint8_t *x)

Convert a single character into a hexadecimal nibble.

Parameters

- **c** – The character to convert
- **x** – The address of storage for the converted number.

Returns

Zero on success or (negative) error code otherwise.

int **hex2char**(uint8_t x, char *c)

Convert a single hexadecimal nibble into a character.

Parameters

- **c** – The number to convert
- **x** – The address of storage for the converted character.

Returns

Zero on success or (negative) error code otherwise.

size_t **bin2hex**(const uint8_t *buf, size_t buflen, char *hex, size_t hexlen)

Convert a binary array into string representation.

Parameters

- **buf** – The binary array to convert
- **buflen** – The length of the binary array to convert
- **hex** – Address of where to store the string representation.
- **hexlen** – Size of the storage area for string representation.

Returns

The length of the converted string, or 0 if an error occurred.

size_t **hex2bin**(const char *hex, size_t hexlen, uint8_t *buf, size_t buflen)

Convert a hexadecimal string into a binary array.

Parameters

- **hex** – The hexadecimal string to convert
- **hexlen** – The length of the hexadecimal string to convert.
- **buf** – Address of where to store the binary data
- **buflen** – Size of the storage area for binary data

Returns

The length of the binary array, or 0 if an error occurred.

static inline uint8_t **bcd2bin**(uint8_t bcd)

Convert a binary coded decimal (BCD 8421) value to binary.

Parameters

- **bcd** – BCD 8421 value to convert.

Returns

Binary representation of input value.

static inline uint8_t **bin2bcd**(uint8_t bin)

Convert a binary value to binary coded decimal (BCD 8421).

Parameters

- **bin** – Binary value to convert.

Returns
BCD 8421 representation of input value.

uint8_t **u8_to_dec**(char *buf, uint8_t buflen, uint8_t value)

Convert a uint8_t into a decimal string representation.

Convert a uint8_t value into its ASCII decimal string representation. The string is terminated if there is enough space in buf.

Parameters

- **buf** – Address of where to store the string representation.
- **buflen** – Size of the storage area for string representation.
- **value** – The value to convert to decimal string

Returns
The length of the converted string (excluding terminator if any), or 0 if an error occurred.

static inline int32_t **sign_extend**(uint32_t value, uint8_t index)

Sign extend an 8, 16 or 32 bit value using the index bit as sign bit.

Parameters

- **value** – The value to sign expand.
- **index** – 0 based bit index to sign bit (0 to 31)

static inline int64_t **sign_extend_64**(uint64_t value, uint8_t index)

Sign extend a 64 bit value using the index bit as sign bit.

Parameters

- **value** – The value to sign expand.
- **index** – 0 based bit index to sign bit (0 to 63)

char ***utf8_trunc**(char *utf8_str)

Properly truncate a NULL-terminated UTF-8 string.

Take a NULL-terminated UTF-8 string and ensure that if the string has been truncated (by setting the NULL terminator) earlier by other means, that the string ends with a properly formatted UTF-8 character (1-4 bytes).

Parameters

- **utf8_str** – NULL-terminated string

Returns
Pointer to the `utf8_str`

char ***utf8_lcpy**(char *dst, const char *src, size_t n)

Copies a UTF-8 encoded string from `src` to `dst`.

The resulting `dst` will always be NULL terminated if `n` is larger than 0, and the `dst` string will always be properly UTF-8 truncated.

Parameters

- **dst** – The destination of the UTF-8 string.
- **src** – The source string

- **n** – The size of the dst buffer. Maximum number of characters copied is n - 1. If 0 nothing will be done, and the dst will not be NULL terminated.

Returns
Pointer to the dst

static inline void **mem_xor_n**(uint8_t *dst, const uint8_t *src1, const uint8_t *src2, size_t len)

XOR n bytes.

Parameters

- **dst** – Destination of where to store result. Shall be len bytes.
- **src1** – First source. Shall be len bytes.
- **src2** – Second source. Shall be len bytes.
- **len** – Number of bytes to XOR.

static inline void **mem_xor_32**(uint8_t dst[4], const uint8_t src1[4], const uint8_t src2[4])

XOR 32 bits.

Parameters

- **dst** – Destination of where to store result. Shall be 32 bits.
- **src1** – First source. Shall be 32 bits.
- **src2** – Second source. Shall be 32 bits.

static inline void **mem_xor_128**(uint8_t dst[16], const uint8_t src1[16], const uint8_t src2[16])

XOR 128 bits.

Parameters

- **dst** – Destination of where to store result. Shall be 128 bits.
- **src1** – First source. Shall be 128 bits.
- **src2** – Second source. Shall be 128 bits.

3.10 Iterable Sections

This page contains the reference documentation for the iterable sections APIs, which can be used for defining iterable areas of equally-sized data structures, that can be iterated on using *STRUCT_SECTION_FOREACH*.

3.10.1 Usage

Iterable section elements are typically used by defining the data structure and associated initializer in a common header file, so that they can be instantiated anywhere in the code base.

```
struct my_data {
        int a, b;
};

#define DEFINE_DATA(name, _a, _b) \
        STRUCT_SECTION_ITERABLE(my_data, name) = { \
                .a = _a, \
                .b = _b, \
```

(continues on next page)

(continued from previous page)

```
        }

...

DEFINE_DATA(d1, 1, 2);
DEFINE_DATA(d2, 3, 4);
DEFINE_DATA(d3, 5, 6);
```

Then the linker has to be setup to place the place the structure in a contiguous segment using one of the linker macros such as *ITERABLE_SECTION_RAM* or *ITERABLE_SECTION_ROM*. Custom linker snippets are normally declared using one of the `zephyr_linker_sources()` CMake functions, using the appropriate section identifier, `DATA_SECTIONS` for RAM structures and `SECTIONS` for ROM ones.

```
# CMakeLists.txt
zephyr_linker_sources(DATA_SECTIONS iterables.ld)
```

```
# iterables.ld
ITERABLE_SECTION_RAM(my_data, 4)
```

The data can then be accessed using *STRUCT_SECTION_FOREACH*.

```
STRUCT_SECTION_FOREACH(my_data, data) {
        printk("%p: a: %d, b: %d\n", data, data->a, data->b);
}
```

Note

The linker is going to place the entries sorted by name, so the example above would visit `d1`, `d2` and `d3` in that order, regardless of how they were defined in the code.

3.10.2 API Reference

group **iterable_section_apis**

Iterable Sections APIs.

Defines

ITERABLE_SECTION_ROM(struct_type, subalign)

Define a read-only iterable section output.

Define an output section which will set up an iterable area of equally-sized data structures. For use with *STRUCT_SECTION_ITERABLE()*. Input sections will be sorted by name, per ld's SORT_BY_NAME.

This macro should be used for read-only data.

Note that this keeps the symbols in the image even though they are not being directly referenced. Use this when symbols are indirectly referenced by iterating through the section.

ITERABLE_SECTION_ROM_NUMERIC(struct_type, subalign)

Define a read-only iterable section output, sorted numerically.

This version of *ITERABLE_SECTION_ROM()* sorts the entries numerically, that is, SECNAME_10 will come after SECNAME_2. _ separator is required, and up to 2 numeric digits are handled (0-99).

See also

ITERABLE_SECTION_ROM()

ITERABLE_SECTION_ROM_GC_ALLOWED(struct_type, subalign)

Define a garbage collectable read-only iterable section output.

Define an output section which will set up an iterable area of equally-sized data structures. For use with *STRUCT_SECTION_ITERABLE()*. Input sections will be sorted by name, per ld's SORT_BY_NAME.

This macro should be used for read-only data.

Note that the symbols within the section can be garbage collected.

ITERABLE_SECTION_RAM(struct_type, subalign)

Define a read-write iterable section output.

Define an output section which will set up an iterable area of equally-sized data structures. For use with *STRUCT_SECTION_ITERABLE()*. Input sections will be sorted by name, per ld's SORT_BY_NAME.

This macro should be used for read-write data that is modified at runtime.

Note that this keeps the symbols in the image even though they are not being directly referenced. Use this when symbols are indirectly referenced by iterating through the section.

ITERABLE_SECTION_RAM_NUMERIC(struct_type, subalign)

Define a read-write iterable section output, sorted numerically.

This version of *ITERABLE_SECTION_RAM()* sorts the entries numerically, that is, SECNAME10 will come after SECNAME2. Up to 2 numeric digits are handled (0-99).

See also

ITERABLE_SECTION_RAM()

ITERABLE_SECTION_RAM_GC_ALLOWED(struct_type, subalign)

Define a garbage collectable read-write iterable section output.

Define an output section which will set up an iterable area of equally-sized data structures. For use with *STRUCT_SECTION_ITERABLE()*. Input sections will be sorted by name, per ld's SORT_BY_NAME.

This macro should be used for read-write data that is modified at runtime.

Note that the symbols within the section can be garbage collected.

TYPE_SECTION_ITERABLE(type, varname, secname, section_postfix)

Defines a new element for an iterable section for a generic type.

Convenience helper combining __in_section() and Z_DECL_ALIGN(). The section name will be '.[SECNAME].static.[SECTION_POSTFIX]'

In the linker script, create output sections for these using *ITERABLE_SECTION_ROM()* or *ITERABLE_SECTION_RAM()*.

> **Note**
>
> In order to store the element in ROM, a const specifier has to be added to the declaration: const *TYPE_SECTION_ITERABLE(...)*;

Parameters

- `type` – **[in]** data type of variable
- `varname` – **[in]** name of variable to place in section
- `secname` – **[in]** type name of iterable section.
- `section_postfix` – **[in]** postfix to use in section name

`TYPE_SECTION_START`(secname)

iterable section start symbol for a generic type

will return '_[OUT_TYPE]_list_start'.

Parameters

- `secname` – **[in]** type name of iterable section. For 'struct foobar' this would be *TYPE_SECTION_START(foobar)*

`TYPE_SECTION_END`(secname)

iterable section end symbol for a generic type

will return '_<SECNAME>_list_end'.

Parameters

- `secname` – **[in]** type name of iterable section. For 'struct foobar' this would be *TYPE_SECTION_START(foobar)*

`TYPE_SECTION_START_EXTERN`(type, secname)

iterable section extern for start symbol for a generic type

Helper macro to give extern for start of iterable section. The macro typically will be called *TYPE_SECTION_START_EXTERN(struct foobar, foobar)*. This allows the macro to hand different types as well as cases where the type and section name may differ.

Parameters

- `type` – **[in]** data type of section
- `secname` – **[in]** name of output section

`TYPE_SECTION_END_EXTERN`(type, secname)

iterable section extern for end symbol for a generic type

Helper macro to give extern for end of iterable section. The macro typically will be called *TYPE_SECTION_END_EXTERN(struct foobar, foobar)*. This allows the macro to hand different types as well as cases where the type and section name may differ.

Parameters

- `type` – **[in]** data type of section
- `secname` – **[in]** name of output section

TYPE_SECTION_FOREACH(type, secname, iterator)

Iterate over a specified iterable section for a generic type.

Iterator for structure instances gathered by *TYPE_SECTION_ITERABLE()*. The linker must provide a _<SECNAME>_list_start symbol and a _<SECNAME>_list_end symbol to mark the start and the end of the list of struct objects to iterate over. This is normally done using *ITERABLE_SECTION_ROM()* or *ITERABLE_SECTION_RAM()* in the linker script.

TYPE_SECTION_GET(type, secname, i, dst)

Get element from section for a generic type.

Note

There is no protection against reading beyond the section.

Parameters

- `type` – **[in]** type of element
- `secname` – **[in]** name of output section
- `i` – **[in]** Index.
- `dst` – **[out]** Pointer to location where pointer to element is written.

TYPE_SECTION_COUNT(type, secname, dst)

Count elements in a section for a generic type.

Parameters

- `type` – **[in]** type of element
- `secname` – **[in]** name of output section
- `dst` – **[out]** Pointer to location where result is written.

STRUCT_SECTION_START(struct_type)

iterable section start symbol for a struct type

Parameters

- `struct_type` – **[in]** data type of section

STRUCT_SECTION_START_EXTERN(struct_type)

iterable section extern for start symbol for a struct

Helper macro to give extern for start of iterable section.

Parameters

- `struct_type` – **[in]** data type of section

STRUCT_SECTION_END(struct_type)

iterable section end symbol for a struct type

Parameters

- `struct_type` – **[in]** data type of section

STRUCT_SECTION_END_EXTERN(struct_type)

iterable section extern for end symbol for a struct

Helper macro to give extern for end of iterable section.

Parameters

- `struct_type` – **[in]** data type of section

`STRUCT_SECTION_ITERABLE_ALTERNATE`(secname, struct_type, varname)

Defines a new element of alternate data type for an iterable section.

Special variant of *STRUCT_SECTION_ITERABLE()*, for placing alternate data types within the iterable section of a specific data type. The data type sizes and semantics must be equivalent!

`STRUCT_SECTION_ITERABLE_ARRAY_ALTERNATE`(secname, struct_type, varname, size)

Defines an array of elements of alternate data type for an iterable section.

See also

STRUCT_SECTION_ITERABLE_ALTERNATE

`STRUCT_SECTION_ITERABLE`(struct_type, varname)

Defines a new element for an iterable section.

Convenience helper combining __in_section() and Z_DECL_ALIGN(). The section name is the struct type prepended with an underscore. The subsection is "static" and the subsubsection is the variable name.

In the linker script, create output sections for these using *ITERABLE_SECTION_ROM()* or *ITERABLE_SECTION_RAM()*.

Note

In order to store the element in ROM, a const specifier has to be added to the declaration: const *STRUCT_SECTION_ITERABLE(...)*;

`STRUCT_SECTION_ITERABLE_ARRAY`(struct_type, varname, size)

Defines an array of elements for an iterable section.

See also

STRUCT_SECTION_ITERABLE

`STRUCT_SECTION_ITERABLE_NAMED`(struct_type, name, varname)

Defines a new element for an iterable section with a custom name.

The name can be used to customize how iterable section entries are sorted.

See also

STRUCT_SECTION_ITERABLE()

`STRUCT_SECTION_ITERABLE_NAMED_ALTERNATE`(struct_type, secname, name, varname)

Defines a new element for an iterable section with a custom name, placed in a custom section.

The name can be used to customize how iterable section entries are sorted.

> **See also**
>
> *STRUCT_SECTION_ITERABLE_NAMED()*

`STRUCT_SECTION_FOREACH_ALTERNATE`(secname, struct_type, iterator)

Iterate over a specified iterable section (alternate).

Iterator for structure instances gathered by *STRUCT_SECTION_ITERABLE()*. The linker must provide a _<SECNAME>_list_start symbol and a _<SECNAME>_list_end symbol to mark the start and the end of the list of struct objects to iterate over. This is normally done using *ITERABLE_SECTION_ROM()* or *ITERABLE_SECTION_RAM()* in the linker script.

`STRUCT_SECTION_FOREACH`(struct_type, iterator)

Iterate over a specified iterable section.

Iterator for structure instances gathered by *STRUCT_SECTION_ITERABLE()*. The linker must provide a _<struct_type>_list_start symbol and a _<struct_type>_list_end symbol to mark the start and the end of the list of struct objects to iterate over. This is normally done using *ITERABLE_SECTION_ROM()* or *ITERABLE_SECTION_RAM()* in the linker script.

`STRUCT_SECTION_GET`(struct_type, i, dst)

Get element from section.

> **Note**
>
> There is no protection against reading beyond the section.

Parameters

- `struct_type` – **[in]** Struct type.
- `i` – **[in]** Index.
- `dst` – **[out]** Pointer to location where pointer to element is written.

`STRUCT_SECTION_COUNT`(struct_type, dst)

Count elements in a section.

Parameters

- `struct_type` – **[in]** Struct type
- `dst` – **[out]** Pointer to location where result is written.

3.11 Code And Data Relocation

3.11.1 Overview

This feature will allow relocating .text, .rodata, .data, and .bss sections from required files and place them in the required memory region. The memory region and file are given to the *scripts/build/gen_relocate_app.py* script in the form of a string. This script is always invoked from inside cmake.

This script provides a robust way to re-order the memory contents without actually having to modify the code. In simple terms this script will do the job of `__attribute__((section("name")))` for a bunch of files together.

3.11.2 Details

The memory region and file are given to the *scripts/build/gen_relocate_app.py* script in the form of a string.

An example of such a string is: `SRAM2:/home/xyz/zephyr/samples/hello_world/src/main.c, SRAM1:/home/xyz/zephyr/samples/hello_world/src/main2.c`

This script is invoked with the following parameters: `python3 gen_relocate_app.py -i input_string -o generated_linker -c generated_code`

Kconfig `CONFIG_CODE_DATA_RELOCATION` option, when enabled in `prj.conf`, will invoke the script and do the required relocation.

This script also trigger the generation of `linker_relocate.ld` and `code_relocation.c` files. The `linker_relocate.ld` file creates appropriate sections and links the required functions or variables from all the selected files.

Note

The text section is split into 2 parts in the main linker script. The first section will have some info regarding vector tables and other debug related info. The second section will have the complete text section. This is needed to force the required functions and data variables to the correct locations. This is due to the behavior of the linker. The linker will only link once and hence this text section had to be split to make room for the generated linker script.

The `code_relocation.c` file has code that is needed for initializing data sections, and a copy of the text sections (if XIP). Also this contains code needed for bss zeroing and for data copy operations from ROM to required memory type.

The procedure to invoke this feature is:

- Enable `CONFIG_CODE_DATA_RELOCATION` in the `prj.conf` file
- Inside the `CMakeLists.txt` file in the project, mention all the files that need relocation.

  `zephyr_code_relocate(FILES src/*.c LOCATION SRAM2)`

  Where the first argument is the file/files and the second argument is the memory where it must be placed.

  Note

  function zephyr_code_relocate() can be called as many times as required.

Additional Configurations

This section shows additional configuration options that can be set in `CMakeLists.txt`

- if the memory is SRAM1, SRAM2, CCD, or AON, then place the full object in the sections for example:

```
zephyr_code_relocate(FILES src/file1.c LOCATION SRAM2)
zephyr_code_relocate(FILES src/file2.c LOCATION SRAM)
```

- if the memory type is appended with _DATA, _TEXT, _RODATA or _BSS, only the selected memory is placed in the required memory region. for example:

```
zephyr_code_relocate(FILES src/file1.c LOCATION SRAM2_DATA)
zephyr_code_relocate(FILES src/file2.c LOCATION SRAM2_TEXT)
```

- Multiple regions can also be appended together such as: SRAM2_DATA_BSS. This will place data and bss inside SRAM2.
- Multiple files can be passed to the FILES argument, or CMake generator expressions can be used to relocate a comma-separated list of files

```
file(GLOB sources "file*.c")
zephyr_code_relocate(FILES ${sources} LOCATION SRAM)
zephyr_code_relocate(FILES $<TARGET_PROPERTY:my_tgt,SOURCES> LOCATION SRAM)
```

NOKEEP flag

By default, all relocated functions and variables will be marked with `KEEP()` when generating `linker_relocate.ld`. Therefore, if any input file happens to contain unused symbols, then they will not be discarded by the linker, even when it is invoked with `--gc-sections`. If you'd like to override this behavior, you can pass `NOKEEP` to your `zephyr_code_relocate()` call.

```
zephyr_code_relocate(FILES src/file1.c LOCATION SRAM2_TEXT NOKEEP)
```

The example above will help ensure that any unused code found in the .text sections of `file1.c` will not stick to SRAM2.

NOCOPY flag

When a `NOCOPY` option is passed to the `zephyr_code_relocate()` function, the relocation code is not generated in `code_relocation.c`. This flag can be used when we want to move the content of a specific file (or set of files) to a XIP area.

This example will place the .text section of the `xip_external_flash.c` file to the `EXTFLASH` memory region where it will be executed from (XIP). The .data will be relocated as usual into SRAM.

```
zephyr_code_relocate(FILES src/xip_external_flash.c LOCATION EXTFLASH_TEXT NOCOPY)
zephyr_code_relocate(FILES src/xip_external_flash.c LOCATION SRAM_DATA)
```

Relocating libraries

Libraries can be relocated using the LIBRARY argument to `zephyr_code_relocation()` with the library name. For example, the following snippet will relocate serial drivers to SRAM2:

```
zephyr_code_relocate(LIBRARY drivers__serial LOCATION SRAM2)
```

Tips

Take care if relocating kernel/arch files, some contain early initialization code that executes before code relocation takes place.

Additional MPU/MMU configuration may be required to ensure that the destination memory region is configured to allow code execution.

Samples/ Tests

A test showcasing this feature is provided at `$ZEPHYR_BASE/tests/application_development/code_relocation`

This test shows how the code relocation feature is used.

This test will place .text, .data, .bss from 3 files to various parts in the SRAM using a custom linker file derived from `include/zephyr/arch/arm/cortex_m/scripts/linker.ld`

A sample showcasing the NOCOPY flag is provided at `$ZEPHYR_BASE/samples/application_development/code_relocation_nocopy/`

Chapter 4

OS Services

4.1 Binary Descriptors

Binary Descriptors are constant data objects storing information about the binary executable. Unlike "regular" constants, binary descriptors are linked to a known offset in the binary, making them accessible to other programs, such as a different image running on the same device or a host tool. A few examples of constants that would make useful binary descriptors are: kernel version, app version, build time, compiler version, environment variables, compiling host name, etc.

Binary descriptors are created by using the `DEFINE_BINDESC_*` macros. For example:

```
#include <zephyr/bindesc.h>

BINDESC_STR_DEFINE(my_string, 2, "Hello world!"); // Unique ID is 2
```

`my_string` could then be accessed using:

```
printk("my_string: %s\n", BINDESC_GET_STR(my_string));
```

But it could also be retrieved by `west bindesc`:

```
$ west bindesc custom_search STR 2 build/zephyr/zephyr.bin
"Hello world!"
```

4.1.1 Internals

Binary descriptors are implemented with a TLV (tag, length, value) header linked to a known offset in the binary image. This offset may vary between architectures, but generally the descriptors are linked as close to the beginning of the image as possible. In architectures where the image must begin with a vector table (such as ARM), the descriptors are linked right after the vector table. The reset vector points to the beginning of the text section, which is after the descriptors. In architectures where the image must begin with executable code (e.g. x86), a jump instruction is injected at the beginning of the image, in order to skip over the binary descriptors, which are right after the jump instruction.

Each tag is a 16 bit unsigned integer, where the most significant nibble (4 bits) is the type (currently uint, string or bytes), and the rest is the ID. The ID is globally unique to each descriptor. For example, the ID of the app version string is `0x800`, and a string is denoted by 0x1, making the app version tag `0x1800`. The length is a 16 bit number equal to the length of the data in bytes. The data is the actual descriptor value. All binary descriptor numbers (magic, tags, uints) are laid out in memory in the endianness native to the SoC. `west bindesc` assumes little endian by

default, so if the image belongs to a big endian SoC, the appropriate flag should be given to the tool.

The binary descriptor header starts with the magic number `0xb9863e5a7ea46046`. It's followed by the TLVs, and ends with the `DESCRIPTORS_END` (`0xffff`) tag. The tags are always aligned to 32 bits. If the value of the previous descriptor had a non-aligned length, zero padding will be added to ensure that the current tag is aligned.

Putting it all together, here is what the example above would look like in memory (of a little endian SoC):

```
 46 60 a4 7e 5a 3e 86 b9 02 10  0d 00  48 65 6c 6c 6f 20 77 6f 72 6c 64 21 00 00 00 00 ff ff
|         magic         | tag |length| H  e  l  l  o     w  o  r  l  d  !    |   pad  | end␣
↪|
```

4.1.2 Usage

Binary descriptors are always created by the `BINDESC_*_DEFINE` macros. As shown in the example above, a descriptor can be generated from any string or integer, with any ID. However, it is recommended to comply with the standard tags defined in `include/zephyr/bindesc.h`, as that would have the following benefits:

1. The `west bindesc` tool would be able to recognize what the descriptor means and print a meaningful tag
2. It would enforce consistency between various apps from various sources
3. It allows upstream-ability of descriptor generation (see Standard Descriptors)

To define a descriptor with a standard tag, just use the tags included from `bindesc.h`:

```
#include <zephyr/bindesc.h>

BINDESC_STR_DEFINE(app_version, BINDESC_ID_APP_VERSION_STRING, "1.2.3");
```

Standard Descriptors

Some descriptors might be trivial to implement, and could therefore be implemented in a standard way in upstream Zephyr. These could then be enabled via Kconfig, instead of requiring every user to reimplement them. These include build times, kernel version, and host info. For example, to add the build date and time as a string, the following configs should be enabled:

```
# Enable binary descriptors
CONFIG_BINDESC=y

# Enable definition of binary descriptors
CONFIG_BINDESC_DEFINE=y

# Enable default build time binary descriptors
CONFIG_BINDESC_DEFINE_BUILD_TIME=y
CONFIG_BINDESC_BUILD_DATE_TIME_STRING=y
```

To avoid collisions with user defined descriptors, the standard descriptors were allotted the range between `0x800-0xfff`. This leaves `0x000-0x7ff` to users. For more information read the `help` sections of these Kconfig symbols. By convention, each Kconfig symbol corresponds to a binary descriptor whose name is the Kconfig name (with `CONFIG_BINDESC_` removed) in lower case. For example, `CONFIG_BINDESC_KERNEL_VERSION_STRING` creates a descriptor that can be accessed using `BINDESC_GET_STR(kernel_version_string)`.

west bindesc tool

`west` is able to parse and display binary descriptors from a given executable image.

For more information refer to `west bindesc --help` or the *documentation*.

4.1.3 API Reference

Related code samples

Binary descriptors "Hello World"
Set and access binary descriptors for a basic Zephyr application.

group `bindesc_define`

Binary Descriptor Definition.

Defines

`BINDESC_ID_APP_VERSION_STRING`
The app version string such as "1.2.3".

`BINDESC_ID_APP_VERSION_MAJOR`
The app version major such as 1.

`BINDESC_ID_APP_VERSION_MINOR`
The app version minor such as 2.

`BINDESC_ID_APP_VERSION_PATCHLEVEL`
The app version patchlevel such as 3.

`BINDESC_ID_APP_VERSION_NUMBER`
The app version number such as 0x10203.

`BINDESC_ID_KERNEL_VERSION_STRING`
The kernel version string such as "3.4.0".

`BINDESC_ID_KERNEL_VERSION_MAJOR`
The kernel version major such as 3.

`BINDESC_ID_KERNEL_VERSION_MINOR`
The kernel version minor such as 4.

`BINDESC_ID_KERNEL_VERSION_PATCHLEVEL`
The kernel version patchlevel such as 0.

`BINDESC_ID_KERNEL_VERSION_NUMBER`
The kernel version number such as 0x30400.

BINDESC_ID_BUILD_TIME_YEAR
: The year the image was compiled in.

BINDESC_ID_BUILD_TIME_MONTH
: The month of the year the image was compiled in.

BINDESC_ID_BUILD_TIME_DAY
: The day of the month the image was compiled in.

BINDESC_ID_BUILD_TIME_HOUR
: The hour of the day the image was compiled in.

BINDESC_ID_BUILD_TIME_MINUTE
: The minute the image was compiled in.

BINDESC_ID_BUILD_TIME_SECOND
: The second the image was compiled in.

BINDESC_ID_BUILD_TIME_UNIX
: The UNIX time (seconds since midnight of 1970/01/01) the image was compiled in.

BINDESC_ID_BUILD_DATE_TIME_STRING
: The date and time of compilation such as “2023/02/05 00:07:04”.

BINDESC_ID_BUILD_DATE_STRING
: The date of compilation such as “2023/02/05”.

BINDESC_ID_BUILD_TIME_STRING
: The time of compilation such as “00:07:04”.

BINDESC_ID_HOST_NAME
: The name of the host that compiled the image.

BINDESC_ID_C_COMPILER_NAME
: The C compiler name.

BINDESC_ID_C_COMPILER_VERSION
: The C compiler version.

BINDESC_ID_CXX_COMPILER_NAME
: The C++ compiler name.

BINDESC_ID_CXX_COMPILER_VERSION
: The C++ compiler version.

BINDESC_TAG_DESCRIPTORS_END

4.2 Console

Related code samples

Console echo
Echo an input character back to the output using the Console API.

group `console_api`

Console API.

Functions

int `console_init`(void)

Initialize console device.

This function should be called once to initialize pull-style access to console via *console_getchar()* function and buffered output using *console_putchar()* function. This function supersedes, and incompatible with, callback (push-style) console handling (via console_input_fn callback, etc.).

Returns
0 on success, error code (<0) otherwise

ssize_t `console_read`(void *dummy, void *buf, size_t size)

Read data from console.

Parameters

- `dummy` – ignored, present to follow read() prototype
- `buf` – buffer to read data to
- `size` – maximum number of bytes to read

Returns
>0, number of actually read bytes (can be less than size param) =0, in case of EOF <0, in case of error (e.g. -EAGAIN if timeout expired). errno variable is also set.

ssize_t `console_write`(void *dummy, const void *buf, size_t size)

Write data to console.

Parameters

- `dummy` – ignored, present to follow write() prototype
- `buf` – buffer to write data to
- `size` – maximum number of bytes to write

Returns
=>0, number of actually written bytes (can be less than size param) <0, in case of error (e.g. -EAGAIN if timeout expired). errno variable is also set.

int `console_getchar`(void)

Get next char from console input buffer.

Return next input character from console. If no characters available, this function will block. This function is similar to ANSI C getchar() function and is intended to ease porting of existing software. Before this function can be used, *console_init()* should be

called once. This function is incompatible with native Zephyr callback-based console input processing, shell subsystem, or *console_getline()*.

Returns
0-255: a character read, including control characters. <0: error occurred.

int **console_putchar**(char c)

Output a char to console (buffered).

Puts a character into console output buffer. It will be sent to a console asynchronously, e.g. using an IRQ handler.

Returns
<0 on error, otherwise 0.

void **console_getline_init**(void)

Initialize *console_getline()* call.

This function should be called once to initialize pull-style access to console via *console_getline()* function. This function supersedes, and incompatible with, callback (push-style) console handling (via console_input_fn callback, etc.).

char ***console_getline**(void)

Get next line from console input buffer.

Return next input line from console. Until full line is available, this function will block. This function is similar to ANSI C gets() function (except a line is returned in system-owned buffer, and system takes care of the buffer overflow checks) and is intended to ease porting of existing software. Before this function can be used, *console_getline_init()* should be called once. This function is incompatible with native Zephyr callback-based console input processing, shell subsystem, or *console_getchar()*.

Returns
A pointer to a line read, not including EOL character(s). A line resides in a system-owned buffer, so an application should finish any processing of this line immediately after *console_getline()* call, or the buffer can be reused.

4.3 Cryptography

The crypto section contains information regarding the cryptographic primitives supported by the Zephyr kernel. Use the information to understand the principles behind the operation of the different algorithms and how they were implemented.

The following crypto libraries have been included:

4.3.1 PSA Crypto

Overview

The PSA (Platform Security Architecture) Crypto API offers a portable programming interface for cryptographic operations and key storage across a wide range of hardware. It is designed to be user-friendly while still providing access to the low-level primitives essential for modern cryptography.

It is created and maintained by Arm. Arm developed the PSA as a comprehensive security framework to address the increasing security needs of connected devices.

In Zephyr, the PSA Crypto API is implemented using Mbed TLS, an open-source cryptographic library that provides the underlying cryptographic functions.

Design Goals

The interface is suitable for a vast range of devices: from special-purpose cryptographic processors that process data with a built-in key, to constrained devices running custom application code, such as microcontrollers, and multi-application devices, such as servers. It follows the principle of cryptographic agility.

Algorithm Flexibility
: The PSA Crypto API supports a wide range of cryptographic algorithms, allowing developers to switch between different cryptographic methods as needed. This flexibility is crucial for maintaining security as new algorithms emerge and existing ones become obsolete.

Key Management
: The PSA Crypto API includes robust key management features that support the creation, storage, and use of cryptographic keys in a secure and flexible manner. It uses opaque key identifiers, which allows for easy key replacement and updates without exposing key material.

Implementation Independence
: The PSA Crypto API abstracts the underlying cryptographic library, meaning that the specific implementation can be changed without affecting the application code. This abstraction supports cryptographic agility by enabling the use of different cryptographic libraries or hardware accelerators as needed.

Future-Proofing
: By adhering to cryptographic agility, PSA Crypto ensures that applications can quickly adapt to new cryptographic standards and practices, enhancing long-term security and compliance.

Examples of Applications

Network Security (TLS)
: The API provides all of the cryptographic primitives needed to establish TLS connections.

Secure Storage
: The API provides all primitives related to storage encryption, block or file-based, with master encryption keys stored inside a key store.

Network Credentials
: The API provides network credential management inside a key store, for example, for X.509-based authentication or pre-shared keys on enterprise networks.

Device Pairing
: The API provides support for key agreement protocols that are often used for secure pairing of devices over wireless channels. For example, the pairing of an NFC token or a Bluetooth device might use key agreement protocols upon first use.

Secure Boot
: The API provides primitives for use during firmware integrity and authenticity validation, during a secure or trusted boot process.

Attestation
: The API provides primitives used in attestation activities. Attestation is the ability for a device to sign an array of bytes with a device private key and return the result to the caller. There are several use cases; ranging from attestation of the device state, to the ability to generate a key pair and prove that it has been generated inside a secure key store. The API provides access to the algorithms commonly used for attestation.

Factory Provisioning
: Most IoT devices receive a unique identity during the factory provisioning process, or once they have been deployed to the field. This API provides the APIs necessary for populating a device with keys that represent that identity.

Usage considerations

Always check for errors

Most functions in the PSA Crypto API can return errors. All functions that can fail have the return type `psa_status_t`. A few functions cannot fail, and thus, return void or some other type.

If an error occurs, unless otherwise specified, the content of the output parameters is undefined and must not be used.

Some common causes of errors include:

- In implementations where the keys are stored and processed in a separate environment from the application, all functions that need to access the cryptography processing environment might fail due to an error in the communication between the two environments.
- If an algorithm is implemented with a hardware accelerator, which is logically separate from the application processor, the accelerator might fail, even when the application processor keeps running normally.
- Most functions might fail due to a lack of resources. However, some implementations guarantee that certain functions always have sufficient memory.
- All functions that access persistent keys might fail due to a storage failure.
- All functions that require randomness might fail due to a lack of entropy. Implementations are encouraged to seed the random generator with sufficient entropy during the execution of `psa_crypto_init()`. However, some security standards require periodic reseeding from a hardware random generator, which can fail.

Shared memory and concurrency

Some environments allow applications to be multithreaded, while others do not. In some environments, applications can share memory with a different security context. In environments with multithreaded applications or shared memory, applications must be written carefully to avoid data corruption or leakage. This specification requires the application to obey certain constraints.

In general, the PSA Crypto API allows either one writer or any number of simultaneous readers, on any given object. In other words, if two or more calls access the same object concurrently, then the behavior is only well-defined if all the calls are only reading from the object and do not modify it. Read accesses include reading memory by input parameters and reading keystore content by using a key. For more details, refer to Concurrent calls

If an application shares memory with another security context, it can pass shared memory blocks as input buffers or output buffers, but not as non-buffer parameters. For more details, refer to Stability of parameters.

Cleaning up after use

To minimize impact if the system is compromised, it is recommended that applications wipe all sensitive data from memory when it is no longer used. That way, only data that is currently in use can be leaked, and past data is not compromised.

Wiping sensitive data includes:

- Clearing temporary buffers in the stack or on the heap.
- Aborting operations if they will not be finished.
- Destroying keys that are no longer used.

References

- PSA Crypto

- Mbed TLS

4.3.2 Random Number Generation

The random API subsystem provides random number generation APIs in both cryptographically and non-cryptographically secure instances. Which random API to use is based on the cryptographic requirements of the random number. The non-cryptographic APIs will return random values much faster if non-cryptographic values are needed.

The cryptographically secure random functions shall be compliant to the FIPS 140-2 [**?**] recommended algorithms. Hardware based random-number generators (RNG) can be used on platforms with appropriate hardware support. Platforms without hardware RNG support shall use the CTR-DRBG algorithm. The algorithm can be provided by TinyCrypt or mbedTLS depending on your application performance and resource requirements.

> **Note**
>
> The CTR-DRBG generator needs an entropy source to establish and maintain the cryptographic security of the PRNG.

Kconfig Options

These options can be found in the following path subsys/random/Kconfig.

`CONFIG_TEST_RANDOM_GENERATOR`
: For testing, this option allows a non-random number generator to be used and permits random number APIs to return values that are not truly random.

The random number generator choice group allows selection of the RNG source function for the system via the RNG_GENERATOR_CHOICE choice group. An override of the default value can be specified in the SOC or board .defconfig file by using:

```
choice RNG_GENERATOR_CHOICE
        default XOSHIRO_RANDOM_GENERATOR
endchoice
```

The random number generators available include:

`CONFIG_TIMER_RANDOM_GENERATOR`
: enables number generator based on system timer clock. This number generator is not random and used for testing only.

`CONFIG_ENTROPY_DEVICE_RANDOM_GENERATOR`
: enables a random number generator that uses the enabled hardware entropy gathering driver to generate random numbers.

`CONFIG_XOSHIRO_RANDOM_GENERATOR`
: enables the Xoshiro128++ pseudo-random number generator, that uses the entropy driver as a seed source.

The CSPRNG_GENERATOR_CHOICE choice group provides selection of the cryptographically secure random number generator source function. An override of the default value can be specified in the SOC or board .defconfig file by using:

```
choice CSPRNG_GENERATOR_CHOICE
        default CTR_DRBG_CSPRNG_GENERATOR
endchoice
```

The cryptographically secure random number generators available include:

`CONFIG_HARDWARE_DEVICE_CS_GENERATOR`
: enables a cryptographically secure random number generator using the hardware random generator driver

`CONFIG_CTR_DRBG_CSPRNG_GENERATOR`
: enables the CTR-DRBG pseudo-random number generator. The CTR-DRBG is a FIPS140-2 recommended cryptographically secure random number generator.

Personalization data can be provided in addition to the entropy source to make the initialization of the CTR-DRBG as unique as possible.

`CONFIG_CS_CTR_DRBG_PERSONALIZATION`
: CTR-DRBG Initialization Personalization string

API Reference

Related code samples

AWS IoT Core MQTT
: Connect to AWS IoT Core and publish messages using MQTT.

Microsoft Azure IoT Hub MQTT
: Connect to Azure IoT Hub and publish messages using MQTT.

group `random_api`

Random Function APIs.

Since
: 1.0

Version
: 1.0.0

Functions

void `sys_rand_get`(void *dst, size_t len)

Fill the destination buffer with random data values that should pass general randomness tests.

Note

The random values returned are not considered cryptographically secure random number values.

Parameters

- `dst` – **[out]** destination buffer to fill with random data.
- `len` – size of the destination buffer.

int `sys_csrand_get`(void *dst, size_t len)

Fill the destination buffer with cryptographically secure random data values.

Note

If the random values requested do not need to be cryptographically secure then use *sys_rand_get()* instead.

Parameters

- `dst` – **[out]** destination buffer to fill.
- `len` – size of the destination buffer.

Returns

0 if success, -EIO if entropy reseed error

static inline uint8_t `sys_rand8_get`(void)

Return a 8-bit random value that should pass general randomness tests.

Note

The random value returned is not a cryptographically secure random number value.

Returns

8-bit random value.

static inline uint16_t `sys_rand16_get`(void)

Return a 16-bit random value that should pass general randomness tests.

Note

The random value returned is not a cryptographically secure random number value.

Returns

16-bit random value.

static inline uint32_t `sys_rand32_get`(void)

Return a 32-bit random value that should pass general randomness tests.

Note

The random value returned is not a cryptographically secure random number value.

Returns

32-bit random value.

static inline uint64_t `sys_rand64_get`(void)

Return a 64-bit random value that should pass general randomness tests.

Note

The random value returned is not a cryptographically secure random number value.

Returns
64-bit random value.

4.3.3 Crypto APIs

Overview

API Reference

Related code samples

Crypto
Use the crypto APIs to perform various encryption/decryption operations.

Generic API for crypto drivers

group `crypto`

Crypto APIs.

Since
1.7

Version
1.0.0

Defines

`CAP_OPAQUE_KEY_HNDL`

`CAP_RAW_KEY`

`CAP_KEY_LOADING_API`

`CAP_INPLACE_OPS`
Whether the output is placed in separate buffer or not.

`CAP_SEPARATE_IO_BUFS`

`CAP_SYNC_OPS`
These denotes if the output (completion of a cipher_xxx_op) is conveyed by the op function returning, or it is conveyed by an async notification.

`CAP_ASYNC_OPS`

`CAP_AUTONONCE`
Whether the hardware/driver supports autononce feature.

`CAP_NO_IV_PREFIX`
Don't prefix IV to cipher blocks.

Functions

static inline int **crypto_query_hwcaps**(const struct *device* *dev)
Query the crypto hardware capabilities.

This API is used by the app to query the capabilities supported by the crypto device. Based on this the app can specify a subset of the supported options to be honored for a session during *cipher_begin_session()*.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Returns
bitmask of supported options.

struct **crypto_driver_api**
#include <*crypto.h*> Crypto driver API definition.

Ciphers API

group `crypto_cipher`
Crypto Cipher APIs.

Typedefs

typedef int (***block_op_t**)(struct *cipher_ctx* *ctx, struct *cipher_pkt* *pkt)

typedef int (***cbc_op_t**)(struct *cipher_ctx* *ctx, struct *cipher_pkt* *pkt, uint8_t *iv)

typedef int (***ctr_op_t**)(struct *cipher_ctx* *ctx, struct *cipher_pkt* *pkt, uint8_t *ctr)

typedef int (***ccm_op_t**)(struct *cipher_ctx* *ctx, struct *cipher_aead_pkt* *pkt, uint8_t *nonce)

typedef int (***gcm_op_t**)(struct *cipher_ctx* *ctx, struct *cipher_aead_pkt* *pkt, uint8_t *nonce)

typedef void (***cipher_completion_cb**)(struct *cipher_pkt* *completed, int status)

Enums

enum **cipher_algo**
Cipher Algorithm.

Values:

enumerator `CRYPTO_CIPHER_ALGO_AES` = 1

enum **cipher_op**

Cipher Operation.

Values:

enumerator **CRYPTO_CIPHER_OP_DECRYPT** = 0

enumerator **CRYPTO_CIPHER_OP_ENCRYPT** = 1

enum **cipher_mode**

Possible cipher mode options.

More to be added as required.

Values:

enumerator **CRYPTO_CIPHER_MODE_ECB** = 1

enumerator **CRYPTO_CIPHER_MODE_CBC** = 2

enumerator **CRYPTO_CIPHER_MODE_CTR** = 3

enumerator **CRYPTO_CIPHER_MODE_CCM** = 4

enumerator **CRYPTO_CIPHER_MODE_GCM** = 5

Functions

static inline int **cipher_begin_session**(const struct *device* *dev, struct *cipher_ctx* *ctx, enum *cipher_algo* algo, enum *cipher_mode* mode, enum *cipher_op* optype)

Setup a crypto session.

Initializes one time parameters, like the session key, algorithm and cipher mode which may remain constant for all operations in the session. The state may be cached in hardware and/or driver data state variables.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **ctx** – Pointer to the context structure. Various one time parameters like key, keylength, etc. are supplied via this structure. The structure documentation specifies which fields are to be populated by the app before making this call.
- **algo** – The crypto algorithm to be used in this session. e.g AES
- **mode** – The cipher mode to be used in this session. e.g CBC, CTR
- **optype** – Whether we should encrypt or decrypt in this session

Returns

0 on success, negative errno code on fail.

static inline int **cipher_free_session**(const struct *device* *dev, struct *cipher_ctx* *ctx)

Cleanup a crypto session.

Clears the hardware and/or driver state of a previous session.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **ctx** – Pointer to the crypto context structure of the session to be freed.

Returns

0 on success, negative errno code on fail.

static inline int **cipher_callback_set**(const struct *device* *dev, *cipher_completion_cb* cb)

Registers an async crypto op completion callback with the driver.

The application can register an async crypto op completion callback handler to be invoked by the driver, on completion of a prior request submitted via cipher_do_op(). Based on crypto device hardware semantics, this is likely to be invoked from an ISR context.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **cb** – Pointer to application callback to be called by the driver.

Returns

0 on success, -ENOTSUP if the driver does not support async op, negative errno code on other error.

static inline int **cipher_block_op**(struct *cipher_ctx* *ctx, struct *cipher_pkt* *pkt)

Perform single-block crypto operation (ECB cipher mode).

This should not be overloaded to operate on multiple blocks for security reasons.

Parameters

- **ctx** – Pointer to the crypto context of this op.
- **pkt** – Structure holding the input/output buffer pointers.

Returns

0 on success, negative errno code on fail.

static inline int **cipher_cbc_op**(struct *cipher_ctx* *ctx, struct *cipher_pkt* *pkt, uint8_t *iv)

Perform Cipher Block Chaining (CBC) crypto operation.

Parameters

- **ctx** – Pointer to the crypto context of this op.
- **pkt** – Structure holding the input/output buffer pointers.
- **iv** – Initialization Vector (IV) for the operation. Same IV value should not be reused across multiple operations (within a session context) for security.

Returns

0 on success, negative errno code on fail.

static inline int **cipher_ctr_op**(struct *cipher_ctx* *ctx, struct *cipher_pkt* *pkt, uint8_t *iv)

Perform Counter (CTR) mode crypto operation.

Parameters

- **ctx** – Pointer to the crypto context of this op.
- **pkt** – Structure holding the input/output buffer pointers.

- `iv` – Initialization Vector (IV) for the operation. We use a split counter formed by appending IV and ctr. Consequently ivlen = keylen - ctrlen. 'ctrlen' is specified during session setup through the 'ctx.mode_params.ctr_params.ctr_len' parameter. IV should not be reused across multiple operations (within a session context) for security. The non-IV part of the split counter is transparent to the caller and is fully managed by the crypto provider.

Returns
0 on success, negative errno code on fail.

static inline int **cipher_ccm_op**(struct *cipher_ctx* *ctx, struct *cipher_aead_pkt* *pkt, uint8_t *nonce)

Perform Counter with CBC-MAC (CCM) mode crypto operation.

Parameters

- `ctx` – Pointer to the crypto context of this op.
- `pkt` – Structure holding the input/output, Associated Data (AD) and auth tag buffer pointers.
- `nonce` – Nonce for the operation. Same nonce value should not be reused across multiple operations (within a session context) for security.

Returns
0 on success, negative errno code on fail.

static inline int **cipher_gcm_op**(struct *cipher_ctx* *ctx, struct *cipher_aead_pkt* *pkt, uint8_t *nonce)

Perform Galois/Counter Mode (GCM) crypto operation.

Parameters

- `ctx` – Pointer to the crypto context of this op.
- `pkt` – Structure holding the input/output, Associated Data (AD) and auth tag buffer pointers.
- `nonce` – Nonce for the operation. Same nonce value should not be reused across multiple operations (within a session context) for security.

Returns
0 on success, negative errno code on fail.

struct **cipher_ops**

#include <cipher.h>

struct **ccm_params**

#include <cipher.h>

struct **ctr_params**

#include <cipher.h>

struct **gcm_params**

#include <cipher.h>

struct **cipher_ctx**

#include <cipher.h> Structure encoding session parameters.

Refer to comments for individual fields to know the contract in terms of who fills what and when w.r.t begin_session() call.

Public Members

struct *cipher_ops* **ops**

Place for driver to return function pointers to be invoked per cipher operation.

To be populated by crypto driver on return from begin_session() based on the algo/mode chosen by the app.

union *cipher_ctx* **key**

To be populated by the app before calling begin_session()

const struct *device* ***device**

The device driver instance this crypto context relates to.

Will be populated by the begin_session() API.

void ***drv_sessn_state**

If the driver supports multiple simultaneously crypto sessions, this will identify the specific driver state this crypto session relates to.

Since dynamic memory allocation is not possible, it is suggested that at build time drivers allocate space for the max simultaneous sessions they intend to support. To be populated by the driver on return from begin_session().

void ***app_sessn_state**

Place for the user app to put info relevant stuff for resuming when completion callback happens for async ops.

Totally managed by the app.

union *cipher_ctx* **mode_params**

Cypher mode parameters, which remain constant for all ops in a session.

To be populated by the app before calling begin_session().

uint16_t **keylen**

Cryptographic keylength in bytes.

To be populated by the app before calling begin_session()

uint16_t **flags**

How certain fields are to be interpreted for this session.

(A bitmask of CAP_* below.) To be populated by the app before calling begin_session(). An app can obtain the capability flags supported by a hw/driver by calling *crypto_query_hwcaps()*.

struct **cipher_pkt**

#include <*cipher.h*> Structure encoding IO parameters of one cryptographic operation like encrypt/decrypt.

The fields which has not been explicitly called out has to be filled up by the app before making the cipher_xxx_op() call.

Public Members

uint8_t *in_buf

Start address of input buffer.

int in_len

Bytes to be operated upon.

uint8_t *out_buf

Start of the output buffer, to be allocated by the application.

Can be NULL for in-place ops. To be populated with contents by the driver on return from op / async callback.

int out_buf_max

Size of the out_buf area allocated by the application.

Drivers should not write past the size of output buffer.

int out_len

To be populated by driver on return from cipher_xxx_op() and holds the size of the actual result.

struct *cipher_ctx* *ctx

Context this packet relates to.

This can be useful to get the session details, especially for async ops. Will be populated by the cipher_xxx_op() API based on the ctx parameter.

struct cipher_aead_pkt

#include <*cipher.h*> Structure encoding IO parameters in AEAD (Authenticated Encryption with Associated Data) scenario like in CCM.

App has to furnish valid contents prior to making *cipher_ccm_op()* call.

Public Members

uint8_t *ad

Start address for Associated Data.

This has to be supplied by app.

uint32_t ad_len

Size of Associated Data.

This has to be supplied by the app.

uint8_t *tag

Start address for the auth hash.

For an encryption op this will be populated by the driver when it returns from cipher_ccm_op call. For a decryption op this has to be supplied by the app.

4.4 Debugging

4.4.1 Thread analyzer

The thread analyzer module enables all the Zephyr options required to track the thread information, e.g. thread stack size usage and other runtime thread runtime statistics.

The analysis is performed on demand when the application calls *thread_analyzer_run()* or *thread_analyzer_print()*.

For example, to build the synchronization sample with Thread Analyser enabled, do the following:

```
west build -b qemu_x86 samples/synchronization/ -- -DCONFIG_QEMU_ICOUNT=n -
↪DCONFIG_THREAD_ANALYZER=y \
-DCONFIG_THREAD_ANALYZER_USE_PRINTK=y -DCONFIG_THREAD_ANALYZER_AUTO=y \
-DCONFIG_THREAD_ANALYZER_AUTO_INTERVAL=5
```

When you run the generated application in Qemu, you will get the additional information from Thread Analyzer:

```
thread_a: Hello World from cpu 0 on qemu_x86!
Thread analyze:
 thread_b            : STACK: unused 740 usage 284 / 1024 (27 %); CPU: 0 %
 thread_analyzer     : STACK: unused 8 usage 504 / 512 (98 %); CPU: 0 %
 thread_a            : STACK: unused 648 usage 376 / 1024 (36 %); CPU: 98 %
 idle                : STACK: unused 204 usage 116 / 320 (36 %); CPU: 0 %
thread_b: Hello World from cpu 0 on qemu_x86!
thread_a: Hello World from cpu 0 on qemu_x86!
thread_b: Hello World from cpu 0 on qemu_x86!
thread_a: Hello World from cpu 0 on qemu_x86!
thread_b: Hello World from cpu 0 on qemu_x86!
thread_a: Hello World from cpu 0 on qemu_x86!
thread_b: Hello World from cpu 0 on qemu_x86!
thread_a: Hello World from cpu 0 on qemu_x86!
Thread analyze:
 thread_b            : STACK: unused 648 usage 376 / 1024 (36 %); CPU: 7 %
 thread_analyzer     : STACK: unused 8 usage 504 / 512 (98 %); CPU: 0 %
 thread_a            : STACK: unused 648 usage 376 / 1024 (36 %); CPU: 9 %
 idle                : STACK: unused 204 usage 116 / 320 (36 %); CPU: 82 %
thread_b: Hello World from cpu 0 on qemu_x86!
thread_a: Hello World from cpu 0 on qemu_x86!
thread_b: Hello World from cpu 0 on qemu_x86!
thread_a: Hello World from cpu 0 on qemu_x86!
thread_b: Hello World from cpu 0 on qemu_x86!
thread_a: Hello World from cpu 0 on qemu_x86!
thread_b: Hello World from cpu 0 on qemu_x86!
thread_a: Hello World from cpu 0 on qemu_x86!
Thread analyze:
 thread_b            : STACK: unused 648 usage 376 / 1024 (36 %); CPU: 7 %
 thread_analyzer     : STACK: unused 8 usage 504 / 512 (98 %); CPU: 0 %
 thread_a            : STACK: unused 648 usage 376 / 1024 (36 %); CPU: 8 %
 idle                : STACK: unused 204 usage 116 / 320 (36 %); CPU: 83 %
thread_b: Hello World from cpu 0 on qemu_x86!
thread_a: Hello World from cpu 0 on qemu_x86!
thread_b: Hello World from cpu 0 on qemu_x86!
```

Configuration

Configure this module using the following options.

- THREAD_ANALYZER: enable the module.
- THREAD_ANALYZER_USE_PRINTK: use printk for thread statistics.
- THREAD_ANALYZER_USE_LOG: use the logger for thread statistics.
- THREAD_ANALYZER_AUTO: run the thread analyzer automatically. You do not need to add any code to the application when using this option.
- THREAD_ANALYZER_AUTO_INTERVAL: the time for which the module sleeps between consecutive printing of thread analysis in automatic mode.
- THREAD_ANALYZER_AUTO_STACK_SIZE: the stack for thread analyzer automatic thread.
- THREAD_NAME: enable this option in the kernel to print the name of the thread instead of its ID.
- THREAD_RUNTIME_STATS: enable this option to print thread runtime data such as utilization (This options is automatically selected by THREAD_ANALYZER).

API documentation

group **thread_analyzer**

Module for analyzing threads.

This module implements functions and the configuration that simplifies thread analysis.

Typedefs

typedef void (***thread_analyzer_cb**)(struct *thread_analyzer_info* *info)

Thread analyzer stack size callback function.

Callback function with thread analysis information.

Param info
Thread analysis information.

Functions

void **thread_analyzer_run**(*thread_analyzer_cb* cb)

Run the thread analyzer and provide information to the callback.

This function analyzes the current state for all threads and calls a given callback on every thread found.

Parameters

- **cb** – The callback function handler

void **thread_analyzer_print**(void)

Run the thread analyzer and print stack size statistics.

This function runs the thread analyzer and prints the output in standard form.

struct **thread_analyzer_info**

#include <thread_analyzer.h>

Public Members

const char *name
: The name of the thread or stringified address of the thread handle if name is not set.

size_t stack_size
: The total size of the stack.

size_t stack_used
: Stack size in used.

4.4.2 Core Dump

The core dump module enables dumping the CPU registers and memory content for offline debugging. This module is called when a fatal error is encountered and prints or stores data according to which backends are enabled.

Configuration

Configure this module using the following options.

- DEBUG_COREDUMP: enable the module.

Here are the options to enable output backends for core dump:

- DEBUG_COREDUMP_BACKEND_LOGGING: use log module for core dump output.
- DEBUG_COREDUMP_BACKEND_FLASH_PARTITION: use flash partition for core dump output.
- DEBUG_COREDUMP_BACKEND_NULL: fallback core dump backend if other backends cannot be enabled. All output is sent to null.

Here are the choices regarding memory dump:

- DEBUG_COREDUMP_MEMORY_DUMP_MIN: only dumps the stack of the exception thread, its thread struct, and some other bare minimal data to support walking the stack in the debugger. Use this only if absolute minimum of data dump is desired.

Additional memory can be included in a dump (even with the "DEBUG_COREDUMP_MEMORY_DUMP_MIN" config selected) through one or more *coredump devices*

Usage

When the core dump module is enabled, during a fatal error, CPU registers and memory content are printed or stored according to which backends are enabled. This core dump data can fed into a custom-made GDB server as a remote target for GDB (and other GDB compatible debuggers). CPU registers, memory content and stack can be examined in the debugger.

This usually involves the following steps:

1. Get the core dump log from the device depending on enabled backends. For example, if the log module backend is used, get the log output from the log module backend.
2. Convert the core dump log into a binary format that can be parsed by the GDB server. For example, scripts/coredump/coredump_serial_log_parser.py can be used to convert the serial console log into a binary file.

3. Start the custom GDB server using the script scripts/coredump/coredump_gdbserver.py with the core dump binary log file, and the Zephyr ELF file as parameters. The GDB server can also be started from within GDB, see below.
4. Start the debugger corresponding to the target architecture.

> **Note**
>
> Developers for Intel ADSP CAVS 15-25 platforms using ZEPHYR_TOOLCHAIN_VARIANT=zephyr should use the debugger in the xtensa-intel_apl_adsp toolchain of the SDK.

5. When DEBUG_COREDUMP_BACKEND_FLASH_PARTITION is enabled the core dump data is stored in the flash partition. The flash partition must be defined in the device tree:

```
&flash0 {
        partitions {
                coredump_partition: partition@255000 {
                        label = "coredump-partition";
                        reg = <0x255000 DT_SIZE_K(4)>;
                };
};
```

Example This example uses the log module backend tied to serial console. This was done on qemu_x86 where a null pointer was dereferenced.

This is the core dump log from the serial console, and is stored in coredump.log:

```
Booting from ROM..*** Booting Zephyr OS build zephyr-v2.3.0-1840-g7bba91944a63  ***
Hello World! qemu_x86
E: Page fault at address 0x0 (error code 0x2)
E: Linear address not present in page tables
E:   PDE: 0x0000000000115827 Writable, User, Execute Enabled
E:   PTE: Non-present
E: EAX: 0x00000000, EBX: 0x00000000, ECX: 0x00119d74, EDX: 0x000003f8
E: ESI: 0x00000000, EDI: 0x00101aa7, EBP: 0x00119d10, ESP: 0x00119d00
E: EFLAGS: 0x00000206 CS: 0x0008 CR3: 0x00119000
E: call trace:
E: EIP: 0x00100459
E:      0x00100477 (0x0)
E:      0x00100492 (0x0)
E:      0x001004c8 (0x0)
E:      0x00105465 (0x105465)
E:      0x00101abe (0x0)
E: >>> ZEPHYR FATAL ERROR 0: CPU exception on CPU 0
E: Current thread: 0x00119080 (unknown)
E: #CD:BEGIN#
E: #CD:5a4501000100050000000000
E: #CD:4101003800
E: #CD:0e0000000200000000000000749d1100f80300000000000009d1100109d1100
E: #CD:00000000a71a10005904100006020000080000000901100
E: #CD:4d010080901100e0901100
E: #CD:0100000000000000000000000018000000000000000000000000000000000000
E: #CD:00000000000000000000000000000000e364100000000000000000004c9c1100
E: #CD:000000000000000000000000b4991100004000000000000fc03000000000000
E: #CD:4d0100b4991100b49d1100
E: #CD:f80300000200000002000000020000f8030000fd03000a02000000dc9e1100
E: #CD:149a1160fd03000002000000dc9e1100249a110087201000049f11000a000000
E: #CD:349a11000a4f1000049f11000a9e1100449a11000a8b1000020000002000000
E: #CD:449a1100388b1000049f11000a000000549a1100ad201000049f11000a000000
E: #CD:749a11000a201000049f11000a000000649a11000a201000049f11000a000000
```

(continues on next page)

(continued from previous page)

```
E: #CD:749a1100e8201000049f11000a000000949a1100890b10000a0000000a000000
E: #CD:a49a1100890b10000a0000000a000000f8030000189b1100020000002000000
E: #CD:f49a1100289b11000a000000189b1100049b11009b0710000a000000289b1100
E: #CD:f49a110087201000049f110045000000f49a1100509011000a00000020901100
E: #CD:f49a110060901100049f1100ffffffff0000000000000000049f1100ffffffff
E: #CD:000000000000000630b1000189b1100349b1100af0b1000630b1000289b1100
E: #CD:55891000789b11000000000020901100549b1100480000004a891000609b1100
E: #CD:649b1100d00b10004a891000709b110000000000609b11000a00000000000000
E: #CD:849b1100709b11004a8910000000000949b1100794a1000000000058901100
E: #CD:20901100c34a10000a0000173402000d00100000000000d49b110038000000
E: #CD:c49b110078481000b4991100000400000000000000000000c9c11000c9c1100
E: #CD:149c110000000000d49b110038000000f49b1100da481000b499110000040000
E: #CD:0e000000200000000000000744d0100b4991100b49d1100009d1100109d1100
E: #CD:149c110099471000b4991100000400000800000000901100ad861000409c1100
E: #CD:349c1100e947100080901100000000000349c1100b64710008086100045000000
E: #CD:849c11002d53100000000000d09c11008090110020861000f5ffffff8c9c1100
E: #CD:00000000000000000000000a71a1000a49c1100020200008090110000000000
E: #CD:a49c11000202000008000000000000000a49c1100193710000000000d09c1100
E: #CD:0c9d0000bc9c0000b49d1100b4991100c49c1100ae37100000000000d09c1100
E: #CD:0800000000000000c8881000000000000109d11005d031000d09c1100009d1100
E: #CD:109d11000000000000000000a71a1000f80300000000000749d110002000000
E: #CD:5904100008000000060200000e000000020200000202000000000002c9d1100
E: #CD:7704100000000000d00b1000c9881000549d110000000000489d110092041000
E: #CD:00000000689d1100549d11000000000000000000689d1100c804100000000000
E: #CD:c0881000000000007c9d110000000000749d11007c9d11006554100065541000
E: #CD:0000000000000009c9d1100be1a10000000000000000000000000038041000
E: #CD:0800000002020000000000000000000f45310000000000000000000000000000
E: #CD:END#
E: Halting system
```

1. Run the core dump serial log converter:

   ```
   ./scripts/coredump/coredump_serial_log_parser.py coredump.log coredump.bin
   ```

2. Start the custom GDB server:

   ```
   ./scripts/coredump/coredump_gdbserver.py build/zephyr/zephyr.elf coredump.bin
   ```

3. Start GDB:

   ```
   <path to SDK>/x86_64-zephyr-elf/bin/x86_64-zephyr-elf-gdb build/zephyr/zephyr.elf
   ```

4. Inside GDB, connect to the GDB server via port 1234:

   ```
   (gdb) target remote localhost:1234
   ```

5. Examine the CPU registers:

   ```
   (gdb) info registers
   ```

   Output from GDB:

   ```
   eax            0x0                 0
   ecx            0x119d74            1154420
   edx            0x3f8               1016
   ebx            0x0                 0
   esp            0x119d00            0x119d00 <z_main_stack+844>
   ebp            0x119d10            0x119d10 <z_main_stack+860>
   esi            0x0                 0
   edi            0x101aa7            1055399
   eip            0x100459            0x100459 <func_3+16>
   ```

(continues on next page)

(continued from previous page)

```
eflags         0x206              [ PF IF ]
cs             0x8                8
ss             <unavailable>
ds             <unavailable>
es             <unavailable>
fs             <unavailable>
gs             <unavailable>
```

6. Examine the backtrace:

```
(gdb) bt
```

Output from GDB:

```
#0  0x00100459 in func_3 (addr=0x0) at zephyr/rtos/zephyr/samples/hello_world/src/main.
↪c:14
#1  0x00100477 in func_2 (addr=0x0) at zephyr/rtos/zephyr/samples/hello_world/src/main.
↪c:21
#2  0x00100492 in func_1 (addr=0x0) at zephyr/rtos/zephyr/samples/hello_world/src/main.
↪c:28
#3  0x001004c8 in main () at zephyr/rtos/zephyr/samples/hello_world/src/main.c:42
```

Starting the GDB server from within GDB You can use `target remote |` to start the custom GDB server from inside GDB, instead of in a separate shell.

1. Start GDB:

```
<path to SDK>/x86_64-zephyr-elf/bin/x86_64-zephyr-elf-gdb build/zephyr/zephyr.elf
```

2. Inside GDB, start the GDB server using the `--pipe` option:

```
(gdb) target remote | ./scripts/coredump/coredump_gdbserver.py --pipe build/zephyr/
↪zephyr.elf coredump.bin
```

File Format

The core dump binary file consists of one file header, one architecture-specific block, and multiple memory blocks. All numbers in the headers below are little endian.

File Header The file header consists of the following fields:

Table 1: Core dump binary file header

Field	Data Type	Description
ID	`char[2]`	`Z`, `E` as identifier of file.
Header version	`uint16_`	Identify the version of the header. This needs to be incremented whenever the header struct is modified. This allows parser to reject older header versions so it will not incorrectly parse the header.
Target code	`uint16_`	Indicate which target (e.g. architecture or SoC) so the parser can instantiate the correct register block parser.
Pointer size	'uint8_t'	Size of `uintptr_t` in power of 2. (e.g. 5 for 32-bit, 6 for 64-bit). This is needed to accommodate 32-bit and 64-bit target in parsing the memory block addresses.
Flags	`uint8_t`	
Fatal error reason	`unsigned int`	Reason for the fatal error, as the same in `enum k_fatal_error_reason` defined in include/zephyr/fatal.h

Architecture-specific Block The architecture-specific block contains the byte stream of data specific to the target architecture (e.g. CPU registers)

Table 2: Architecture-specific Block

Field	Data Type	Description
ID	`char`	`A` to indicate this is a architecture-specific block.
Header version	`uint16_`	Identify the version of this block. To be interpreted by the target architecture specific block parser.
Number of bytes	`uint16_`	Number of bytes following the header which contains the byte stream for target data. The format of the byte stream is specific to the target and is only being parsed by the target parser.
Register byte stream	`uint8_t`	Contains target architecture specific data.

Memory Block The memory block contains the start and end addresses and the data within the memory region.

Table 3: Memory Block

Field	Data Type	Description
ID	`char`	`M` to indicate this is a memory block.
Header version	`uint16_`	Identify the version of the header. This needs to be incremented whenever the header struct is modified. This allows parser to reject older header versions so it will not incorrectly parse the header.
Start address	`uintptr_`	The start address of the memory region.
End address	`uintptr_`	The end address of the memory region.
Memory byte stream	`uint8_t`	Contains the memory content between the start and end addresses.

Adding New Target

The architecture-specific block is target specific and requires new dumping routine and parser for new targets. To add a new target, the following needs to be done:

1. Add a new target code to the `enum coredump_tgt_code` in include/zephyr/debug/coredump.h.
2. Implement *`arch_coredump_tgt_code_get()`* simply to return the newly introduced target code.
3. Implement *`arch_coredump_info_dump()`* to construct a target architecture block and call *`coredump_buffer_output()`* to output the block to core dump backend.
4. Add a parser to the core dump GDB stub scripts under `scripts/coredump/gdbstubs/`
    1. Extends the `gdbstubs.gdbstub.GdbStub` class.
    2. During `__init__`, store the GDB signal corresponding to the exception reason in `self.gdb_signal`.
    3. Parse the architecture-specific block from `self.logfile.get_arch_data()`. This needs to match the format as implemented in step 3 (inside *`arch_coredump_info_dump()`*).
    4. Implement the abstract method `handle_register_group_read_packet` where it returns the register group as GDB expected. Refer to GDB's code and documentation on what it is expecting for the new target.
    5. Optionally implement `handle_register_single_read_packet` for registers not covered in the `g` packet.
5. Extend `get_gdbstub()` in scripts/coredump/gdbstubs/__init__.py to return the newly implemented GDB stub.

API documentation

group **coredump_apis**

Coredump APIs.

Enums

enum **coredump_query_id**

Query ID.

Values:

enumerator **COREDUMP_QUERY_GET_ERROR**

Returns error code from backend.

enumerator **COREDUMP_QUERY_HAS_STORED_DUMP**

Check if there is a stored coredump from backend.

Returns:
- 1 if there is a stored coredump, 0 if none.
- -ENOTSUP if this query is not supported.
- Otherwise, error code from backend.

enumerator COREDUMP_QUERY_GET_STORED_DUMP_SIZE

Returns:

- coredump raw size from backend, 0 if none.
- -ENOTSUP if this query is not supported.
- Otherwise, error code from backend.

enumerator COREDUMP_QUERY_MAX

Max value for query ID.

enum coredump_cmd_id

Command ID.

Values:

enumerator COREDUMP_CMD_CLEAR_ERROR

Clear error code from backend.

Returns 0 if successful, failed otherwise.

enumerator COREDUMP_CMD_VERIFY_STORED_DUMP

Verify that the stored coredump is valid.

Returns:
- 1 if valid.
- 0 if not valid or no stored coredump.
- -ENOTSUP if this command is not supported.
- Otherwise, error code from backend.

enumerator COREDUMP_CMD_ERASE_STORED_DUMP

Erase the stored coredump.

Returns:
- 0 if successful.
- -ENOTSUP if this command is not supported.
- Otherwise, error code from backend.

enumerator COREDUMP_CMD_COPY_STORED_DUMP

Copy the raw stored coredump.

Returns:
- copied size if successful
- 0 if stored coredump is not found
- -ENOTSUP if this command is not supported.
- Otherwise, error code from backend.

enumerator COREDUMP_CMD_INVALIDATE_STORED_DUMP

Invalidate the stored coredump.

This is faster than erasing the whole partition.

Returns:
- 0 if successful.
- -ENOTSUP if this command is not supported.
- Otherwise, error code from backend.

enumerator COREDUMP_CMD_MAX

Max value for command ID.

Functions

static inline void **coredump**(unsigned int reason, const struct arch_esf *esf, struct *k_thread* *thread)

Perform coredump.

Normally, this is called inside z_fatal_error() to generate coredump when a fatal error is encountered. This can also be called on demand whenever a coredump is desired.

Parameters

- **reason** – Reason for the fatal error
- **esf** – Exception context
- **thread** – Thread information to dump

static inline void **coredump_memory_dump**(uintptr_t start_addr, uintptr_t end_addr)

Dump memory region.

Parameters

- **start_addr** – Start address of memory region to be dumped
- **end_addr** – End address of memory region to be dumped

static inline void **coredump_buffer_output**(uint8_t *buf, size_t buflen)

Output the buffer via coredump.

This outputs the buffer of byte array to the coredump backend. For example, this can be called to output the coredump section containing registers, or a section for memory dump.

Parameters

- **buf** – Buffer to be send to coredump output
- **buflen** – Buffer length

static inline int **coredump_query**(enum *coredump_query_id* query_id, void *arg)

Perform query on coredump subsystem.

Query the coredump subsystem for information, for example, if there is an error.

Parameters

- **query_id** – **[in]** Query ID
- **arg** – **[inout]** Pointer to argument for exchanging information

Returns

Depends on the query

static inline int **coredump_cmd**(enum *coredump_cmd_id* query_id, void *arg)

Perform command on coredump subsystem.

Perform command on coredump subsystem, for example, output the stored coredump via logging.

Parameters

- **cmd_id** – **[in]** Command ID
- **arg** – **[inout]** Pointer to argument for exchanging information

Returns
Depends on the command

struct **coredump_cmd_copy_arg**

#include <*coredump.h*> Coredump copy command (*CORE-DUMP_CMD_COPY_STORED_DUMP*) argument definition.

Public Members

off_t **offset**
Copy offset.

uint8_t ***buffer**
Copy destination buffer.

size_t **length**
Copy length.

group **arch-coredump**

Functions

void **arch_coredump_info_dump**(const struct arch_esf *esf)
Architecture-specific handling during coredump.

This dumps architecture-specific information during coredump.

Parameters
- **esf** – Exception Stack Frame (arch-specific)

uint16_t **arch_coredump_tgt_code_get**(void)
Get the target code specified by the architecture.

4.4.3 GDB stub

- *Overview*
- *Features*
- *Enabling GDB Stub*
  - *Using Serial Backend*
- *Debugging*
  - *Using Serial Backend*
- *Example*

Overview

The gdbstub feature provides an implementation of the GDB Remote Serial Protocol (RSP) that allows you to remotely debug Zephyr using GDB.

The protocol supports different connection types: serial, UDP/IP and TCP/IP. Zephyr currently supports only serial device communication.

The GDB program acts as a client while the Zephyr gdbstub acts as a server. When this feature is enabled, Zephyr stops its execution after `gdb_init()` starts gdbstub service and waits for a GDB connection. Once a connection is established it is possible to synchronously interact with Zephyr. Note that currently it is not possible to asynchronously send commands to the target.

Features

The following features are supported:

- Add and remove breakpoints
- Continue and step the target
- Print backtrace
- Read or write general registers
- Read or write the memory

Enabling GDB Stub

GDB stub can be enabled with the `CONFIG_GDBSTUB` option.

Using Serial Backend The serial backend for GDB stub can be enabled with the `CONFIG_GDBSTUB_SERIAL_BACKEND` option.

Since serial backend utilizes UART devices to send and receive GDB commands,

- If there are spare UART devices on the board, set `zephyr,gdbstub-uart` property of the chosen node to the spare UART device so that `printk()` and log messages are not being printed to the same UART device used for GDB.
- For boards with only one UART device, `printk()` and logging must be disabled if they are also using the same UART device for output. GDB related messages may interleave with log messages which may have unintended consequences. Usually this can be done by disabling `CONFIG_PRINTK` and `CONFIG_LOG`.

Debugging

Using Serial Backend

1. Build with GDB stub and serial backend enabled.
2. Flash built image onto board and reset the board.
   - Execution should now be paused at `gdb_init()`.
3. Execute GDB on development machine and connect to the GDB stub.

   ```
   target remote <serial device>
   ```

   For example,

```
target remote /dev/ttyUSB1
```

4. GDB commands can be used to start debugging.

Example

There is a test application tests/subsys/debug/gdbstub with one of its test cases `debug.gdbstub.breakpoints` demonstrating how the Zephyr GDB stub can be used. The test also has a case to connect to the QEMU's GDB stub implementation (at a custom port `tcp:1235`) as a reference to validate the test script itself.

Run the test with the following command from your *ZEPHYR_BASE* directory:

```
./scripts/twister -p qemu_x86 -T tests/subsys/debug/gdbstub
```

The test should run successfully, and now let's do something similar step-by-step to demonstrate how the Zephyr GDB stub works from the GDB user's perspective.

In the snippets below use and expect your appropriate directories instead of `<SDK install directory>`, `<build_directory>`, `<ZEPHYR_BASE>`.

1. Open two terminal windows.
2. On the first terminal, build and run the test application:

```
# From the root of the zephyr repository
west build -b qemu_x86 tests/subsys/debug/gdbstub -- '-DCONFIG_QEMU_EXTRA_FLAGS="-
↪serial tcp:localhost:5678,server"'
west build -t run
```

   Note how we set `CONFIG_QEMU_EXTRA_FLAGS` to direct QEMU serial console port to the `localhost` TCP port `5678` to wait for a connection from the GDB `remote` command we are going to do on the next steps.

3. On the second terminal, start GDB:

```
<SDK install directory>/x86_64-zephyr-elf/bin/x86_64-zephyr-elf-gdb
```

   1. Tell GDB where to look for the built ELF file:

```
(gdb) symbol-file <build directory>/zephyr/zephyr.elf
```

      Response from GDB:

```
Reading symbols from <build directory>/zephyr/zephyr.elf...
```

   2. Tell GDB to connect to the Zephyr gdbstub serial backend which is exposed earlier as a server through the TCP port `-serial` redirection at QEMU.

```
(gdb) target remote localhost:5678
```

      Response from GDB:

```
Remote debugging using localhost:5678
arch_gdb_init () at <ZEPHYR_BASE>/arch/x86/core/ia32/gdbstub.c:252
252     }
```

      GDB also shows where the code execution is stopped. In this case, it is at arch/x86/core/ia32/gdbstub.c, line 252.

   3. Use command `bt` or `backtrace` to show the backtrace of stack frames.

```
(gdb) bt
#0  arch_gdb_init () at <ZEPHYR_BASE>/arch/x86/core/ia32/gdbstub.c:252
#1  0x00104140 in gdb_init () at <ZEPHYR_BASE>/zephyr/subsys/debug/gdbstub.c:852
#2  0x00109c13 in z_sys_init_run_level (level=INIT_LEVEL_PRE_KERNEL_2) at <ZEPHYR_
↪BASE>/kernel/init.c:360
#3  0x00109e73 in z_cstart () at <ZEPHYR_BASE>/kernel/init.c:630
#4  0x00104422 in z_prep_c (arg=0x1245bc <x86_cpu_boot_arg>) at <ZEPHYR_BASE>/
↪arch/x86/core/prep_c.c:80
#5  0x001000c9 in __csSet () at <ZEPHYR_BASE>/arch/x86/core/ia32/crt0.S:290
#6  0x001245bc in uart_dev ()
#7  0x00134988 in z_interrupt_stacks ()
#8  0x00000000 in ?? ()
```

4. Use command `list` to show the source code and surroundings where code execution is stopped.

```
(gdb) list
247             __asm__ volatile ("int3");
248
249     #ifdef CONFIG_GDBSTUB_TRACE
250             printk("gdbstub:%s GDB is connected\n", __func__);
251     #endif
252     }
253
254     /* Hook current IDT. */
255     _EXCEPTION_CONNECT_NOCODE(z_gdb_debug_isr, IV_DEBUG, 3);
256     _EXCEPTION_CONNECT_NOCODE(z_gdb_break_isr, IV_BREAKPOINT, 3);
```

5. Use command `s` or `step` to step through program until it reaches a different source line. Now that it finished executing *`arch_gdb_init()`* and is continuing in `gdb_init()`.

```
(gdb) s
gdb_init () at <ZEPHYR_BASE>/subsys/debug/gdbstub.c:857
857     return 0;
```

```
(gdb) list
852             arch_gdb_init();
853
854     #ifdef CONFIG_GDBSTUB_TRACE
855             printk("gdbstub:%s exit\n", __func__);
856     #endif
857             return 0;
858     }
859
860     #ifdef CONFIG_XTENSA
861     /*
```

6. Use command `br` or `break` to setup a breakpoint. For this example set up a breakpoint at `main()`, and let code execution continue without any intervention using command `c` (or `continue`).

```
(gdb) break main
Breakpoint 1 at 0x10064d: file <ZEPHYR_BASE>/tests/subsys/debug/gdbstub/src/main.
↪c, line 27.
```

```
(gdb) continue
Continuing.
```

Once code execution reaches `main()`, execution will be stopped and GDB prompt returns.

```
Breakpoint 1, main () at <ZEPHYR_BASE>/tests/subsys/debug/gdbstub/src/main.c:27
27              printk("%s():enter\n", __func__);
```

Now GDB is waiting at the beginning of `main()`:

```
(gdb) list
22
23      int main(void)
24      {
25              int ret;
26
27              printk("%s():enter\n", __func__);
28              ret = test();
29              printk("ret=%d\n", ret);
30              return 0;
31      }
```

7. To examine the value of `ret`, the command `p` or `print` can be used.

```
(gdb) p ret
$1 = 1273788
```

Since `ret` has not been initialized, it contains some random value.

8. If step (`s` or `step`) is used here, it will continue execution skipping the interior of `test()`. To examine code execution inside `test()`, a breakpoint can be set for `test()`, or simply using `si` (or `stepi`) to execute one machine instruction, where it has the side effect of going into the function. The GDB command `finish` can be used to continue execution without intervention until the function returns.

```
(gdb) finish
Run till exit from #0  test () at <ZEPHYR_BASE>/tests/subsys/debug/gdbstub/src/
↪main.c:17
0x00100667 in main () at <ZEPHYR_BASE>/tests/subsys/debug/gdbstub/src/main.c:28
28              ret = test();
Value returned is $2 = 30
```

9. Examine `ret` again which should have the return value from `test()`. Sometimes, the assignment is not done until another `step` is issued, as in this case. This is due to the assignment code is done after returning from function. The assignment code is generated by the toolchain as machine instructions which are not visible when viewing the corresponding C source file.

```
(gdb) p ret
$3 = 1273788
(gdb) step
29              printk("ret=%d\n", ret);
(gdb) p ret
$4 = 30
```

10. If `continue` is issued here, code execution will continue indefinitely as there are no breakpoints to further stop execution. Breaking execution in GDB via `Ctrl-C` does not currently work as the Zephyr gdbstub does not support this functionality yet. Switch to the first console with QEMU running the Zephyr image and stop it manually with `Ctrl+a x`. When the same test is executed by Twister, it automatically takes care of stopping the QEMU instance.

4.4.4 Cortex-M Debug Monitor

Monitor mode debugging is a Cortex-M feature, that provides a non-halting approach to debugging. With this it's possible to continue the execution of high-priority interrupts, even when waiting on a breakpoint. This strategy makes it possible to debug time-sensitive software, that would otherwise crash when the core halts (e.g. applications that need to keep communication links alive).

Zephyr provides support for enabling and configuring the Debug Monitor exception. It also contains a ready implementation of the interrupt, which can be used with SEGGER J-Link debuggers.

Configuration

Configure this module using the following options.

- `CONFIG_CORTEX_M_DEBUG_MONITOR_HOOK`: enable the module. This option, by itself, requires an implementation of debug monitor interrupt that will be executed every time the program enters a breakpoint.

With a SEGGER debug probe, it's possible to use a ready, SEGGER-provided implementation of the interrupt.

- `CONFIG_SEGGER_DEBUGMON`: enables SEGGER debug monitor interrupt. Can be used with SEGGER JLinkGDBServer and a SEGGER debug probe.

Usage

When monitor mode debugging is enabled, entering a breakpoint will not halt the processor, but rather generate an interrupt with ISR implemented under `z_arm_debug_monitor` symbol. `CONFIG_CORTEX_M_DEBUG_MONITOR_HOOK` config configures this interrupt to be the lowest available priority, which will allow other interrupts to execute while processor spins on a breakpoint.

Using SEGGER-provided ISR The ready implementation provided with `CONFIG_SEGGER_DEBUGMON` provides functionality required to debug in the monitor mode using regular GDB commands. Steps to configure SEGGER debug monitor:

1. Build a sample with `CONFIG_CORTEX_M_DEBUG_MONITOR_HOOK`` and `CONFIG_SEGGER_DEBUGMON` configs enabled.
2. Attach JLink GDB server to the target. Example linux command: `JLinkGDBServerCLExe -device <device> -if swd`.
3. Connect to the server with your GDB installation. Example linux command: `arm-none-eabi-gdb --ex="file build/zephyr.elf" --ex="target remote localhost:2331"`.
4. Enable monitor mode debugging in GDB using command: `monitor exec SetMonModeDebug=1`.

After these steps use regular gdb commands to debug your program.

Using other custom ISR In order to provide a custom debug monitor interrupt, override `z_arm_debug_monitor` symbol. Additionally, manual configuration of some registers is required (see debug monitor sample).

4.4.5 MIPI STP Decoder

The MIPI System Trace Protocol (MIPI STP) was developed as a generic base protocol that can be shared by multiple application-specific trace protocols. It serves as a wrapper protocol that merges disparate streams that typically contain different trace protocols from different trace sources. Stream consists of opcode (shortest is 4 bit long) followed by optional data and optional timestamp. There are opcodes for data (8, 16, 32, 64 bit data marked/not marked, with or without timestamp), stream recognition (master and channel), synchronization (ASYNC opcode) and others.

One example where protocol is used is ARM Coresight STM (System Trace Macrocell) where data written to Stimulus Port registers maps directly to STP stream.

This module can be used to perform on-chip decoding of the data stream. STP v2 is used.

Usage

Decoder is initialized with a callback. A callback is called on each decoded opcode. Decoder has internal state since there are dependency between opcodes (e.g. timestamp can be relative). Decoder can be in synchronization or not. Initial state is configurable. If decoder is not synchronized to the stream then it decodes each nibble in search for ASYNC opcode. Loss of synchronization can be indicated to the decoder by calling *mipi_stp_decoder_sync_loss()*. *mipi_stp_decoder_decode()* is used to decode the data.

Limitations

There are following limitations:

- Decoder supports only little endian architectures.
- When decoding nibbles, it is more efficient when core supports unaligned memory access. Implementation supports optimized version with unaligned memory access and generic one. Optimized version is used for ARM Cortex-M (expect for M0).
- Limited set of the most common opcodes is implemented.

API documentation

group `mipi_stp_decoder_apis`

Defines

`STP_DECODER_TYPE2STR`(_type)

Convert type to a string literal.

Parameters

- **`_type`** – type

Returns

String literal.

Typedefs

typedef void (*`mipi_stp_decoder_cb`)(enum *mipi_stp_decoder_ctrl_type* type, union *mipi_stp_decoder_data* data, uint64_t *ts, bool marked)

Callback signature.

Callback is called whenever an element from STPv2 stream is decoded.

> **Note**
>
> Callback is called with interrupts locked.

Param type
Type. See *mipi_stp_decoder_ctrl_type*.

Param data
Data. Data associated with a given `type`.

Param ts
Timestamp. Present if not NULL.

Param marked
Set to true if opcode was marked.

Enums

enum `mipi_stp_decoder_ctrl_type`

STPv2 opcodes.

Values:

enumerator `STP_DATA4` = 1

enumerator `STP_DATA8` = 2

enumerator `STP_DATA16` = 4

enumerator `STP_DATA32` = 8

enumerator `STP_DATA64` = 16

enumerator `STP_DECODER_NULL` = 128

enumerator `STP_DECODER_MASTER`

enumerator `STP_DECODER_MERROR`

enumerator `STP_DECODER_CHANNEL`

enumerator `STP_DECODER_VERSION`

enumerator STP_DECODER_FREQ

enumerator STP_DECODER_GERROR

enumerator STP_DECODER_FLAG

enumerator STP_DECODER_ASYNC

enumerator STP_DECODER_NOT_SUPPORTED

Functions

int **mipi_stp_decoder_init**(const struct *mipi_stp_decoder_config* *config)

Initialize the decoder.

Parameters

- **config** – Configuration.

Return values

- **0** – On successful initialization.
- **negative** – On failure.

int **mipi_stp_decoder_decode**(const uint8_t *data, size_t len)

Decode STPv2 stream.

Function decodes the stream and calls the callback for every decoded element.

Parameters

- **data** – Data.
- **len** – Data length.

Return values

- **0** – On successful decoding.
- **negative** – On failure.

void **mipi_stp_decoder_sync_loss**(void)

Indicate synchronization loss.

If detected, then decoder starts to look for ASYNC marker and drops all data until ASYNC is found. Synchronization can be lost when there is data loss (e.g. due to overflow).

union **mipi_stp_decoder_data**

#include <mipi_stp_decoder.h> Union with data associated with a given STP opcode.

Public Members

uint16_t **id**

ID - used for master and channel.

uint64_t **freq**
Frequency.

uint32_t **ver**
Version.

uint32_t **err**
Error code.

uint32_t **dummy**
Dummy.

uint64_t **data**
Data.

struct **mipi_stp_decoder_config**
#include <mipi_stp_decoder.h> Decoder configuration.

Public Members

bool **start_out_of_sync**
Indicates that decoder start in out of sync state.

mipi_stp_decoder_cb **cb**
Callback.

4.4.6 Symbol Table (Symtab)

The Symtab module, when enabled, will generate full symbol table during the Zephyr linking stage that keep tracks of the information about the functions' name and address, for advanced application with a lot of functions, this is expected to consume a sizable amount of ROM.

Currently, this is being used to look up the function names during a stack trace in supported architectures.

Usage

Application can gain access to the symbol table data structure by including the `symtab.h` header file and call *symtab_get()*. For now, we only provide *symtab_find_symbol_name()* function to look-up the symbol name and offset of an address. More advanced functionalities and be achieved by directly accessing the members of the data structure.

Configuration

Configure this module using the following options.

- `CONFIG_SYMTAB`: enable the generation of the symbol table.

API documentation

group `symtab_apis`

Functions

const struct *symtab_info* *const `symtab_get`(void)

Get the pointer to the symbol table.

Returns
Pointer to the symbol table.

const char *const `symtab_find_symbol_name`(uintptr_t addr, uint32_t *offset)

Find the symbol name with a binary search.

Parameters

- `addr` – **[in]** Address of the symbol to find
- `offset` – **[out]** Offset of the symbol from the nearest symbol. If the symbol can't be found, this will be 0.

Returns
Name of the nearest symbol if found, otherwise "?" is returned.

struct `symtab_info`

#include <symtab.h>

4.5 Device Management

4.5.1 MCUmgr

Overview

The management subsystem allows remote management of Zephyr-enabled devices. The following management operations are available:

- Image management
- File System management
- OS management
- Settings (config) management
- Shell management
- Statistic management
- Zephyr management

over the following transports:

- BLE (Bluetooth Low Energy)
- Serial (UART)
- UDP over IP

The management subsystem is based on the Simple Management Protocol (SMP) provided by MCUmgr, an open source project that provides a management subsystem that is portable across multiple real-time operating systems.

The management subsystem is located in subsys/mgmt/ inside of the Zephyr tree.

Additionally, there is a sample server that provides management functionality over BLE and serial.

Tools/libraries

There are various tools and libraries available which enable usage of MCUmgr functionality on a device which are listed below. Note that these tools are not part of or related to the Zephyr project.

Table 4: Tools and Libraries for MCUmgr

Name	OS support	Transports	Groups							Type	Language
			OS	IMG	Stat	Settings	FS	Shell	Zephyr		
AuTerm	Windows, Linux, macOS	Serial, Bluetooth, UDP	✓	✓	✓	✓	✓	✓	✓	App	C++ (Qt)
mcumgr-client	Windows, Linux, macOS	Serial	×	✓	×	×	×	×	×	App	Rust
mcumgr-web	Windows, Linux, macOS	Bluetooth	×	✓	×	×	×	×	×	Web (chrome only)	Javascript
nRF Connect Device Manager: Android and iOS	iOS, Android	Bluetooth	✓	✓	✓	✓	✓	✓	✓	Library, App	Java, Kotlin, Swift
Zephyr MCUmgr client (in-tree)	Linux, Zephyr	Serial, Bluetooth, UDP	✓	✓	×	×	×	×	×	Library	C

Note that a tick for a particular group indicates basic support for that group in the code, it is possible that not all commands/features of a group are supported by the implementation.

J-Link Virtual MSD Interaction Note

On boards where a J-Link OB is present which has both CDC and MSC (virtual Mass Storage Device, also known as drag-and-drop) support, the MSD functionality can prevent MCUmgr commands over the CDC UART port from working due to how USB endpoints are configured in the J-Link firmware (for example on the Nordic nrf52840dk/nrf52840 board) because of limiting the maximum packet size (most likely to occur when using image management commands for updating firmware). This issue can be resolved by disabling MSD functionality on the J-Link device, follow the instructions on nordic_segger_msd to disable MSD support.

Bootloader Integration

The *Device Firmware Upgrade* subsystem integrates the management subsystem with the bootloader, providing the ability to send and upgrade a Zephyr image to a device.

Currently only the MCUboot bootloader is supported. See *MCUboot* for more information.

Discord channel

Developers welcome!

- Discord mcumgr channel: https://discord.com/invite/Ck7jw53nU2

API Reference

group `mcumgr_mgmt_api`

MCUmgr mgmt API.

Since
1.11

Version
1.0.0

Defines

`MGMT_CTXT_SET_RC_RSN`(mc, rsn)

`MGMT_CTXT_RC_RSN`(mc)

`MGMT_RETURN_CHECK`(ok)

Used at end of MCUmgr handlers to return an error if the message size limit was reached, or OK if it was not.

`MGMT_HDR_SIZE`

Typedefs

typedef void *(*`mgmt_alloc_rsp_fn`)(const void *src_buf, void *arg)

Allocates a buffer suitable for holding a response.

If a source buf is provided, its user data is copied into the new buffer.

Param src_buf
An optional source buffer to copy user data from.

Param arg
Optional streamer argument.

Return
Newly-allocated buffer on success NULL on failure.

typedef void (*`mgmt_reset_buf_fn`)(void *buf, void *arg)

Resets a buffer to a length of 0.

The buffer's user data remains, but its payload is cleared.

Param buf
The buffer to reset.

Param arg
Optional streamer argument.

typedef int (*`mgmt_handler_fn`)(struct smp_streamer *ctxt)

Processes a request and writes the corresponding response.

A separate handler is required for each supported op-ID pair.

Param ctxt
The mcumgr context to use.

Return
0 if a response was successfully encoded, *mcumgr_err_t* code on failure.

Enums

enum `mcumgr_op_t`

Opcodes; encoded in first byte of header.

Values:

enumerator `MGMT_OP_READ` = 0

Read op-code.

enumerator `MGMT_OP_READ_RSP`

Read response op-code.

enumerator `MGMT_OP_WRITE`

Write op-code.

enumerator `MGMT_OP_WRITE_RSP`

Write response op-code.

enum `mcumgr_group_t`

MCUmgr groups.

The first 64 groups are reserved for system level mcumgr commands. Per-user commands are then defined after group 64.

Values:

enumerator `MGMT_GROUP_ID_OS` = 0

OS (operating system) group.

enumerator `MGMT_GROUP_ID_IMAGE`

Image management group, used for uploading firmware images.

enumerator `MGMT_GROUP_ID_STAT`

Statistic management group, used for retrieving statistics.

enumerator `MGMT_GROUP_ID_SETTINGS`

Settings management (config) group, used for reading/writing settings.

enumerator `MGMT_GROUP_ID_LOG`

Log management group (unused)

enumerator `MGMT_GROUP_ID_CRASH`

Crash group (unused)

enumerator `MGMT_GROUP_ID_SPLIT`

Split image management group (unused)

enumerator `MGMT_GROUP_ID_RUN`

Run group (unused)

enumerator `MGMT_GROUP_ID_FS`

FS (file system) group, used for performing file IO operations.

enumerator `MGMT_GROUP_ID_SHELL`

Shell management group, used for executing shell commands.

enumerator `MGMT_GROUP_ID_PERUSER` = 64

User groups defined from 64 onwards.

enumerator `ZEPHYR_MGMT_GRP_BASIC` = (*MGMT_GROUP_ID_PERUSER* - 1)

Zephyr-specific groups decrease from PERUSER to avoid collision with upstream and user-defined groups.

Zephyr-specific: Basic group

enum `mcumgr_err_t`

MCUmgr error codes.

Values:

enumerator MGMT_ERR_EOK = 0
No error (success).

enumerator MGMT_ERR_EUNKNOWN
Unknown error.

enumerator MGMT_ERR_ENOMEM
Insufficient memory (likely not enough space for CBOR object).

enumerator MGMT_ERR_EINVAL
Error in input value.

enumerator MGMT_ERR_ETIMEOUT
Operation timed out.

enumerator MGMT_ERR_ENOENT
No such file/entry.

enumerator MGMT_ERR_EBADSTATE
Current state disallows command.

enumerator MGMT_ERR_EMSGSIZE
Response too large.

enumerator MGMT_ERR_ENOTSUP
Command not supported.

enumerator MGMT_ERR_ECORRUPT
Corrupt.

enumerator MGMT_ERR_EBUSY
Command blocked by processing of other command.

enumerator MGMT_ERR_EACCESSDENIED
Access to specific function, command or resource denied.

enumerator MGMT_ERR_UNSUPPORTED_TOO_OLD
Requested SMP MCUmgr protocol version is not supported (too old)

enumerator MGMT_ERR_UNSUPPORTED_TOO_NEW
Requested SMP MCUmgr protocol version is not supported (too new)

enumerator MGMT_ERR_EPERUSER = 256
User errors defined from 256 onwards.

Functions

void **mgmt_register_group**(struct *mgmt_group* *group)

Registers a full command group.

Parameters

- **group** – The group to register.

void **mgmt_unregister_group**(struct *mgmt_group* *group)

Unregisters a full command group.

Parameters

- **group** – The group to register.

const struct *mgmt_handler* ***mgmt_find_handler**(uint16_t group_id, uint16_t command_id)

Finds a registered command handler.

Parameters

- **group_id** – The group of the command to find.
- **command_id** – The ID of the command to find.

Returns

The requested command handler on success; NULL on failure.

const struct *mgmt_group* ***mgmt_find_group**(uint16_t group_id)

Finds a registered command group.

Parameters

- **group_id** – The group id of the command group to find.

Returns

The requested group on success; NULL on failure.

const struct *mgmt_handler* ***mgmt_get_handler**(const struct *mgmt_group* *group, uint16_t command_id)

Finds a registered command handler.

Parameters

- **group** – The group of the command to find.
- **command_id** – The ID of the command to find.

Returns

The requested command handler on success; NULL on failure.

struct **mgmt_handler**

#include <mgmt.h> Read handler and write handler for a single command ID.

Set use_custom_payload to true when using a user defined payload type

struct **mgmt_group**

#include <mgmt.h> A collection of handlers for an entire command group.

Public Members

sys_snode_t **node**

Entry list node.

const struct *mgmt_handler* **mg_handlers**
: Array of handlers; one entry per command ID.

uint16_t **mg_group_id**
: The numeric ID of this group.

4.5.2 MCUmgr handlers

Overview

MCUmgr functions by having group handlers which identify a group of functions relating to a specific management area, which is addressed with a 16-bit identification value, *mcumgr_group_t* contains the management groups available in Zephyr with their corresponding group ID values. The group ID is included in SMP headers to identify which group a command belongs to, there is also an 8-bit command ID which identifies the function of that group to execute - see *SMP Protocol Specification* for details on the SMP protocol and header. There can only be one registered group per unique ID.

Implementation

MCUmgr handlers can be added externally by application code or by module code, they do not have to reside in the upstream Zephyr tree to be usable. The first step to creating a handler is to create the folder structure for it, the typical Zephyr MCUmgr group layout is as follows:

```
<dir>/grp/<grp_name>_mgmt/
├── CMakeLists.txt
├── Kconfig
├── include
├──── <grp_name>_mgmt.h
├──── <grp_name>_mgmt_callbacks.h
├── src
└──── <grp_name>_mgmt.c
```

Note that the header files in upstream Zephyr MCUmgr handlers reside in the `zephyr/include/zephyr/mgmt/mcumgr/grp/<grp_name>_mgmt` directory to allow the files to be globally included by applications.

Initial header <grp_name>_mgmt.h The purpose of the header file is to provide defines which can be used by the MCUmgr handler itself and application code, e.g. to reference the command IDs for executing functions. An example file would look similar to:

```
/*
 * Copyright (c) 2023 Nordic Semiconductor ASA
 *
 * SPDX-License-Identifier: Apache-2.0
 */
#ifndef H_EXAMPLE_MGMT_
#define H_EXAMPLE_MGMT_

#ifdef __cplusplus
extern "C" {
#endif

/**
 * Group ID for example management group.
```

(continues on next page)

(continued from previous page)

```
 */
#define MGMT_GROUP_ID_EXAMPLE  MGMT_GROUP_ID_PERUSER

/**
 * Command IDs for example management group.
 */
#define EXAMPLE_MGMT_ID_TEST   0
#define EXAMPLE_MGMT_ID_OTHER  1

/**
 * Command result codes for example management group.
 */
enum example_mgmt_err_code_t {
        /** No error, this is implied if there is no ret value in the response */
        EXAMPLE_MGMT_ERR_OK = 0,

        /** Unknown error occurred. */
        EXAMPLE_MGMT_ERR_UNKNOWN,

        /** The provided value is not wanted at this time. */
        EXAMPLE_MGMT_ERR_NOT_WANTED,

        /** The provided value was rejected by a hook. */
        EXAMPLE_MGMT_ERR_REJECTED_BY_HOOK,
};

#ifdef __cplusplus
}
#endif

#endif
```

This provides the defines for 2 command `test` and `other` and sets up the SMP version 2 error responses (which have unique error codes per group as opposed to the legacy SMP version 1 error responses that return a *mcumgr_err_t* - there should always be an OK error code with the value 0 and an unknown error code with the value 1. The above example then adds an error code of `not wanted` with value 2. In addition, the group ID is set to be *MGMT_GROUP_ID_PERUSER*, which is the start group ID for user-defined groups, note that group IDs need to be unique so other custom groups should use different values, a central index header file (as upstream Zephyr has) can be used to distribute group IDs more easily.

Initial header <grp_name>_mgmt_callbacks.h The purpose of the header file is to provide defines which can be used by the MCUmgr handler itself and application code, e.g. to reference the command IDs for executing functions. An example file would look similar to:

```
/*
 * Copyright (c) 2023 Nordic Semiconductor ASA
 *
 * SPDX-License-Identifier: Apache-2.0
 */
#ifndef H_MCUMGR_EXAMPLE_MGMT_CALLBACKS_
#define H_MCUMGR_EXAMPLE_MGMT_CALLBACKS_
#include <stdint.h>
#include <zephyr/mgmt/mcumgr/mgmt/callbacks.h>

#ifdef __cplusplus
extern "C" {
#endif

```

(continues on next page)

(continued from previous page)

```
/* This is the event ID for the example group */
#define MGMT_EVT_GRP_EXAMPLE MGMT_EVT_GRP_USER_CUSTOM_START

/* MGMT event opcodes for example management group */
enum example_mgmt_group_events {
        /* Callback when the other command is received, data is example_mgmt_other_data */
        MGMT_EVT_OP_EXAMPLE_OTHER = MGMT_DEF_EVT_OP_ID(MGMT_EVT_GRP_EXAMPLE, 0),

        /* Used to enable all smp_group events */
        MGMT_EVT_OP_EXAMPLE_MGMT_ALL  = MGMT_DEF_EVT_OP_ALL(MGMT_EVT_GRP_EXAMPLE),
};

/* Structure provided in the #MGMT_EVT_OP_EXAMPLE_OTHER notification callback */
struct example_mgmt_other_data {
        /* Contains the user supplied value */
        uint32_t user_value;
};

#ifdef __cplusplus
}
#endif

#endif
```

This sets up a single event which application (or module) code can register for to receive a callback when the function handler is executed, which allows the flow of the handler to be changed (i.e. to return an error instead of continuing). The event group ID is set to *MGMT_EVT_GRP_USER_CUSTOM_START*, which is the start event ID for user-defined groups, note that event IDs need to be unique so other custom groups should use different values, a central index header file (as upstream Zephyr has) can be used to distribute event IDs more easily.

Initial source <grp_name>_mgmt.c The purpose of this source file is to handle the incoming MCUmgr commands, provide responses, and register the transport with MCUmgr so that commands will be sent to it. An example file would look similar to:

```
/*
 * Copyright (c) 2023 Nordic Semiconductor ASA
 *
 * SPDX-License-Identifier: Apache-2.0
 */
#include <zephyr/kernel.h>
#include <zephyr/mgmt/mcumgr/mgmt/mgmt.h>
#include <zephyr/mgmt/mcumgr/smp/smp.h>
#include <zephyr/mgmt/mcumgr/mgmt/handlers.h>
/* The below should be updated with the real name of the file */
#include <example_mgmt.h>
#include <zephyr/logging/log.h>
#include <assert.h>
#include <limits.h>
#include <string.h>
#include <stdio.h>
#include <zcbor_common.h>
#include <zcbor_decode.h>
#include <zcbor_encode.h>
#include <mgmt/mcumgr/util/zcbor_bulk.h>

#if defined(CONFIG_MCUMGR_MGMT_NOTIFICATION_HOOKS)
#include <zephyr/mgmt/mcumgr/mgmt/callbacks.h>
#if defined(CONFIG_MCUMGR_GRP_EXAMPLE_OTHER_HOOK)
```

(continues on next page)

(continued from previous page)

```
/* The below should be updated with the real name of the file */
#include <example_mgmt_callbacks.h>
#endif
#endif

LOG_MODULE_REGISTER(mcumgr_example_grp, CONFIG_MCUMGR_GRP_EXAMPLE_LOG_LEVEL);
/* Example function with "read" command support, requires both parameters are supplied */
static int example_mgmt_test(struct smp_streamer *ctxt)
{
        uint32_t uint_value = 0;
        zcbor_state_t *zse = ctxt->writer->zs;
        zcbor_state_t *zsd = ctxt->reader->zs;
        bool ok;
        struct zcbor_string string_value = { 0 };
        size_t decoded;
        struct zcbor_map_decode_key_val example_test_decode[] = {
                ZCBOR_MAP_DECODE_KEY_DECODER("uint_key", zcbor_uint32_decode, &uint_value),
                ZCBOR_MAP_DECODE_KEY_DECODER("string_key", zcbor_tstr_decode, &string_
↪value),
        };

        LOG_DBG("Example test function called");

        ok = zcbor_map_decode_bulk(zsd, example_test_decode, ARRAY_SIZE(example_test_
↪decode),
                                   &decoded) == 0;
        /* Check that both parameters were supplied and that the value of "string_key" is␣
↪not
         * empty
         */
        if (!ok || string_value.len == 0 || !zcbor_map_decode_bulk_key_found(
                        example_test_decode, ARRAY_SIZE(example_test_decode), "uint_key
↪")) {
                return MGMT_ERR_EINVAL;
        }

        /* If the value of "uint_key" is over 50, return an error of "not wanted" */
        if (uint_value > 50) {
                ok = smp_add_cmd_err(zse, MGMT_GROUP_ID_EXAMPLE, EXAMPLE_MGMT_ERR_NOT_
↪WANTED);
                goto end;
        }

        /* Otherwise, return an integer value of 4691 */
        ok = zcbor_tstr_put_lit(zse, "return_int") &&
             zcbor_int32_put(zse, 4691);

end:
        /* If "ok" is false, then there was an error processing the output cbor message,␣
↪which
         * likely indicates a lack of available memory
         */
        return (ok ? MGMT_ERR_EOK : MGMT_ERR_EMSGSIZE);
}

/* Example function with "write" command support */
static int example_mgmt_other(struct smp_streamer *ctxt)
{
        uint32_t user_value = 0;
        zcbor_state_t *zse = ctxt->writer->zs;
        zcbor_state_t *zsd = ctxt->reader->zs;
```

(continues on next page)

(continued from previous page)

```
        bool ok;
        size_t decoded;
        struct zcbor_map_decode_key_val example_other_decode[] = {
                ZCBOR_MAP_DECODE_KEY_DECODER("user_value", zcbor_uint32_decode, &user_
↪value),
        };

#if defined(CONFIG_MCUMGR_GRP_EXAMPLE_OTHER_HOOK)
        struct example_mgmt_other_data other_data;
        enum mgmt_cb_return status;
        int32_t err_rc;
        uint16_t err_group;
#endif

        LOG_DBG("Example other function called");

        ok = zcbor_map_decode_bulk(zsd, example_other_decode, ARRAY_SIZE(example_other_
↪decode),
                                   &decoded) == 0;

        /* The supplied value is optional, therefore do not return an error if it was not
         * provided
         */
        if (!ok) {
                return MGMT_ERR_EINVAL;
        }

#if defined(CONFIG_MCUMGR_GRP_EXAMPLE_OTHER_HOOK)
        /* Send request to application to check what to do */
        other_data.user_value = user_value;
        status = mgmt_callback_notify(MGMT_EVT_OP_EXAMPLE_OTHER, &other_data, sizeof(other_
↪data),
                                      &err_rc, &err_group);
        if (status != MGMT_CB_OK) {
                /* If a callback returned an RC error, exit out, if it returned a group␣
↪error
                 * code, add the error code to the response and return to the calling␣
↪function to
                 * have it sent back to the client
                 */
                if (status == MGMT_CB_ERROR_RC) {
                        return err_rc;
                }

                ok = smp_add_cmd_err(zse, err_group, (uint16_t)err_rc);
                goto end;
        }
#endif
        /* Return some dummy data to the client */
        ok = zcbor_tstr_put_lit(zse, "return_string") &&
             zcbor_tstr_put_lit(zse, "some dummy data!");

#if defined(CONFIG_MCUMGR_GRP_EXAMPLE_OTHER_HOOK)
end:
#endif
        /* If "ok" is false, then there was an error processing the output cbor message,␣
↪which
         * likely indicates a lack of available memory
         */
        return (ok ? MGMT_ERR_EOK : MGMT_ERR_EMSGSIZE);
}
```

(continues on next page)

(continued from previous page)

```
135
136 #ifdef CONFIG_MCUMGR_SMP_SUPPORT_ORIGINAL_PROTOCOL
137 /* This is a lookup function that converts from SMP version 2 group error codes to legacy
138  * MCUmgr error codes, it is only included if support for the original protocol is enabled.
139  * Note that in SMP version 2, MCUmgr error codes can still be returned, but are to be used
140  * only for general SMP/MCUmgr errors. The success/OK error code is not used in translation
141  * functions as it is automatically handled by the base SMP code.
142  */
143 static int example_mgmt_translate_error_code(uint16_t err)
144 {
145         int rc;
146
147         switch (err) {
148         case EXAMPLE_MGMT_ERR_NOT_WANTED:
149                 rc = MGMT_ERR_ENOENT;
150                 break;
151
152         case EXAMPLE_MGMT_ERR_REJECTED_BY_HOOK:
153                 rc = MGMT_ERR_EBADSTATE;
154                 break;
155
156         case EXAMPLE_MGMT_ERR_UNKNOWN:
157         default:
158                 rc = MGMT_ERR_EUNKNOWN;
159         }
160
161         return rc;
162 }
163 #endif
164
165 static const struct mgmt_handler example_mgmt_handlers[] = {
166         [EXAMPLE_MGMT_ID_TEST] = {
167                 .mh_read = example_mgmt_test,
168                 .mh_write = NULL,
169         },
170         [EXAMPLE_MGMT_ID_OTHER] = {
171                 .mh_read = NULL,
172                 .mh_write = example_mgmt_other,
173         },
174 };
175
176 static struct mgmt_group example_mgmt_group = {
177         .mg_handlers = example_mgmt_handlers,
178         .mg_handlers_count = ARRAY_SIZE(example_mgmt_handlers),
179         .mg_group_id = MGMT_GROUP_ID_EXAMPLE,
180 #ifdef CONFIG_MCUMGR_SMP_SUPPORT_ORIGINAL_PROTOCOL
181         .mg_translate_error = example_mgmt_translate_error_code,
182 #endif
183 };
184
185 static void example_mgmt_register_group(void)
186 {
187         /* This function is called during system init before main() is invoked, if the
188          * handler needs to set anything up before it can be used, it should do it here.
189          * This register the group so that clients can call the function handlers.
190          */
191         mgmt_register_group(&example_mgmt_group);
192 }
193
194 MCUMGR_HANDLER_DEFINE(example_mgmt, example_mgmt_register_group);
```

The above code creates 2 function handlers, `test` which supports read requests and takes 2 re-

quired parameters, and `other` which supports write requests and takes 1 optional parameter, this function handler has an optional notification callback feature that allows other parts of the code to listen for the event and take any required actions that are necessary or prevent further execution of the function by returning an error, further details on MCUmgr callback functionality can be found on *MCUmgr Callbacks*.

Note that other code referencing callbacks for custom MCUmgr handlers needs to include both the base Zephyr callback include file and the custom handler callback file, only in-tree Zephyr handler headers are included when including the upstream Zephyr callback header file.

Initial Kconfig The purpose of the Kconfig file is to provide options which users can enable or change relating to the functionality of the handler being implemented. An example file would look similar to:

```
# Copyright Nordic Semiconductor ASA 2023. All rights reserved.
# SPDX-License-Identifier: Apache-2.0
# The Kconfig file is dedicated to example management group of
# of MCUmgr subsystem and provides Kconfig options to configure
# group commands behaviour and other aspects.
#
# Options defined in this file should be prefixed:
#  MCUMGR_GRP_EXAMPLE_ -- general group options;
#
# When adding Kconfig options, that control the same feature,
# try to group them together by the same stem after prefix.
if MCUMGR

menuconfig MCUMGR_GRP_EXAMPLE_APP
        bool "MCUmgr handlers for example management (app)"
        select MCUMGR_SMP_CBOR_MIN_DECODING_LEVEL_2
        default y
        help
          Enables MCUmgr handlers for example management. This demonstrates the
          file at application-level.

if MCUMGR_GRP_EXAMPLE_APP
config MCUMGR_GRP_EXAMPLE_OTHER_HOOK
        bool "Other hook"
        depends on MCUMGR_MGMT_NOTIFICATION_HOOKS
        help
          Allows applications to receive callback when the "other" example
          management function is called

module = MCUMGR_GRP_EXAMPLE
module-str = mcumgr_grp_example
source "subsys/logging/Kconfig.template.log_config"

endif

endif

source "Kconfig.zephyr"
```

Initial CMakeLists.txt The CMakeLists.txt file is used by the build system to setup files to compile, include directories to add and specify options that can be changed. A basic file only need to include the source files if the Kconfig options are enabled. An example file would look similar to:

Zephyr module

```
# Copyright (c) 2023 Nordic Semiconductor ASA
# SPDX-License-Identifier: Apache-2.0

if(CONFIG_MCUMGR_GRP_EXAMPLE_MODULE)
  zephyr_library(mgmt_mcumgr_grp_example)
  # The below should be updated with the real name of the file
  zephyr_library_sources(src/example_mgmt.c)
  zephyr_include_directories(include)
endif()
```

Application

```
if(CONFIG_MCUMGR_GRP_EXAMPLE_APP)
  target_sources(app PRIVATE example_as_module/src/example_mgmt.c)
  zephyr_include_directories(example_as_module/include)
endif()
```

Including from application

Application-specific MCUmgr handlers can be added by creating/editing application build files. Example modifications are shown below.

Example CMakeLists.txt The application `CMakeLists.txt` file can load the CMake file for the example MCUmgr handler by adding the following:

```
add_subdirectory(mcumgr/grp/<grp_name>)
```

Example Kconfig The application Kconfig file can include the Kconfig file for the example MCUmgr handler by adding the following to the `Kconfig` file in the application directory (or creating it if it does not exist):

```
rsource "mcumgr/grp/<grp_name>/Kconfig"

# Include Zephyr's Kconfig
source "Kconfig.zephyr"
```

Including from Zephyr Module

Zephyr *Modules (External projects)* can be used to add custom MCUmgr handlers to multiple different applications without needing to duplicate the code in each application's source tree, see *Module yaml file description* for details on how to set up the module files. Example files are shown below.

Example zephyr/module.yml This is an example file which can be used to load the Kconfig and CMake files from the root of the module directory, and would be placed at `zephyr/module.yml`:

```
build:
  kconfig: Kconfig
  cmake: .
```

Example CMakeLists.txt This is an example CMakeLists.txt file which loads the CMake file for the example MCUmgr handler, and would be placed at `CMakeLists.txt`:

```
add_subdirectory(mcumgr/grp/<grp_name>)
```

Example Kconfig This is an example Kconfig file which loads the Kconfig file for the example MCUmgr handler, and would be placed at `Kconfig`:

```
rsource "mcumgr/grp/<grp_name>/Kconfig"
```

Demonstration handler

There is a demonstration project which includes configuration for both application and zephyr module-MCUmgr handlers which can be used as a basis for created your own in tests/subsys/mgmt/mcumgr/handler_demo/.

4.5.3 MCUmgr Callbacks

Overview

MCUmgr has a customisable callback/notification system that allows application (and module) code to receive callbacks for MCUmgr events that they are interested in and react to them or return a status code to the calling function that provides control over if the action should be allowed or not. An example of this is with the fs_mgmt group, whereby file access can be gated, the callback allows the application to inspect the request path and allow or deny access to said file, or it can rewrite the provided path to a different path for transparent file redirection support.

Implementation

Enabling The base callback/notification system can be enabled using `CONFIG_MCUMGR_MGMT_NOTIFICATION_HOOKS` which will compile the registration and notification system into the code. This will not provide any callbacks by default as the callbacks that are supported by a build must also be selected by enabling the Kconfig's for the required callbacks (see *Events* for further details). A callback function with the *mgmt_cb* type definition can then be declared and registered by calling *mgmt_callback_register()* for the desired event inside of a :c:struct‘mgmt_callback‘ structure. Handlers are called in the order that they were registered.

With the system enabled, a basic handler can be set up and defined in application code as per:

```
#include <zephyr/kernel.h>
#include <zephyr/mgmt/mcumgr/mgmt/mgmt.h>
#include <zephyr/mgmt/mcumgr/mgmt/callbacks.h>

struct mgmt_callback my_callback;

enum mgmt_cb_return my_function(uint32_t event, enum mgmt_cb_return prev_status,
                                int32_t *rc, uint16_t *group, bool *abort_more,
                                void *data, size_t data_size)
{
    if (event == MGMT_EVT_OP_CMD_DONE) {
        /* This is the event we registered for */
    }

    /* Return OK status code to continue with acceptance to underlying handler */
```

(continues on next page)

(continued from previous page)

```
    return MGMT_CB_OK;
}

int main()
{
    my_callback.callback = my_function;
    my_callback.event_id = MGMT_EVT_OP_CMD_DONE;
    mgmt_callback_register(&my_callback);
}
```

This code registers a handler for the *MGMT_EVT_OP_CMD_DONE* event, which will be called after a MCUmgr command has been processed and output generated, note that this requires that CONFIG_MCUMGR_SMP_COMMAND_STATUS_HOOKS be enabled to receive this callback.

Multiple callbacks can be setup to use a single function as a common callback, and many different functions can be used for each event by registering each group once, or all notifications for a whole group can be enabled by using one of the MGMT_EVT_OP_*_ALL events, alternatively a handler can setup for every notification by using *MGMT_EVT_OP_ALL*. When setting up handlers, events can be combined that are in the same group only, for example 5 img_mgmt callbacks can be setup with a single registration call, but to also setup a callback for an os_mgmt callback, this must be done as a separate registration. Group IDs are numerical increments, event IDs are bitmask values, hence the restriction.

As an example, the following registration is allowed, which will register for 3 SMP events with a single callback function in a single registration:

```
my_callback.callback = my_function;
my_callback.event_id = (MGMT_EVT_OP_CMD_RECV |
                        MGMT_EVT_OP_CMD_STATUS |
                        MGMT_EVT_OP_CMD_DONE);
mgmt_callback_register(&my_callback);
```

The following code is not allowed, and will cause undefined operation, because it mixes the IMG management group with the OS management group whereby the group is **not** a bitmask value, only the event is:

```
my_callback.callback = my_function;
my_callback.event_id = (MGMT_EVT_OP_IMG_MGMT_DFU_STARTED |
                        MGMT_EVT_OP_OS_MGMT_RESET);
mgmt_callback_register(&my_callback);
```

Events Events can be selected by enabling their corresponding Kconfig option:

- **CONFIG_MCUMGR_SMP_COMMAND_STATUS_HOOKS**
  MCUmgr command status (*MGMT_EVT_OP_CMD_RECV*, *MGMT_EVT_OP_CMD_STATUS*, *MGMT_EVT_OP_CMD_DONE*)
- **CONFIG_MCUMGR_GRP_FS_FILE_ACCESS_HOOK**
  fs_mgmt file access (*MGMT_EVT_OP_FS_MGMT_FILE_ACCESS*)
- **CONFIG_MCUMGR_GRP_IMG_UPLOAD_CHECK_HOOK**
  img_mgmt upload check (*MGMT_EVT_OP_IMG_MGMT_DFU_CHUNK*)
- **CONFIG_MCUMGR_GRP_IMG_STATUS_HOOKS**
  img_mgmt upload status (*MGMT_EVT_OP_IMG_MGMT_DFU_STOPPED*, *MGMT_EVT_OP_IMG_MGMT_DFU_STARTED*, *MGMT_EVT_OP_IMG_MGMT_DFU_PENDING*, *MGMT_EVT_OP_IMG_MGMT_DFU_CONFIRMED*)
- **CONFIG_MCUMGR_GRP_OS_RESET_HOOK**
  os_mgmt reset check (*MGMT_EVT_OP_OS_MGMT_RESET*)

- CONFIG_MCUMGR_GRP_SETTINGS_ACCESS_HOOK
  settings_mgmt access (*MGMT_EVT_OP_SETTINGS_MGMT_ACCESS*)

Actions Some callbacks expect a return status to either allow or disallow an operation, an example is the fs_mgmt access hook which allows for access to files to be allowed or denied. With these handlers, the first non-OK error code returned by a handler will be returned to the MCUmgr client.

An example of selectively denying file access:

```
#include <zephyr/kernel.h>
#include <zephyr/mgmt/mcumgr/mgmt/mgmt.h>
#include <zephyr/mgmt/mcumgr/mgmt/callbacks.h>
#include <string.h>

struct mgmt_callback my_callback;

enum mgmt_cb_return my_function(uint32_t event, enum mgmt_cb_return prev_status,
                                int32_t *rc, uint16_t *group, bool *abort_more,
                                void *data, size_t data_size)
{
    /* Only run this handler if a previous handler has not failed */
    if (event == MGMT_EVT_OP_FS_MGMT_FILE_ACCESS && prev_status == MGMT_CB_OK) {
        struct fs_mgmt_file_access *fs_data = (struct fs_mgmt_file_access *)data;

        /* Check if this is an upload and deny access if it is, otherwise check
         * the path and deny if is matches a name
         */
        if (fs_data->access == FS_MGMT_FILE_ACCESS_WRITE) {
            /* Return an access denied error code to the client and abort calling
             * further handlers
             */
            *abort_more = true;
            *rc = MGMT_ERR_EACCESSDENIED;

            return MGMT_CB_ERROR_RC;
        } else if (strcmp(fs_data->filename, "/lfs1/false_deny.txt") == 0) {
            /* Return a no entry error code to the client, call additional handlers
             * (which will have failed set to true)
             */
            *rc = MGMT_ERR_ENOENT;

            return MGMT_CB_ERROR_RC;
        }
    }

    /* Return OK status code to continue with acceptance to underlying handler */
    return MGMT_CB_OK;
}

int main()
{
    my_callback.callback = my_function;
    my_callback.event_id = MGMT_EVT_OP_FS_MGMT_FILE_ACCESS;
    mgmt_callback_register(&my_callback);
}
```

This code registers a handler for the *MGMT_EVT_OP_FS_MGMT_FILE_ACCESS* event, which will be called after a fs_mgmt file read/write command has been received to check if access to the file should be allowed or not, note that this requires that CONFIG_MCUMGR_GRP_FS_FILE_ACCESS_HOOK be enabled to receive this callback. Two types of errors can be returned, the rc parameter can be set to an *mcumgr_err_t* error code and *MGMT_CB_ERROR_RC* can be returned, or a group error

code (introduced with version 2 of the MCUmgr protocol) can be set by setting the `group` value to the group and `rc` value to the group error code and returning *MGMT_CB_ERROR_ERR*.

MCUmgr Command Callback Usage/Adding New Event Types To add a callback to a MCUmgr command, *mgmt_callback_notify()* can be called with the event ID and, optionally, a data struct to pass to the callback (which can be modified by handlers). If no data needs to be passed back, `NULL` can be used instead, and size of the data set to 0.

An example MCUmgr command handler:

```
#include <zephyr/kernel.h>
#include <zcbor_common.h>
#include <zcbor_encode.h>
#include <zephyr/mgmt/mcumgr/smp/smp.h>
#include <zephyr/mgmt/mcumgr/mgmt/mgmt.h>
#include <zephyr/mgmt/mcumgr/mgmt/callbacks.h>

#define MGMT_EVT_GRP_USER_ONE MGMT_EVT_GRP_USER_CUSTOM_START

enum user_one_group_events {
    /** Callback on first post, data is test_struct. */
    MGMT_EVT_OP_USER_ONE_FIRST  = MGMT_DEF_EVT_OP_ID(MGMT_EVT_GRP_USER_ONE, 0),

    /** Callback on second post, data is test_struct. */
    MGMT_EVT_OP_USER_ONE_SECOND = MGMT_DEF_EVT_OP_ID(MGMT_EVT_GRP_USER_ONE, 1),

    /** Used to enable all user_one events. */
    MGMT_EVT_OP_USER_ONE_ALL    = MGMT_DEF_EVT_OP_ALL(MGMT_EVT_GRP_USER_ONE),
};

struct test_struct {
    uint8_t some_value;
};

static int test_command(struct mgmt_ctxt *ctxt)
{
    int rc;
    int err_rc;
    uint16_t err_group;
    zcbor_state_t *zse = ctxt->cnbe->zs;
    bool ok;
    struct test_struct test_data = {
        .some_value = 8,
    };

    rc = mgmt_callback_notify(MGMT_EVT_OP_USER_ONE_FIRST, &test_data,
                              sizeof(test_data), &err_rc, &err_group);

    if (rc != MGMT_CB_OK) {
        /* A handler returned a failure code */
        if (rc == MGMT_CB_ERROR_RC) {
            /* The failure code is the RC value */
            return err_rc;
        }

        /* The failure is a group and ID error value */
        ok = smp_add_cmd_err(zse, err_group, (uint16_t)err_rc);
        goto end;
    }

    /* All handlers returned success codes */
```

(continues on next page)

(continued from previous page)

```
    ok = zcbor_tstr_put_lit(zse, "output_value") &&
         zcbor_int32_put(zse, 1234);

end:
    rc = (ok ? MGMT_ERR_EOK : MGMT_ERR_EMSGSIZE);

    return rc;
}
```

If no response is required for the callback, the function call be called and casted to void.

Migration

If there is existing code using the previous callback system(s) in Zephyr 3.2 or earlier, then it will need to be migrated to the new system. To migrate code, the following callback registration functions will need to be migrated to register for callbacks using *mgmt_callback_register()* (note that CONFIG_MCUMGR_MGMT_NOTIFICATION_HOOKS will need to be set to enable the new notification system in addition to any migrations):

- **mgmt_evt**
  Using *MGMT_EVT_OP_CMD_RECV*, *MGMT_EVT_OP_CMD_STATUS*, or *MGMT_EVT_OP_CMD_DONE* as drop-in replacements for events of the same name, where the provided data is *mgmt_evt_op_cmd_arg*. CONFIG_MCUMGR_SMP_COMMAND_STATUS_HOOKS needs to be set.
- **fs_mgmt_register_evt_cb**
  Using *MGMT_EVT_OP_FS_MGMT_FILE_ACCESS* where the provided data is fs_mgmt_file_access. Instead of returning true to allow the action or false to deny, a MCUmgr result code needs to be returned, *MGMT_ERR_EOK* will allow the action, any other return code will disallow it and return that code to the client (*MGMT_ERR_EACCESSDENIED* can be used for an access denied error). CONFIG_MCUMGR_GRP_IMG_STATUS_HOOKS needs to be set.
- **img_mgmt_register_callbacks**
  Using *MGMT_EVT_OP_IMG_MGMT_DFU_STARTED* if dfu_started_cb was used, *MGMT_EVT_OP_IMG_MGMT_DFU_STOPPED* if dfu_stopped_cb was used, *MGMT_EVT_OP_IMG_MGMT_DFU_PENDING* if dfu_pending_cb was used or *MGMT_EVT_OP_IMG_MGMT_DFU_CONFIRMED* if dfu_confirmed_cb was used. These callbacks do not have any return status. CONFIG_MCUMGR_GRP_IMG_STATUS_HOOKS needs to be set.
- **img_mgmt_set_upload_cb**
  Using *MGMT_EVT_OP_IMG_MGMT_DFU_CHUNK* where the provided data is img_mgmt_upload_check. Instead of returning true to allow the action or false to deny, a MCUmgr result code needs to be returned, *MGMT_ERR_EOK* will allow the action, any other return code will disallow it and return that code to the client (*MGMT_ERR_EACCESSDENIED* can be used for an access denied error). CONFIG_MCUMGR_GRP_IMG_UPLOAD_CHECK_HOOK needs to be set.
- **os_mgmt_register_reset_evt_cb**
  Using *MGMT_EVT_OP_OS_MGMT_RESET*. Instead of returning true to allow the action or false to deny, a MCUmgr result code needs to be returned, *MGMT_ERR_EOK* will allow the action, any other return code will disallow it and return that code to the client (*MGMT_ERR_EACCESSDENIED* can be used for an access denied error). CONFIG_MCUMGR_SMP_COMMAND_STATUS_HOOKS needs to be set.

API Reference

group `mcumgr_callback_api`

MCUmgr callback API.

Defines

`MGMT_EVT_GET_GROUP`(event)

Get group from event.

`MGMT_EVT_GET_ID`(event)

Get event ID from event.

`MGMT_CB_ERROR_RET`

Typedefs

typedef enum *mgmt_cb_return* (***mgmt_cb**)(uint32_t event, enum *mgmt_cb_return* prev_status, int32_t *rc, uint16_t *group, bool *abort_more, void *data, size_t data_size)

Function to be called on MGMT notification/event.

This callback function is used to notify an application or system about a MCUmgr mgmt event.

Param event
mcumgr_op_t.

Param prev_status
mgmt_cb_return of the previous handler calls, if it is an error then it will be the first error that was returned by a handler (i.e. this handler is being called for a notification only, the return code will be ignored).

Param rc
If `prev_status` is *MGMT_CB_ERROR_RC* then this is the SMP error that was returned by the first handler that failed. If `prev_status` is *MGMT_CB_ERROR_ERR* then this will be the group error rc code returned by the first handler that failed. If the handler wishes to raise an SMP error, this must be set to the *mcumgr_err_t* status and *MGMT_CB_ERROR_RC* must be returned by the function, if the handler wishes to raise a ret error, this must be set to the group ret status and *MGMT_CB_ERROR_ERR* must be returned by the function.

Param group
If `prev_status` is *MGMT_CB_ERROR_ERR* then this is the group of the ret error that was returned by the first handler that failed. If the handler wishes to raise a ret error, this must be set to the group ret status and *MGMT_CB_ERROR_ERR* must be returned by the function.

Param abort_more
Set to true to abort further processing by additional handlers.

Param data
Optional event argument.

Param data_size
Size of optional event argument (0 if no data is provided).

Return
mgmt_cb_return indicating the status to return to the calling code (only checked when this is the first failure reported by a handler).

Enums

enum **mgmt_cb_return**

MGMT event callback return value.

Values:

enumerator **MGMT_CB_OK**

No error.

enumerator **MGMT_CB_ERROR_RC**

SMP protocol error and `err_rc` contains the *mcumgr_err_t* error code.

enumerator **MGMT_CB_ERROR_ERR**

Group (application-level) error and `err_group` contains the group ID that caused the error and `err_rc` contains the error code of that group to return.

enum **mgmt_cb_groups**

MGMT event callback group IDs.

Note that this is not a 1:1 mapping with *mcumgr_group_t* values.

Values:

enumerator **MGMT_EVT_GRP_ALL** = 0

enumerator **MGMT_EVT_GRP_SMP**

enumerator **MGMT_EVT_GRP_OS**

enumerator **MGMT_EVT_GRP_IMG**

enumerator **MGMT_EVT_GRP_FS**

enumerator **MGMT_EVT_GRP_SETTINGS**

enumerator **MGMT_EVT_GRP_USER_CUSTOM_START** = *MGMT_GROUP_ID_PERUSER*

enum **smp_all_events**

MGMT event opcodes for all command processing.

Values:

enumerator **MGMT_EVT_OP_ALL** = MGMT_DEF_EVT_OP_ALL(*MGMT_EVT_GRP_ALL*)

Used to enable all events.

enum **smp_group_events**

MGMT event opcodes for base SMP command processing.

Values:

enumerator **MGMT_EVT_OP_CMD_RECV** = MGMT_DEF_EVT_OP_ID(*MGMT_EVT_GRP_SMP*, 0)

Callback when a command is received, data is *mgmt_evt_op_cmd_arg()*.

enumerator **MGMT_EVT_OP_CMD_STATUS** = MGMT_DEF_EVT_OP_ID(*MGMT_EVT_GRP_SMP*, 1)

Callback when a status is updated, data is *mgmt_evt_op_cmd_arg()*.

enumerator **MGMT_EVT_OP_CMD_DONE** = MGMT_DEF_EVT_OP_ID(*MGMT_EVT_GRP_SMP*, 2)

Callback when a command has been processed, data is *mgmt_evt_op_cmd_arg()*.

enumerator **MGMT_EVT_OP_CMD_ALL** = MGMT_DEF_EVT_OP_ALL(*MGMT_EVT_GRP_SMP*)

Used to enable all smp_group events.

enum **fs_mgmt_group_events**

MGMT event opcodes for filesystem management group.

Values:

enumerator **MGMT_EVT_OP_FS_MGMT_FILE_ACCESS** = MGMT_DEF_EVT_OP_ID(*MGMT_EVT_GRP_FS*, 0)

Callback when a file has been accessed, data is fs_mgmt_file_access().

enumerator **MGMT_EVT_OP_FS_MGMT_ALL** = MGMT_DEF_EVT_OP_ALL(*MGMT_EVT_GRP_FS*)

Used to enable all fs_mgmt_group events.

enum **img_mgmt_group_events**

MGMT event opcodes for image management group.

Values:

enumerator **MGMT_EVT_OP_IMG_MGMT_DFU_CHUNK** = MGMT_DEF_EVT_OP_ID(*MGMT_EVT_GRP_IMG*, 0)

Callback when a client sends a file upload chunk, data is img_mgmt_upload_check().

enumerator **MGMT_EVT_OP_IMG_MGMT_DFU_STOPPED** = MGMT_DEF_EVT_OP_ID(*MGMT_EVT_GRP_IMG*, 1)

Callback when a DFU operation is stopped.

enumerator **MGMT_EVT_OP_IMG_MGMT_DFU_STARTED** = MGMT_DEF_EVT_OP_ID(*MGMT_EVT_GRP_IMG*, 2)

Callback when a DFU operation is started.

enumerator **MGMT_EVT_OP_IMG_MGMT_DFU_PENDING** = MGMT_DEF_EVT_OP_ID(*MGMT_EVT_GRP_IMG*, 3)

Callback when a DFU operation has finished being transferred.

enumerator **MGMT_EVT_OP_IMG_MGMT_DFU_CONFIRMED** = MGMT_DEF_EVT_OP_ID(*MGMT_EVT_GRP_IMG*, 4)

Callback when an image has been confirmed.

enumerator **MGMT_EVT_OP_IMG_MGMT_DFU_CHUNK_WRITE_COMPLETE** = MGMT_DEF_EVT_OP_ID(*MGMT_EVT_GRP_IMG*, 5)

Callback when an image write command has finished writing to flash.

enumerator **MGMT_EVT_OP_IMG_MGMT_ALL** = MGMT_DEF_EVT_OP_ALL(*MGMT_EVT_GRP_IMG*)

Used to enable all img_mgmt_group events.

enum **os_mgmt_group_events**

MGMT event opcodes for operating system management group.

Values:

enumerator **MGMT_EVT_OP_OS_MGMT_RESET** = MGMT_DEF_EVT_OP_ID(*MGMT_EVT_GRP_OS*, 0)

Callback when a reset command has been received, data is os_mgmt_reset_data.

enumerator **MGMT_EVT_OP_OS_MGMT_INFO_CHECK** = MGMT_DEF_EVT_OP_ID(*MGMT_EVT_GRP_OS*, 1)

Callback when an info command is processed, data is os_mgmt_info_check.

enumerator **MGMT_EVT_OP_OS_MGMT_INFO_APPEND** = MGMT_DEF_EVT_OP_ID(*MGMT_EVT_GRP_OS*, 2)

Callback when an info command needs to output data, data is os_mgmt_info_append.

enumerator **MGMT_EVT_OP_OS_MGMT_DATETIME_GET** = MGMT_DEF_EVT_OP_ID(*MGMT_EVT_GRP_OS*, 3)

Callback when a datetime get command has been received.

enumerator **MGMT_EVT_OP_OS_MGMT_DATETIME_SET** = MGMT_DEF_EVT_OP_ID(*MGMT_EVT_GRP_OS*, 4)

Callback when a datetime set command has been received, data is struct *rtc_time()*.

enumerator **MGMT_EVT_OP_OS_MGMT_ALL** = MGMT_DEF_EVT_OP_ALL(*MGMT_EVT_GRP_OS*)

Used to enable all os_mgmt_group events.

enum **settings_mgmt_group_events**

MGMT event opcodes for settings management group.

Values:

enumerator **MGMT_EVT_OP_SETTINGS_MGMT_ACCESS** = MGMT_DEF_EVT_OP_ID(*MGMT_EVT_GRP_SETTINGS*, 0)

Callback when a setting is read/written/deleted.

enumerator **MGMT_EVT_OP_SETTINGS_MGMT_ALL** = MGMT_DEF_EVT_OP_ALL(*MGMT_EVT_GRP_SETTINGS*)

Used to enable all settings_mgmt_group events.

Functions

uint8_t **mgmt_evt_get_index**(uint32_t event)

Get event ID index from event.

Parameters

- **event** – Event to get ID index from.

Returns

Event index.

enum *mgmt_cb_return* **mgmt_callback_notify**(uint32_t event, void *data, size_t data_size, int32_t *err_rc, uint16_t *err_group)

This function is called to notify registered callbacks about mcumgr notifications/events.

Parameters

- **event** – *mcumgr_op_t*.
- **data** – Optional event argument.
- **data_size** – Size of optional event argument (0 if none).
- **err_rc** – Pointer to rc value.
- **err_group** – Pointer to group value.

Returns

mgmt_cb_return either *MGMT_CB_OK* if all handlers returned it, or *MGMT_CB_ERROR_RC* if the first failed handler returned an SMP error (in which case `err_rc` will be updated with the SMP error) or *MGMT_CB_ERROR_ERR* if the first failed handler returned a ret group and error (in which case `err_group` will be updated with the failed group ID and `err_rc` will be updated with the group-specific error code).

void **mgmt_callback_register**(struct *mgmt_callback* *callback)

Register event callback function.

Parameters

- **callback** – Callback struct.

void **mgmt_callback_unregister**(struct *mgmt_callback* *callback)

Unregister event callback function.

Parameters

- **callback** – Callback struct.

struct **mgmt_callback**

#include <callbacks.h> MGMT callback struct.

Public Members

sys_snode_t **node**

Entry list node.

mgmt_cb **callback**

Callback that will be called.

uint32_t **event_id**

MGMT_EVT_[...] Event ID for handler to be called on.

This has special meaning if *MGMT_EVT_OP_ALL* is used (which will cover all events for all groups), or MGMT_EVT_OP_*_MGMT_ALL (which will cover all events for a single group). For events that are part of a single group, they can be or'd together for this to have one registration trigger on multiple events, please note that this will only work for a single group, to register for events in different groups, they must be registered separately.

struct **mgmt_evt_op_cmd_arg**

#include <callbacks.h> Arguments for *MGMT_EVT_OP_CMD_RECV*, *MGMT_EVT_OP_CMD_STATUS* and *MGMT_EVT_OP_CMD_DONE*.

Public Members

uint16_t **group**

mcumgr_group_t

uint8_t **id**

Message ID within group.

uint8_t **op**

mcumgr_op_t used in *MGMT_EVT_OP_CMD_RECV*

int **err**

mcumgr_err_t, used in *MGMT_EVT_OP_CMD_DONE*

int **status**

img_mgmt_id_upload_t, used in *MGMT_EVT_OP_CMD_STATUS*

4.5.4 Fixing and backporting fixes to Zephyr v2.7 MCUmgr

The processes described in this document apply to both the zephyr repository itself and the MCUmgr *module* defined in west.yml.

Note

Currently, the backporting process, described in this document, is required only when providing changes to Zephyr version 2.7 LTS

There are two different processes: one for issues that have also been fixed in the current version of Zephyr (backports), and one for issues that are being fixed only in a previous version.

The upstream MCUmgr repository is located in this page. The Zephyr fork used in version 2.7 and earlier is located here. Versions of Zephyr past 2.7 use the MCUmgr library that is part of the Zephyr code base.

Possible origins of a code change

In Zephyr version 2.7 and earlier, you must first apply the fix to the upstream repository of MCUmgr and then bring it to Zephyr with snapshot updates.

As such, there are four possible ways to apply a change to the 2.7 branch:

- The fix, done directly to the Zephyr held code of the MCUmgr library, is backported to the `v2.7-branch`.
- The fix, ported to the Zephyr held code from the upstream repository, is backported to the `v2.7-branch`.
- **The fix, done upstream and no longer relevant to the current version, is directly backported**
  to the `v2.7-branch`.
- **The fix, not present upstream and not relevant for the current version of Zephyr, is**
  directly applied to the `v2.7-branch`.

The first three cases are cases of *backports* , the last one is a case of a *new fix* and has no corresponding fix in the current version.

Applying fixes to previous versions of MCUmgr

This section indicates how to apply fixes to previous versions of MCUmgr.

Creating a bug report Every proposed fix requires a bug report submitted for the specified version of Zephyr affected by the bug.

In case the reported bug in a previous version has already been fixed in the current version, the description of the bug must be copied with the following:

- Additional references to the bug in the current version
- The PR for the current version
- The SHAs of the commits, if the PR has already been merged

You must also apply the `backport v2.7-branch` label to the bug report.

Creating the pull request for the fix You can either create a *backport pull request* or a *new-fix pull request*.

Creating backport pull requests Backporting a fix means that some or all of the fix commits, as they exist in the current version, are ported to a previous version.

> **Note**
>
> Backporting requires the fix for the current version to be already merged.

To create a backport pull request, do the following:

1. Port the fix commits from the current version to the previous version. Even if some of the commits require changes, keep the commit messages of all the ported commits as close to the ones in the original commits as possible, adding the following line:

```
"Backporting commit <sha>"

``<sha>`` indicates the SHA of the commit after it has been already merged in the current␣
↪version.
```

1. Create the pull request selecting `v2.7-branch` as the merge target.
2. Update `west.yml` within Zephyr, creating a pull-request to update the MCUmgr library referenced in Zephyr 2.7.

Creating new-fix pull requests When the fix needed does not have a corresponding fix in the current version, the bug report must follow the ordinary process.

1. Create the pull request selecting `v2.7-branch` as the merge target.
2. Update `west.yml` within Zephyr, creating a pull-request to update the MCUmgr library referenced in Zephyr 2.7.

Configuration management

This chapter describes the maintainers' side of accepting and merging fixes and backports.

Prerequisites As a maintainer, these are the steps required before proceeding with the merge process:

1. Check if the author has followed the correct steps that are required to apply the fix, as described in *Applying fixes to previous versions of MCUmgr*.

1. Ensure that the author of the fix has also provided the `west.yml` update for Zephyr 2.7.

The specific merging process depends on where the fix comes from and whether it is a *backport* or a *new fix*.

Merging a backported fix There are two possible sources of backports:

- The Zephyr code base
- A direct fix from upstream

Both cases are similar and differ only in the branch name.

To merge a backported fix after the pull request for the fix has gone through the review process, as a maintainer, do the following:

1. Create a branch named as follow:

   ```
   backport-<source>-<pr_num>-to_v2.7-branch
   ```

   `<source>` can be one of the following:

   - `upstream` - if the fix has originally been merged to the upstream repository.
   - `zephyr` - if the fix has been applied to the Zephyr internal MCUmgr library (past 2.7 versions).

   `<pr_num>` is the number of the original pull request that has already been merged.

   For example, a branch named `backport-upstream-137-to-v2.7-branch` indicates a backport of pull request 137, which has already been merged to the upstream repository of MCUmgr.

2. Push the reviewed pull-request branch to the newly created branch and merge the backport branch to `v2.7-branch`.

Merging a new fix Merging a new fix, that is not a backport of either any upstream or Zephyr fix, does not require any special treatment. Apply the fix directly at the top of `v2.7-branch`.

Merge west.yml As an MCUmgr maintainer, you may not be able to merge the `west.yml` update, to introduce the fix to Zephyr. However, you are responsible for such a merge to happen as soon as possible after the MCUmgr fixes have been applied to the `v2.7-branch` of the MCUmgr.

4.5.5 SMP Protocol Specification

This is description of Simple Management Protocol, SMP, that is used by MCUmgr to pass requests to devices and receive responses from them.

SMP is an application layer protocol. The underlying transport layer is not in scope of this documentation.

> **Note**
>
> SMP in this context refers to SMP for MCUmgr (Simple Management Protocol), it is unrelated to SMP in Bluetooth (Security Manager Protocol), but there is an MCUmgr SMP transport for Bluetooth.

Frame: The envelope

Each frame consists of a header and data. The `Data Length` field in the header may be used for reassembly purposes if underlying transport layer supports fragmentation. Frames are encoded in “Big Endian” (Network endianness) when fields are more than one byte long, and takes the following form:

3								2								1								0							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Res				Ver		OP		Flags								Data Length															
Group ID																Sequence Num								Command ID							
Data ...																															

> **Note**
>
> The original specification states that SMP should support receiving both the “Little-endian” and “Big-endian” frames but in reality the MCUmgr library is hardcoded to always treat “Network” side as “Big-endian”.

Data is optional and is not present when `Data Length` is zero. The encoding of data depends on the target of group/ID.

A description of the various fields and their meaning:

Field	Description
Res	This is reserved, not-used field and must be always set to 0.
Ver (Version)	This indicates the version of the protocol being used, this should be set to 0b01 to use the newer SMP transport where error codes are more detailed and returned in the map, otherwise left as 0b00 to use the legacy SMP protocol. Versions 0b10 and 0b11 are reserved for future use and should not be used.
OP	*mcumgr_op_t*, determines whether information is written to a device or requested from it and whether a packet contains request to an SMP server or response from it.
Flags	Reserved for flags; there are no flags defined yet, the field should be set to 0
Data Length	Length of the Data field
Group ID	*mcumgr_group_t*, see *Management Group ID's* for further details.
Sequence Num	This is a frame sequence number. The number is increased by one with each request frame. The Sequence Num of a response should match the one in the request.
Command ID	This is a command, within Group.
Data	This is data payload of the Data Length size. It is optional as Data Length may be set to zero, which means that no data follows the header.

Note

Contents of Data depends on a value of an OP, a Group ID, and a Command ID.

Management Group ID's The SMP protocol supports predefined common groups and allows user defined groups. The following table presents a list of common groups:

Decimal ID	Group description
0	*Default/OS Management Group*
1	*Application/software image management group*
2	*Statistics management*
3	*Settings (Config) Management Group*
4	Application/system log management (currently not used by Zephyr)
5	Run-time tests (unused by Zephyr)
6	Split image management (unused by Zephyr)
7	Test crashing application (unused by Zephyr)
8	*File management*
9	*Shell management*
63	*Zephyr Management Group*
64	This is the base group for defining an application specific management groups.

The payload for above groups, except for user groups (64 and above) is always CBOR encoded. The group 64, and above can define their own scheme for data communication.

Minimal response

Regardless of a command issued, as long as there is SMP client on the other side of a request, a response should be issued containing the header followed by CBOR map container. Lack of response is only allowed when there is no SMP service or device is non-responsive.

Minimal response SMP data Minimal response is:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

"err" -> "group"	*mcumgr_group_t* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	*mcumgr_err_t* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Note that in the case of a successful command, an empty map will be returned (`rc`/`err` is only returned if there is an error condition, therefore if only an empty map is returned or a response lacks these, the request can be considered as being successful. For SMP version 2, errors relating to SMP itself that are not group specific will still be returned as `rc` errors, SMP version 2 clients must therefore be able to handle both types of errors.

Specifications of management groups supported by Zephyr

Default/OS Management Group OS management group defines following commands:

Command ID	Command description
0	Echo
1	Console/Terminal echo control; unimplemented by Zephyr
2	Task Statistics
3	Memory pool statistics
4	Date-time string
5	System reset
6	MCUMGR parameters
7	OS/Application info
8	Bootloader information

Echo command Echo command responses by sending back string that it has received.

Echo request Echo request header fields:

OP	Group ID	Command ID
0 or 2	0	0

CBOR data of request:

```
{
    (str)"d" : (str)
}
```

where:

“d”	string to be replied by echo service.

Echo response Echo response header fields:

OP	Group ID	Command ID	Note
1	0	0	When request OP was 0
3	0	0	When request OP was 2

CBOR data of successful response:

```
{
    (str)"r"        : (str)
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

“r”	replying echo string.
“err” -> “group”	*mcumgr_group_t* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
“err” -> “rc”	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
“rc”	*mcumgr_err_t* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Task statistics command The command responds with some system statistics.

Task statistics request Task statistics request header fields:

OP	Group ID	Command ID
0	0	2

The command sends an empty CBOR map as data.

Task statistics response Task statistics response header fields:

OP	Group ID	Command ID
1	0	2

CBOR data of successful response:

```
{
    (str)"tasks" : {
        (str)<task_name> : {
            (str)"prio"         : (uint)
            (str)"tid"          : (uint)
            (str)"state"        : (uint)
            (str)"stkuse"       : (uint)
            (str)"stksiz"       : (uint)
            (str)"cswcnt"       : (uint)
            (str)"runtime"      : (uint)
            (str)"last_checkin" : (uint)
            (str)"next_checkin" : (uint)
        }
        ...
    }
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

<task_nam	string identifying task.
“prio”	task priority.
“tid”	numeric task ID.
“state”	numeric task state.
“stkuse”	task’s/thread’s stack usage.
“stksiz”	task’s/thread’s stack size.
“cswcnt”	task’s/thread’s context switches.
“run-time”	task’s/thread’s runtime in “ticks”.
“last_check	set to 0 by Zephyr.
“next_chec	set to 0 by Zephyr.
“err” -> “group”	*mcumgr_group_t* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
“err” -> “rc”	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
“rc”	*mcumgr_err_t* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Note

The unit for “stkuse” and “stksiz” is system dependent and in case of Zephyr this is number of 4 byte words.

Memory pool statistics The command is used to obtain information on memory pools active in running system.

Memory pool statistic request Memory pool statistics request header fields:

OP	Group ID	Command ID
0	0	3

The command sends an empty CBOR map as data.

Memory pool statistics response Memory pool statistics response header fields:

OP	Group ID	Command ID
1	0	3

CBOR data of successful response:

```
{
    (str)<pool_name> {
        (str)"blksiz"   : (int)
        (str)"nblks"    : (int)
        (str)"nfree"    : (int)
        (str)"min'      : (int)
    }
    ...
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

<pool_nam	string representing the pool name, used as a key for dictionary with pool statistics data.
“blksiz”	size of the memory block in the pool.
“nblks”	number of blocks in the pool.
“nfree”	number of free blocks.
“min”	lowest number of free blocks the pool reached during run-time.
“err” -> “group”	*mcumgr_group_t* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
“err” -> “rc”	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
“rc”	*mcumgr_err_t* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Date-time command The command allows to obtain string representing current time-date on a device or set a new time to a device. The time format used, by both set and get operations, is:

“yyyy-MM-dd’T’HH:mm:ss.SSSSSSZZZZZ”

Date-time get The command allows to obtain date-time from a device.

Date-time get request Date-time request header fields:

OP	Group ID	Command ID
0	0	4

The command sends an empty CBOR map as data.

Date-time get response Date-time get response header fields:

OP	Group ID	Command ID
1	0	4

CBOR data of successful response:

```
{
    (str)"datetime" : (str)
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

"date-time"	String in format: `yyyy-MM-dd'T'HH:mm:ss.SSSSSSZZZZZ`.
"err" -> "group"	*mcumgr_group_t* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	*mcumgr_err_t* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Date-time set The command allows to set date-time to a device.

Date-time set request Date-time set request header fields:

OP	Group ID	Command ID
2	0	4

CBOR data of response:

```
{
    (str)"datetime" : (str)
}
```

where:

"datetime"	String in format: `yyyy-MM-dd'T'HH:mm:ss.SSSSSSZZZZZ`.

Date-time set response Date-time set response header fields:

OP	Group ID	Command ID
3	0	4

The command sends an empty CBOR map as data if successful. In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

"err" -> "group"	*mcumgr_group_t* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	*mcumgr_err_t* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

System reset Performs reset of system. The device should issue response before resetting so that the SMP client could receive information that the command has been accepted. By default, this command is accepted in all conditions, however if the CONFIG_MCUMGR_GRP_OS_RESET_HOOK is enabled and an application registers a callback, the callback will be called when this command is issued and can be used to perform any necessary tidy operations prior to the module rebooting, or to reject the reset request outright altogether with an error response. For details on this functionality, see *ref:'mcumgr_callbacks*.

System reset request System reset request header fields:

OP	Group ID	Command ID
2	0	5

Normally the command sends an empty CBOR map as data, but if a previous reset attempt has responded with "rc" equal to *MGMT_ERR_EBUSY* then the following map may be sent to force a reset:

```
{
    (opt)"force"       : (int)
}
```

where:

"force"	Force reset if value > 0, optional if 0.

System reset response System reset response header fields

OP	Group ID	Command ID
3	0	5

The command sends an empty CBOR map as data if successful. In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

"err" -> "group"	*mcumgr_group_t* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	*mcumgr_err_t* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

MCUmgr Parameters Used to obtain parameters of mcumgr library.

MCUmgr Parameters Request MCUmgr parameters request header fields:

OP	Group ID	Command ID
0	0	6

The command sends an empty CBOR map as data.

MCUmgr Parameters Response MCUmgr parameters response header fields

OP	Group ID	Command ID
1	0	6

CBOR data of successful response:

```
{
    (str)"buf_size"     : (uint)
    (str)"buf_count"    : (uint)
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

"buf_size"	Single SMP buffer size, this includes SMP header and CBOR payload.
"buf_count	Number of SMP buffers supported.
"err" -> "group"	*mcumgr_group_t* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	*mcumgr_err_t* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

OS/Application Info Used to obtain information on running image, similar functionality to the linux uname command, allowing details such as kernel name, kernel version, build date/time, processor type and application-defined details to be returned. This functionality can be enabled with `CONFIG_MCUMGR_GRP_OS_INFO`.

OS/Application Info Request OS/Application info request header fields:

OP	Group ID	Command ID
0	0	7

CBOR data of request:

```
{
    (str,opt)"format"      : (str)
}
```

where:

"format'	Format specifier of returned response, fields are appended in their natural ascending index order, not the order of characters that are received by the command. Format specifiers: * `s` Kernel name * `n` Node name * `r` Kernel release * `v` Kernel version * `b` Build date and time (requires `CONFIG_MCUMGR_GRP_OS_INFO_BUILD_DATE_TIME`) * `m` Machine * `p` Processor * `i` Hardware platform * `o` Operating system * `a` All fields (shorthand for all above options) If this option is not provided, the `s` Kernel name option will be used.

OS/Application Info Response OS/Application info response header fields

OP	Group ID	Command ID
1	0	7

CBOR data of successful response:

```
{
    (str)"output"       : (str)
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

"output"	Text response including requested parameters.
"err" -> "group"	*mcumgr_group_t* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	*mcumgr_err_t* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Bootloader Information Allows retrieving information about the on-board bootloader and its parameters.

Bootloader Information Request Bootloader information request header:

OP	Group ID	Command ID
0	0	8

CBOR data of request:

```
{
    (str,opt)"query"  : (str)
}
```

where:

“que	Is string representing query for parameters, with no restrictions how the query looks like as processing of query is left for bootloader backend. If there is no query, then response will return string identifying the bootloader.

Bootloader Information Response Bootloader information response header:

OP	Group ID	Command ID
1	0	8

In case when no “query” has been provided in request, CBOR data of response:

```
{
    (str)"bootloader"      : (str)
}
```

where:

“bootloader”	String representing bootloader name

In case when “query” is provided:

```
{
    (str,opt)<response>   : ()
    ...
}
```

where:

<re-spor	Response to “query”. This is optional and may be left out in case when query yields no response, SMP version 2 error code of *OS_MGMT_ERR_QUERY_YIELDS_NO_ANSWER* is expected. Response may have more than one parameter reported back or it may be a map, that is dependent on bootloader backednd and query.
...	Parameter characteristic information.

Parameter may be accompanied by additional, parameter specific, information keywords with assigned values.

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

"err" -> "group"	*mcumgr_group_t* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	*mcumgr_err_t* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Bootloader Information: MCUboot In case when MCUboot is application bootloader, empty request will be responded with:

```
{
    (str)"bootloader"       : (str)"MCUboot"
}
```

Currently "MCUboot" supports querying for mode of operation:

```
{
    (str)"query"            : (str)"mode"
}
```

Response to "mode" is:

```
{
    (str)"mode"                     : (int)
    (str,opt)"no-downgrade"         : (bool)
}
```

where "mode" is one of:

-1	Unknown mode of MCUboot.
0	MCUboot is in single application mode.
1	MCUboot is in swap using scratch partition mode.
2	MCUboot is in overwrite (upgrade-only) mode.
3	MCUboot is in swap without scratch mode.
4	MCUboot is in DirectXIP without revert mode.
5	MCUboot is in DirectXIP with revert mode.
6	MCUboot is in RAM loader mode.

The `no-downgrade` field is a flag, which is always sent when true, indicating that MCUboot has downgrade prevention enabled; downgrade prevention means that if the uploaded image has a lower version than the currently running application, it will not be used for an update by MCUboot.

MCUmgr may reject images with a lower version in this configuration.

Application/software image management group Application/software image management group defines following commands:

Command ID	Command description
0	State of images
1	Image upload
2	File (reserved but not supported by Zephyr)
3	Corelist (reserved but not supported by Zephyr)
4	Coreload (reserved but not supported by Zephyr)
5	Image erase

Notion of "slots" and "images" in Zephyr The "slot" and "image" definition comes from mcuboot where "image" would consist of two "slots", further named "primary" and "secondary"; the application is supposed to run from the "primary slot" and update is supposed to be uploaded to the "secondary slot"; the mcuboot is responsible in swapping slots on boot. This means that pair of slots is dedicated to single upgradable application. In case of Zephyr this gets a little bit confusing because DTS will use "slot0_partition" and "slot1_partition", as label of `fixed-partition` dedicated to single application, but will name them as "image-0" and "image-1" respectively.

Currently Zephyr supports at most two images, in which case mapping is as follows:

Image	Slot labels	Slot Names
1	"slot0_partition" "slot1_partition"	"image-0" "image-1"
2	"slot2_partition" "slot3_partition"	"image-2" "image-3"

State of images The command is used to set state of images and obtain list of images with their current state.

Get state of images request Get state of images request header fields:

OP	Group ID	Command ID
0	1	0

The command sends an empty CBOR map as data.

Get state of images response Get state of images response header fields:

OP	Group ID	Command ID
1	1	0

Note

Below definition of the response contains "image" field that has been marked as optional(opt): the field may not appear in response when target application does not support more than one image. The field is mandatory when application supports more than one application image to allow identifying which image information is listed.

A response will only contain information for valid images, if an image can not be identified as valid it is simply skipped.

CBOR data of successful response:

```
{
    (str)"images" : [
        {
            (str,opt)"image"        : (uint)
            (str)"slot"             : (uint)
            (str)"version"          : (str)
            (str,opt*)"hash"        : (byte str)
            (str,opt)"bootable"     : (bool)
            (str,opt)"pending"      : (bool)
            (str,opt)"confirmed"    : (bool)
            (str,opt)"active"       : (bool)
            (str,opt)"permanent"    : (bool)
        }
        ...
    ]
    (str,opt)"splitStatus" : (int)
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
    (str,opt)"rsn"  : (str)
}
```

where:

“image”	semi-optional image number; the field is not required when only one image is supported by the running application.
“slot”	slot number within “image”; each image has two slots : primary (running one) = 0 and secondary (for DFU dual-bank purposes) = 1.
“version”	string representing image version, as set with `imgtool`.
“hash”	SHA256 hash of the image header and body. Note that this will not be the same as the SHA256 of the whole file, it is the field in the MCUboot TLV section that contains a hash of the data which is used for signature verification purposes. This field is optional but only optional when using MCUboot’s serial recovery feature with one pair of image slots, Kconfig `CONFIG_BOOT_SERIAL_IMG_GRP_HASH` can be disabled to remove support for hashes in this configuration. MCUmgr in applications must support sending hashes. **Note** See `IMAGE_TLV_SHA256` in the MCUboot image format documentation link below.
“bootable”	true if image has bootable flag set; this field does not have to be present if false.
“pending”	true if image is set for next swap; this field does not have to be present if false.
“confirmed”	true if image has been confirmed; this field does not have to be present if false.
“active”	true if image is currently active application; this field does not have to be present if false.
“permanent”	true if image is to stay in primary slot after the next boot; this does not have to be present if false.
“splitStatus”	states whether loader of split image is compatible with application part; this is unused by Zephyr.
“err” -> “group”	*`mcumgr_group_t`* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
“err” -> “rc”	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
“rc”	*`mcumgr_err_t`* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.
“rsn”	optional string that clarifies reason for an error; specifically useful when `rc` is *`MGMT_ERR_EUNKNOWN`*.

Note

For more information on how does image/slots function, please refer to the MCUBoot documentation https://docs.mcuboot.com/design.html#image-slots For information on MCUboot image format, please reset to the MCUboot documentation https://docs.mcuboot.com/design.html#image-format

Set state of image request Set state of image request header fields:

OP	Group ID	Command ID
2	1	0

CBOR data of request:

```
{
    (str,opt)"hash"     : (str)
    (str)"confirm"      : (bool)
}
```

If "confirm" is false or not provided, an image with the "hash" will be set for test, which means that it will not be marked as permanent and upon hard reset the previous application will be restored to the primary slot. In case when "confirm" is true, the "hash" is optional as the currently running application will be assumed as target for confirmation.

Set state of image response The response takes the same format as *Get state of images response*

Image upload The image upload command allows to update application image.

Image upload request The image upload request is sent for each chunk of image that is uploaded, until complete image gets uploaded to a device.

Image upload request header fields:

OP	Group ID	Command ID
2	1	1

CBOR data of request:

```
{
    (str,opt)"image"    : (uint)
    (str,opt)"len"      : (uint)
    (str)"off"          : (uint)
    (str,opt)"sha"      : (byte str)
    (str)"data"         : (byte str)
    (str,opt)"upgrade"  : (bool)
}
```

where:

“image”	optional image number, it does not have to appear in request at all, in which case it is assumed to be 0. Should only be present when “off” is 0.
“len”	optional length of an image. Must appear when “off” is 0.
“off”	offset of image chunk the request carries.
“sha”	SHA256 hash of an upload; this is used to identify an upload session (e.g. to allow MCUmgr to continue a broken session), and for image verification purposes. This must be a full SHA256 hash of the whole image being uploaded, or not included if the hash is not available (in which case, upload session continuation and image verification functionality will be unavailable). Should only be present when “off” is 0.
“data”	image data to write at provided offset.
“upgrade”	optional flag that states that only upgrade should be allowed, so if the version of uploaded software is not higher then already on a device, the image upload will be rejected. Zephyr compares major, minor and revision (x.y.z) by default unless `CONFIG_MCUMGR_GRP_IMG_VERSION_CMP_USE_BUILD_NUMBER` is set, whereby it will compare build numbers too. Should only be present when “off” is 0.

Note

There is no field representing size of chunk that is carried as “data” because that information is embedded within “data” field itself.

Note

It is possible that a server will respond to an upload with “off” of 0, this may happen if an upload on another transport (or outside of MCUmgr entirely) is started, if the device has rebooted or if a packet has been lost. If this happens, a client must re-send all the required and optional fields that it sent in the original first packet so that the upload state can be re-created by the server. If the original fields are not included, the upload will be unable to continue.

The MCUmgr library uses “sha” field to tag ongoing update session, to be able to continue it in case when it gets broken, and for upload verification purposes. If library gets request with “off” equal zero it checks stored “sha” within its state and if it matches it will respond to update client application with offset that it should continue with. If this hash is not available (e.g. because a file is being streamed) then it must not be provided, image verification and upload session continuation features will be unavailable in this case.

Image upload response Image upload response header fields:

OP	Group ID	Command ID
3	1	1

CBOR data of successful response:

```
{
    (str,opt)"off"    : (uint)
    (str,opt)"match"  : (bool)
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
    (str,opt)"rsn"  : (str)
}
```

where:

"off"	offset of last successfully written byte of update.
"match"	indicates if the uploaded data successfully matches the provided SHA256 hash or not, only sent in the final packet if `CONFIG_IMG_ENABLE_IMAGE_CHECK` is enabled.
"err" -> "group"	*`mcumgr_group_t`* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	*`mcumgr_err_t`* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.
"rsn"	optional string that clarifies reason for an error; specifically useful when `rc` is *`MGMT_ERR_EUNKNOWN`*.

The "off" field is only included in responses to successfully processed requests; if "rc" is negative then "off" may not appear.

Image erase The command is used for erasing image slot on a target device.

> **Note**
>
> This is synchronous command which means that a sender of request will not receive response until the command completes, which can take a long time.

Image erase request Image erase request header fields:

OP	Group ID	Command ID
2	1	5

CBOR data of request:

```
{
    (str,opt)"slot"     : (uint)
}
```

where:

"slot"	optional slot number, it does not have to appear in the request at all, in which case it is assumed to be 1.

Image erase response Image erase response header fields:

OP	Group ID	Command ID
3	1	5

The command sends an empty CBOR map as data if successful. In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
    (str,opt)"rsn"  : (str)
}
```

where:

"err" -> "group"	*mcumgr_group_t* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	*mcumgr_err_t* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.
"rsn"	optional string that clarifies reason for an error; specifically useful when `rc` is *MGMT_ERR_EUNKNOWN*.

Note

Response from Zephyr running device may have "rc" value of *MGMT_ERR_EBADSTATE*, which means that the secondary image has been marked for next boot already and may not be erased.

Statistics management Statistics management allows to obtain data gathered by Statistics subsystem of Zephyr, enabled with `CONFIG_STATS`.

Statistics management group defines commands:

Command ID	Command description
0	Group data
1	List groups

Statistics: group data The command is used to obtain data for group specified by a name. The name is one of group names as registered, with `STATS_INIT_AND_REG` macro or `stats_init_and_reg()` function call, within module that gathers the statistics.

Statistics: group data request Statistics group data request header:

OP	Group ID	Command ID
0	2	0

CBOR data of request:

```
{
    (str)"name" :  (str)
}
```

where:

“name”	group name.

Statistics: group data response Statistics group data response header:

OP	Group ID	Command ID
1	2	0

CBOR data of successful response:

```
{
    (str)"name"     : (str)
    (str)"fields"   : {
        (str)<entry_name> : (uint)
        ...
    }
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

“name”	this is name of group the response contains data for.
“fields”	this is map of entries within groups that consists of pairs where the entry name is mapped to value it represents in statistics.
<entry_name>	single entry to value mapping; value is hardcoded to unsigned integer type, in a CBOR meaning.
“err” -> “group”	*mcumgr_group_t* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
“err” -> “rc”	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
“rc”	*mcumgr_err_t* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Statistics: list of groups The command is used to obtain list of groups of statistics that are gathered on a device. This is a list of names as given to groups with `STATS_INIT_AND_REG` macro or `stats_init_and_reg()` function calls, within module that gathers the statistics; this means that this command may be considered optional as it is known during compilation what groups will be included into build and listing them is not needed prior to issuing a query.

Statistics: list of groups request Statistics group list request header:

OP	Group ID	Command ID
0	2	1

The command sends an empty CBOR map as data.

Statistics: list of groups response Statistics group list request header:

OP	Group ID	Command ID
1	2	1

CBOR data of successful response:

```
{
    (str)"stat_list" :  [
        (str)<stat_group_name>, ...
    ]
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

“stat_list”	array of strings representing group names; this array may be empty if there are no groups.
“err” -> “group”	*mcumgr_group_t* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
“err” -> “rc”	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
“rc”	*mcumgr_err_t* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Settings (Config) Management Group Settings management group (known as Configuration Manager in the original MCUmgr repository) defines the following commands:

Command ID	Command description
0	Read/write setting
1	Delete setting
2	Commit settings
3	Load/Save settings

Note that the Zephyr version adds additional commands and features which are not supported by the original upstream version, however, the original client functionality should work for read/write functionality.

Read/write setting command Read/write setting command allows updating a setting entry on a device or getting the current value of a setting from a device.

Read setting request Read setting request header fields:

OP	Group ID	Command ID
0	3	0

CBOR data of request:

```
{
    (str)"name"         : (str)
    (str,opt)"max_size" : (uint)
}
```

where:

“name”	string of the setting to retrieve
“max_size”	optional maximum size of data to return

Read setting response Read setting response header fields:

OP	Group ID	Command ID
1	3	0

CBOR data of successful response:

```
{
    (str)"val"          : (bstr)
    (str,opt)"max_size" : (uint)
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1

```
{
    (str)"rc"       : (int)
}
```

where:

“val”	binary string of the returned data, note that the underlying data type cannot be specified through this and must be known by the client.
“max_s	will be set if the maximum supported data size is smaller than the maximum requested data size, and contains the maximum data size which the device supports, equivalent to kconfig:option:*CONFIG_MCUMGR_GRP_SETTINGS_NAME_LEN*.
“err” -> “group	*mcumgr_group_t* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
“err” -> “rc”	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
“rc”	*mcumgr_err_t* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Write setting request Write setting request header fields:

OP	Group ID	Command ID
2	3	0

CBOR data of request:

```
{
    (str)"name"    : (str)
    (str)"val"     : (bstr)
}
```

where:

“name”	string of the setting to update/set
“val”	value to set the setting to

Write setting response Write setting response header fields:

OP	Group ID	Command ID
3	3	0

The command sends an empty CBOR map as data if successful. In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1

```
{
    (str)"rc"       : (int)
}
```

where:

“err” -> “group”	*mcumgr_group_t* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
“err” -> “rc”	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
“rc”	*mcumgr_err_t* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Delete setting command Delete setting command allows deleting a setting on a device.

Delete setting request Delete setting request header fields:

OP	Group ID	Command ID
2	3	1

CBOR data of request:

```
{
    (str)"name"   : (str)
}
```

where:

“name”	string of the setting to delete

Delete setting response Delete setting response header fields:

OP	Group ID	Command ID
3	3	1

The command sends an empty CBOR map as data if successful. In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1

```
{
    (str)"rc"       : (int)
}
```

where:

“err” -> “group”	*mcumgr_group_t* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
“err” -> “rc”	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
“rc”	*mcumgr_err_t* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Commit settings command Commit settings command allows committing all settings that have been set but not yet applied on a device.

Commit settings request Commit settings request header fields:

OP	Group ID	Command ID
2	3	2

The command sends an empty CBOR map as data.

Commit settings response Commit settings response header fields:

OP	Group ID	Command ID
3	3	2

The command sends an empty CBOR map as data if successful. In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1

```
{
    (str)"rc"       : (int)
}
```

where:

"err" -> "group"	*mcumgr_group_t* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	*mcumgr_err_t* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Load/Save settings command Load/Save settings command allows loading/saving all serialized items from/to persistent storage on a device.

Load settings request Load settings request header fields:

OP	Group ID	Command ID
0	3	3

The command sends an empty CBOR map as data.

Load settings response Load settings response header fields:

OP	Group ID	Command ID
1	3	3

The command sends an empty CBOR map as data if successful. In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1

```
{
    (str)"rc"       : (int)
}
```

where:

"err" -> "group"	*mcumgr_group_t* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	*mcumgr_err_t* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Save settings request Save settings request header fields:

OP	Group ID	Command ID
2	3	3

The command sends an empty CBOR map as data.

Save settings response Save settings response header fields:

OP	Group ID	Command ID
3	3	3

The command sends an empty CBOR map as data if successful. In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1

```
{
    (str)"rc"       : (int)
}
```

where:

"err" -> "group"	*mcumgr_group_t* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	*mcumgr_err_t* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Settings access callback There is a settings access MCUmgr callback available (see *MCUmgr Callbacks* for details on callbacks) which allows for applications/modules to know when settings management commands are used and, optionally, block access (for example through the use of a security mechanism). This callback can be enabled with `CONFIG_MCUMGR_GRP_SETTINGS_ACCESS_HOOK`, registered with the event *MGMT_EVT_OP_SETTINGS_MGMT_ACCESS*, whereby the supplied callback data is `settings_mgmt_access`.

File management The file management group provides commands that allow to upload and download files to/from a device.

File management group defines following commands:

Command ID	Command description
0	File download/upload
1	File status
2	File hash/checksum
3	Supported file hash/checksum types
4	File close

File download Command allows to download contents of an existing file from specified path of a target device. Client applications must keep track of data they have already downloaded and where their position in the file is (MCUmgr will cache these also), and issue subsequent requests, with modified offset, to gather the entire file. Request does not carry size of requested chunk, the size is specified by application itself. Note that file handles will remain open for consecutive requests (as long as an idle timeout has not been reached and another transport does not make use of uploading/downloading files using fs_mgmt), but files are not exclusively owned by MCUmgr, for the time of download session, and may change between requests or even be removed.

Note

By default, all file upload/download requests are unconditionally allowed. However, if the Kconfig option `CONFIG_MCUMGR_GRP_FS_FILE_ACCESS_HOOK` is enabled, then an application can register a callback handler for *MGMT_EVT_OP_FS_MGMT_FILE_ACCESS* (see *MCUmgr callbacks*), which allows for allowing or declining access to reading/writing a particular file, or for rewriting the path supplied by the client.

File download request File download request header:

OP	Group ID	Command ID
0	8	0

CBOR data of request:

```
{
    (str)"off" :  (uint)
    (str)"name" : (str)
}
```

where:

“off”	offset to start download at
“name”	absolute path to a file

File download response File download response header:

OP	Group ID	Command ID
1	8	0

CBOR data of successful response:

```
{
    (str)"off"      : (uint)
    (str)"data"     : (byte str)
    (str,opt)"len"  : (uint)
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

“off”	offset the response is for.
“data”	chunk of data read from file; it is CBOR encoded stream of bytes with embedded size; “data” appears only in responses where “rc” is 0.
“len”	length of file, this field is only mandatory when “off” is 0.
“err” -> “group”	*mcumgr_group_t* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
“err” -> “rc”	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
“rc”	*mcumgr_err_t* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

File upload Allows to upload a file to a specified location. Command will automatically overwrite existing file or create a new one if it does not exist at specified path. The protocol supports stateless upload where each requests carries different chunk of a file and it is client side responsibility to track progress of upload.

Note that file handles will remain open for consecutive requests (as long as an idle timeout has not been reached, but files are not exclusively owned by MCUmgr, for the time of download session, and may change between requests or even be removed. Note that file handles will remain open for consecutive requests (as long as an idle timeout has not been reached and another

transport does not make use of uploading/downloading files using fs_mgmt), but files are not exclusively owned by MCUmgr, for the time of download session, and may change between requests or even be removed.

Note

Weirdly, the current Zephyr implementation is half-stateless as is able to hold single upload context, holding information on ongoing upload, that consists of bool flag indicating in-progress upload, last successfully uploaded offset and total length only.

Note

By default, all file upload/download requests are unconditionally allowed. However, if the Kconfig option `CONFIG_MCUMGR_GRP_FS_FILE_ACCESS_HOOK` is enabled, then an application can register a callback handler for *`MGMT_EVT_OP_FS_MGMT_FILE_ACCESS`* (see *MCUmgr callbacks*), which allows for allowing or declining access to reading/writing a particular file, or for rewriting the path supplied by the client.

File upload request File upload request header:

OP	Group ID	Command ID
2	8	0

CBOR data of request:

```
{
    (str)"off"      : (uint)
    (str)"data"     : (str)
    (str)"name"     : (str)
    (str,opt)"len"  : (uint)
}
```

where:

“off”	offset to start/continue upload at.
“data”	chunk of data to write to the file; it is CBOR encoded with length embedded.
“name”	absolute path to a file.
“len”	length of file, this field is only mandatory when “off” is 0.

File upload response File upload response header:

OP	Group ID	Command ID
3	8	0

CBOR data of successful response:

```
{
    (str)"off"      : (uint)
}
```

In case of error the CBOR data takes the form:

where:

“off”	offset of last successfully written data.
“err” -> “group”	*mcumgr_group_t* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
“err” -> “rc”	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
“rc”	*mcumgr_err_t* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

File status Command allows to retrieve status of an existing file from specified path of a target device.

File status request File status request header:

OP	Group ID	Command ID
0	8	1

CBOR data of request:

```
{
    (str)"name" : (str)
}
```

where:

“name”	absolute path to a file.

File status response File status response header:

OP	Group ID	Command ID
1	8	1

CBOR data of successful response:

```
{
    (str)"len"      : (uint)
}
```

In case of error the CBOR data takes form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

“len”		length of file (in bytes).
“err” “group”	->	*mcumgr_group_t* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
“err” “rc”	->	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
“rc”		*mcumgr_err_t* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

File hash/checksum Command allows to generate a hash/checksum of an existing file at a specified path on a target device. Note that kernel heap memory is required for buffers to be allocated for this to function, and large stack memory buffers are required for generation of the output hash/checksum. Requires `CONFIG_MCUMGR_GRP_FS_CHECKSUM_HASH` to be enabled for the base functionality, supported hash/checksum are opt-in with `CONFIG_MCUMGR_GRP_FS_CHECKSUM_IEEE_CRC32` or `CONFIG_MCUMGR_GRP_FS_HASH_SHA256`.

File hash/checksum request File hash/checksum request header:

OP	Group ID	Command ID
0	8	2

CBOR data of request:

```
{
    (str)"name"     : (str)
    (str,opt)"type" : (str)
    (str,opt)"off"  : (uint)
    (str,opt)"len"  : (uint)
}
```

where:

“name”	absolute path to a file.
“type”	type of hash/checksum to perform *Hash/checksum types* or omit to use default.
“off”	offset to start hash/checksum calculation at (optional, 0 if not provided).
“len”	maximum length of data to read from file to generate hash/checksum with (optional, full file size if not provided).

Hash/checksum types

String name	Hash/checksum	Byte string	Size (bytes)
crc32	IEEE CRC32 checksum	No	4
sha256	SHA256 (Secure Hash Algorithm)	Yes	32

Note that the default type will be crc32 if it is enabled, or sha256 if crc32 is not enabled.

File hash/checksum response File hash/checksum response header:

OP	Group ID	Command ID
1	8	2

CBOR data of successful response:

```
{
    (str)"type"     : (str)
    (str,opt)"off"  : (uint)
    (str)"len"      : (uint)
    (str)"output"   : (uint or bstr)
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

"type"	type of hash/checksum that was performed *Hash/checksum types*.
"off"	offset that hash/checksum calculation started at (only present if not 0).
"len"	length of input data used for hash/checksum generation (in bytes).
"output"	output hash/checksum.
"err" -> "group"	*mcumgr_group_t* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	*mcumgr_err_t* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Supported file hash/checksum types Command allows listing which hash and checksum types are available on a device. Requires Kconfig `CONFIG_MCUMGR_GRP_FS_CHECKSUM_HASH_SUPPORTED_CMD` to be enabled.

Supported file hash/checksum types request Supported file hash/checksum types request header:

OP	Group ID	Command ID
0	8	3

The command sends empty CBOR map as data.

Supported file hash/checksum types response Supported file hash/checksum types response header:

OP	Group ID	Command ID
1	8	3

CBOR data of successful response:

```
{
    (str)"types" : {
        (str)<hash_checksum_name> : {
            (str)"format"       : (uint)
            (str)"size"         : (uint)
        }
        ...
    }
}
```

In case of error the CBOR data takes form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

<hash_checksur	name of the hash/checksum type *Hash/checksum types*.
"format"	format that the hash/checksum returns where 0 is for numerical and 1 is for byte array.
"size"	size (in bytes) of output hash/checksum response.
"err" -> "group"	*mcumgr_group_t* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	*mcumgr_err_t* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

File close Command allows closing any open file handles held by fs_mgmt upload/download requests that might have stalled or be incomplete.

File close request File close request header:

OP	Group ID	Command ID
2	8	4

The command sends empty CBOR map as data.

File close response File close response header:

OP	Group ID	Command ID
3	8	4

The command sends an empty CBOR map as data if successful. In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

"err"	->	*mcumgr_group_t* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err"	->	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"		*mcumgr_err_t* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Shell management Shell management allows passing commands to the shell subsystem over the SMP protocol.

Shell management group defines following commands:

Command ID	Command description
0	Shell command line execute

Shell command line execute The command allows to execute command line in a similar way to typing it into a shell, but both a request and a response are transported over SMP.

Shell command line execute request Execute command request header:

OP	Group ID	Command ID
2	9	0

CBOR data of request:

```
{
    (str)"argv"     : [
        (str)<cmd>
        (str,opt)<arg>
        ...
    ]
}
```

where:

“argv”	array consisting of strings representing command and its arguments.
<cmd>	command to be executed.
<arg>	optional arguments to command.

Shell command line execute response Command line execute response header fields:

OP	Group ID	Command ID
3	9	0

CBOR data of successful response:

```
{
    (str)"o"            : (str)
    (str)"ret"          : (int)
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

“o”		command output.
“ret”		return code from shell command execution.
“err” “group”	->	*mcumgr_group_t* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
“err” “rc”	->	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
“rc”		*mcumgr_err_t* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Note

In older versions of Zephyr, "rc" was used for both the mcumgr status code and shell command execution return code, this legacy behaviour can be restored by enabling `CONFIG_MCUMGR_GRP_SHELL_LEGACY_RC_RETURN_CODE`

4.5.6 SMP Transport Specification

The documents specifies information needed for implementing server and client side SMP transports.

BLE (Bluetooth Low Energy)

MCUmgr Clients need to use following BLE Characteristics, when implementing SMP client:

- **Service UUID**: *8D53DC1D-1DB7-4CD3-868B-8A527460AA84*
- **Characteristic UUID**: *DA2E7828-FBCE-4E01-AE9E-261174997C48*

All SMP communication utilizes a single GATT characteristic. An SMP request is sent via a GATT Write Without Response command. An SMP response is sent in the form of a GATT Notification

If an SMP request or response is too large to fit in a single GATT command, the sender fragments it across several packets. No additional framing is introduced when a request or response is fragmented; the payload is simply split among several packets. Since GATT guarantees ordered delivery of packets, the SMP header in the first fragment contains sufficient information for reassembly.

UART/serial and console

SMP protocol specification by MCUmgr subsystem of Zephyr uses basic framing of data to allow multiplexing of UART channel. Multiplexing requires prefixing each frame with two byte marker and terminating it with newline. Currently MCUmgr imposes a 127 byte limit on frame size, although there are no real protocol constraints that require that limit. The limit includes the prefix and the newline character, so the allowed payload size is actually 124 bytes.

Although no such transport exists in Zephyr, it is possible to implement MCUmgr client/server over UART transport that does not have framing at all, or uses hardware serial port control, or other means of framing.

Frame fragmenting SMP protocol over serial is fragmented into MTU size frames; each frame consists of two byte start marker, body and terminating newline character.

There are four types of types of frames: initial, partial, partial-final and initial-final; each frame type differs by start marker and/or body contents.

Frame formats Initial frame requires to be followed by optional sequence of partial frames and finally by partial-final frame. Body is always Base64 encoded, so the body size, here described as MTU - 3, is able to actually carry N = (MTU - 3) / 4 * 3 bytes of raw data.

Body of initial frame is preceded by two byte total packet length, encoded in Big Endian, and equals size of a raw body plus two bytes, size of CRC16; this means that actual body size allowed into an initial frame is N - 2.

If a body size is smaller than N - 4, than it is possible to carry entire body with preceding length and following it CRC in a single frame, here called initial-final; for the description of initial-final frame look below.

Initial frame format:

Content	Size	Description
0x06 0x09	2 bytes	Frame start marker
<base64-i>	no more than MTU - 3 bytes	Base64 encoded body
0x0a	1 byte	Frame termination

<base64-i> is Base64 encoded body of format:

Content	Size	Description
total length	2 bytes	Big endian 16-bit value representing total length of body + 2 bytes for CRC16; note that size of total length field is not added to total length value.
body	no more than MTU - 5	Raw body data fragment

Initial-final frame format is similar to initial frame format, but differs by <base64-i> definition.

<base64-i> of initial-final frame, is Base64 encoded data taking form:

Content	Size	Description
total length	2 bytes	Big endian 16-bit value representing total length of body + 2 bytes for CRC16; note that size of total length field is not added to total length value.
body	no more than MTU - 7	Raw body data fragment
crc16	2 bytes	CRC16 of entire packet body, preceding length not included.

Partial frame is continuation after previous initial or other partial frame. Partial frame takes form:

Content	Size	Description
0x04 0x14	2 bytes	Frame start marker
<base64-i>	no more than MTU - 3 bytes	Base64 encoded body
0x0a	1 byte	Frame termination

The <base64-i> of partial frame is Base64 encoding of data, taking form:

Content	Size	Description
body	no more than MTU - 3	Raw body data fragment

The <base64-i> of partial-final frame is Base64 encoding of data, taking form:

Content	Size	Description
body	no more than MTU - 3	Raw body data fragment
crc16	2 bytes	CRC16 of entire packet body, preceding length not included.

CRC Details The CRC16 included in final type frames is calculated over only raw data and does not include packet length. CRC16 polynomial is 0x1021 and initial value is 0.

API Reference

group `mcumgr_transport_smp`

MCUmgr transport SMP API.

Typedefs

typedef int (*`smp_transport_out_fn`)(struct *net_buf* *nb)

SMP transmit callback for transport.

The supplied *net_buf* is always consumed, regardless of return code.

Param nb
The *net_buf* to transmit.

Return
0 on success, *mcumgr_err_t* code on failure.

typedef uint16_t (*`smp_transport_get_mtu_fn`)(const struct *net_buf* *nb)

SMP MTU query callback for transport.

The supplied *net_buf* should contain a request received from the peer whose MTU is being queried. This function takes a *net_buf* parameter because some transports store connection-specific information in the *net_buf* user header (e.g., the BLE transport stores the peer address).

Param nb
Contains a request from the relevant peer.

Return
The transport's MTU; 0 if transmission is currently not possible.

typedef int (*`smp_transport_ud_copy_fn`)(struct *net_buf* *dst, const struct *net_buf* *src)

SMP copy user_data callback.

The supplied src *net_buf* should contain a user_data that cannot be copied using regular memcpy function (e.g., the BLE transport *net_buf* user_data stores the connection reference that has to be incremented when is going to be used by another buffer).

Param dst
Source buffer user_data pointer.

Param src
Destination buffer user_data pointer.

Return
0 on success, *mcumgr_err_t* code on failure.

typedef void (*smp_transport_ud_free_fn)(void *ud)

SMP free user_data callback.

This function frees *net_buf* user data, because some transports store connection-specific information in the *net_buf* user data (e.g., the BLE transport stores the connection reference that has to be decreased).

Param ud
Contains a user_data pointer to be freed.

typedef bool (*smp_transport_query_valid_check_fn)(struct *net_buf* *nb, void *arg)

Function for checking if queued data is still valid.

This function is used to check if queued SMP data is still valid e.g. on a remote device disconnecting, this is triggered when *smp_rx_remove_invalid()* is called.

Param nb
net buf containing queued request.

Param arg
Argument provided when calling *smp_rx_remove_invalid()* function.

Return
false if data is no longer valid/should be freed, true otherwise.

Enums

enum smp_transport_type

SMP transport type for client registration.

Values:

enumerator SMP_SERIAL_TRANSPORT = 0

SMP serial.

enumerator SMP_BLUETOOTH_TRANSPORT

SMP bluetooth.

enumerator SMP_SHELL_TRANSPORT

SMP shell.

enumerator SMP_UDP_IPV4_TRANSPORT

SMP UDP IPv4.

enumerator SMP_UDP_IPV6_TRANSPORT

SMP UDP IPv6.

enumerator SMP_USER_DEFINED_TRANSPORT

SMP user defined type.

Functions

int **smp_transport_init**(struct *smp_transport* *smpt)

Initializes a Zephyr SMP transport object.

Parameters

- **smpt** – The transport to construct.

Returns
0 If successful

Returns
Negative errno code if failure.

void **smp_rx_remove_invalid**(struct *smp_transport* *zst, void *arg)

Used to remove queued requests for an SMP transport that are no longer valid.

A *smp_transport_query_valid_check_fn()* function must be registered for this to function. If the *smp_transport_query_valid_check_fn()* function returns false during a callback, the queried command will classed as invalid and dropped.

Parameters

- **zst** – The transport to use.
- **arg** – Argument provided to callback *smp_transport_query_valid_check_fn()* function.

void **smp_rx_clear**(struct *smp_transport* *zst)

Used to clear pending queued requests for an SMP transport.

Parameters

- **zst** – The transport to use.

void **smp_client_transport_register**(struct *smp_client_transport_entry* *entry)

Register a Zephyr SMP transport object for client.

Parameters

- **entry** – The transport to construct.

struct *smp_transport* ***smp_client_transport_get**(int smpt_type)

Discover a registered SMP transport client object.

Parameters

- **smpt_type** – Type of transport

Returns
Pointer to registered object. Unknown type return NULL.

struct **smp_transport_api_t**

#include <smp.h> Function pointers of SMP transport functions, if a handler is NULL then it is not supported/implemented.

Public Members

smp_transport_out_fn **output**

Transport's send function.

smp_transport_get_mtu_fn **get_mtu**

Transport's get-MTU function.

smp_transport_ud_copy_fn **ud_copy**
Transport buffer user_data copy function.

smp_transport_ud_free_fn **ud_free**
Transport buffer user_data free function.

smp_transport_query_valid_check_fn **query_valid_check**
Transport's check function for if a query is valid.

struct **smp_transport**
#include <*smp.h*> SMP transport object for sending SMP responses.

struct **smp_client_transport_entry**
#include <*smp.h*> SMP Client transport structure.

Public Members

struct *smp_transport* ***smpt**
Transport structure pointer.

int **smpt_type**
Transport type.

4.5.7 Device Firmware Upgrade

Overview

The Device Firmware Upgrade subsystem provides the necessary frameworks to upgrade the image of a Zephyr-based application at run time. It currently consists of two different modules:

- subsys/dfu/boot/: Interface code to bootloaders
- subsys/dfu/img_util/: Image management code

The DFU subsystem deals with image management, but not with the transport or management protocols themselves required to send the image to the target device. For information on these protocols and frameworks please refer to the *Device Management* section.

Flash Image The flash image API as part of the Device Firmware Upgrade (DFU) subsystem provides an abstraction on top of Flash Stream to simplify writing firmware image chunks to flash.

API Reference

group **flash_img_api**
Abstraction layer to write firmware images to flash.

Functions

int **flash_img_init_id**(struct *flash_img_context* *ctx, uint8_t area_id)

Initialize context needed for writing the image to the flash.

Parameters

- **ctx** – context to be initialized
- **area_id** – flash area id of partition where the image should be written

Returns

0 on success, negative errno code on fail

int **flash_img_init**(struct *flash_img_context* *ctx)

Initialize context needed for writing the image to the flash.

Parameters

- **ctx** – context to be initialized

Returns

0 on success, negative errno code on fail

size_t **flash_img_bytes_written**(struct *flash_img_context* *ctx)

Read number of bytes of the image written to the flash.

Parameters

- **ctx** – context

Returns

Number of bytes written to the image flash.

int **flash_img_buffered_write**(struct *flash_img_context* *ctx, const uint8_t *data, size_t len, bool flush)

Process input buffers to be written to the image slot 1.

flash memory in single blocks. Will store remainder between calls.

A final call to this function with flush set to true will write out the remaining block buffer to flash. Since flash is written to in blocks, the contents of flash from the last byte written up to the next multiple of CONFIG_IMG_BLOCK_BUF_SIZE is padded with 0xff.

Parameters

- **ctx** – context
- **data** – data to write
- **len** – Number of bytes to write
- **flush** – when true this forces any buffered data to be written to flash

Returns

0 on success, negative errno code on fail

int **flash_img_check**(struct *flash_img_context* *ctx, const struct *flash_img_check* *fic, uint8_t area_id)

Verify flash memory length bytes integrity from a flash area.

The start point is indicated by an offset value.

The function is enabled via CONFIG_IMG_ENABLE_IMAGE_CHECK Kconfig options.

Parameters

- **ctx** – **[in]** context.

- `fic` – **[in]** flash img check data.
- `area_id` – **[in]** flash area id of partition where the image should be verified.

Returns
0 on success, negative errno code on fail

struct `flash_img_context`

#include <flash_img.h>

struct `flash_img_check`

#include <flash_img.h> Structure for verify flash region integrity.

Match vector length is fixed and depends on size from hash algorithm used to verify flash integrity. The current available algorithm is SHA-256.

Public Members

size_t `clen`

Match vector data.

MCUBoot API The MCUboot API is provided to get version information and boot status of application images. It allows to select application image and boot type for the next boot.

API Reference

group `mcuboot_api`

MCUboot public API for MCUboot control of image boot process.

Defines

`BOOT_SWAP_TYPE_NONE`

Attempt to boot the contents of slot 0.

`BOOT_SWAP_TYPE_TEST`

Swap to slot 1.

Absent a confirm command, revert back on next boot.

`BOOT_SWAP_TYPE_PERM`

Swap to slot 1, and permanently switch to booting its contents.

`BOOT_SWAP_TYPE_REVERT`

Swap back to alternate slot.

A confirm changes this state to NONE.

`BOOT_SWAP_TYPE_FAIL`

Swap failed because image to be run is not valid.

BOOT_IMG_VER_STRLEN_MAX

BOOT_UPGRADE_TEST

Boot upgrade request modes.

BOOT_UPGRADE_PERMANENT

Functions

int **boot_read_bank_header**(uint8_t area_id, struct *mcuboot_img_header* *header, size_t header_size)

Read the MCUboot image header information from an image bank.

This attempts to parse the image header, From the start of the *area_id* image.

Parameters

- **area_id** – *flash_area* ID of image bank which stores the image.
- **header** – On success, the returned header information is available in this structure.
- **header_size** – Size of the header structure passed by the caller. If this is not large enough to contain all of the necessary information, an error is returned.

Returns

Zero on success, a negative value on error.

bool **boot_is_img_confirmed**(void)

Check if the currently running image is confirmed as OK.

MCUboot can perform "test" upgrades. When these occur, a new firmware image is installed and booted, but the old version will be reverted at the next reset unless the new image explicitly marks itself OK.

This routine can be used to check if the currently running image has been marked as OK.

See also

boot_write_img_confirmed()

Returns

True if the image is confirmed as OK, false otherwise.

int **boot_write_img_confirmed**(void)

Marks the currently running image as confirmed.

This routine attempts to mark the currently running firmware image as OK, which will install it permanently, preventing MCUboot from reverting it for an older image at the next reset.

This routine is safe to call if the current image has already been confirmed. It will return a successful result in this case.

Returns

0 on success, negative errno code on fail.

int **boot_write_img_confirmed_multi**(int image_index)

Marks the image with the given index in the primary slot as confirmed.

This routine attempts to mark the firmware image in the primary slot as OK, which will install it permanently, preventing MCUboot from reverting it for an older image at the next reset.

This routine is safe to call if the current image has already been confirmed. It will return a successful result in this case.

Parameters

- **image_index** – Image pair index.

Returns
0 on success, negative errno code on fail.

int **mcuboot_swap_type**(void)

Determines the action, if any, that mcuboot will take on the next reboot.

Returns
a BOOT_SWAP_TYPE_[...] constant on success, negative errno code on fail.

int **mcuboot_swap_type_multi**(int image_index)

Determines the action, if any, that mcuboot will take on the next reboot.

Parameters

- **image_index** – Image pair index.

Returns
a BOOT_SWAP_TYPE_[...] constant on success, negative errno code on fail.

int **boot_request_upgrade**(int permanent)

Marks the image in slot 1 as pending.

On the next reboot, the system will perform a boot of the slot 1 image.

Parameters

- **permanent** – Whether the image should be used permanently or only tested once: BOOT_UPGRADE_TEST=run image once, then confirm or revert. BOOT_UPGRADE_PERMANENT=run image forever.

Returns
0 on success, negative errno code on fail.

int **boot_request_upgrade_multi**(int image_index, int permanent)

Marks the image with the given index in the secondary slot as pending.

On the next reboot, the system will perform a boot of the secondary slot image.

Parameters

- **image_index** – Image pair index.
- **permanent** – Whether the image should be used permanently or only tested once: BOOT_UPGRADE_TEST=run image once, then confirm or revert. BOOT_UPGRADE_PERMANENT=run image forever.

Returns
0 on success, negative errno code on fail.

int **boot_erase_img_bank**(uint8_t area_id)

Erase the image Bank.

Parameters

- **area_id** – *flash_area* ID of image bank to be erased.

Returns
0 on success, negative errno code on fail.

ssize_t **boot_get_area_trailer_status_offset**(uint8_t area_id)

Get the offset of the status in the image bank.

Parameters

- **area_id** – *flash_area* ID of image bank to get the status offset

Returns
a positive offset on success, negative errno code on fail

ssize_t **boot_get_trailer_status_offset**(size_t area_size)

Get the offset of the status from an image bank size.

Parameters

- **area_size** – size of image bank

Returns
offset of the status. When negative the status will not fit the given size

struct **mcuboot_img_sem_ver**

#include <*mcuboot.h*> MCUboot image header representation for image version.

The header for an MCUboot firmware image contains an embedded version number, in semantic versioning format. This structure represents the information it contains.

struct **mcuboot_img_header_v1**

#include <*mcuboot.h*> Model for the MCUboot image header as of version 1.

This represents the data present in the image header, in version 1 of the header format.

Some information present in the header but not currently relevant to applications is omitted.

Public Members

uint32_t **image_size**

The size of the image, in bytes.

struct *mcuboot_img_sem_ver* **sem_ver**

The image version.

struct **mcuboot_img_header**

#include <*mcuboot.h*> Model for the MCUBoot image header.

This contains the decoded image header, along with the major version of MCUboot that the header was built for.

(The MCUboot project guarantees that incompatible changes to the image header will result in major version changes to the bootloader itself, and will be detectable in the persistent representation of the header.)

Public Members

uint32_t **mcuboot_version**

The version of MCUboot the header is built for.

The value 1 corresponds to MCUboot versions 1.x.y.

struct *mcuboot_img_header_v1* **v1**

Header information for MCUboot version 1.

union *mcuboot_img_header* **h**

The header information.

It is only valid to access fields in the union member corresponding to the mcuboot_version field above.

Bootloaders

MCUboot Zephyr is directly compatible with the open source, cross-RTOS MCUboot boot loader. It interfaces with MCUboot and is aware of the image format required by it, so that Device Firmware Upgrade is available when MCUboot is the boot loader used with Zephyr. The source code itself is hosted in the MCUboot GitHub Project page.

In order to use MCUboot with Zephyr you need to take the following into account:

1. You will need to define the flash partitions required by MCUboot; see *Flash map* for details.
2. You will have to specify your flash partition as the chosen code partition

```
/ {
   chosen {
      zephyr,code-partition = &slot0_partition;
   };
};
```

3. Your application's `.conf` file needs to enable the `CONFIG_BOOTLOADER_MCUBOOT` Kconfig option in order for Zephyr to be built in an MCUboot-compatible manner
4. You need to build and flash MCUboot itself on your device
5. You might need to take precautions to avoid mass erasing the flash and also to flash the Zephyr application image at the correct offset (right after the bootloader)

More detailed information regarding the use of MCUboot with Zephyr can be found in the MCUboot with Zephyr documentation page on the MCUboot website.

4.5.8 Over-the-Air Update

Overview

Over-the-Air (OTA) Update is a method for delivering firmware updates to remote devices using a network connection. Although the name implies a wireless connection, updates received over a wired connection (such as Ethernet) are still commonly referred to as OTA updates. This approach requires server infrastructure to host the firmware binary and implement a method of signaling when an update is available. Security is a concern with OTA updates; firmware binaries should be cryptographically signed and verified before upgrading.

The *Device Firmware Upgrade* section discusses upgrading Zephyr firmware using MCUboot. The same method can be used as part of OTA. The binary is first downloaded into an unoccupied code partition, usually named `slot1_partition`, then upgraded using the *MCUboot* process.

Examples of OTA

Golioth Golioth is an IoT management platform that includes OTA updates. Devices are configured to observe your available firmware revisions on the Golioth Cloud. When a new version is available, the device downloads and flashes the binary. In this implementation, the connection between cloud and device is secured using TLS/DTLS, and the signed firmware binary is confirmed by MCUboot before the upgrade occurs.

1. A working sample can be found on the Golioth Firmware SDK repository
2. The Golioth OTA documentation includes complete information about the versioning process

Eclipse hawkBit™ Eclipse hawkBit™ is an update server framework that uses polling on a REST api to detect firmware updates. When a new update is detected, the binary is downloaded and installed. MCUboot can be used to verify the signature before upgrading the firmware.

There is a hawkbit-api sample included in the Zephyr mgmt-samples section.

UpdateHub UpdateHub is a platform for remotely updating embedded devices. Updates can be manually triggered or monitored via polling. When a new update is detected, the binary is downloaded and installed. MCUboot can be used to verify the signature before upgrading the firmware.

There is an updatehub-fota sample included in the Zephyr mgmt-samples section.

SMP Server A Simple Management Protocol (SMP) server can be used to update firmware via Bluetooth Low Energy (BLE) or UDP. *MCUmgr* is used to send a signed firmware binary to the remote device where it is verified by MCUboot before the upgrade occurs.

There is an smp-svr sample included in the Zephyr mgmt-samples section.

Lightweight M2M (LWM2M) The *Lightweight M2M (LWM2M)* protocol includes support for firmware update via `CONFIG_LWM2M_FIRMWARE_UPDATE_OBJ_SUPPORT`. Devices securely connect to an LwM2M server using DTLS. A lwm2m-client sample is available but it does not demonstrate the firmware update feature.

4.5.9 EC Host Command

Overview

The host command protocol defines the interface for a host, or application processor, to communicate with a target embedded controller (EC). The EC Host command subsystem implements the target side of the protocol, generating responses to commands sent by the host. The host command protocol interface supports multiple versions, but this subsystem implementation only support protocol version 3.

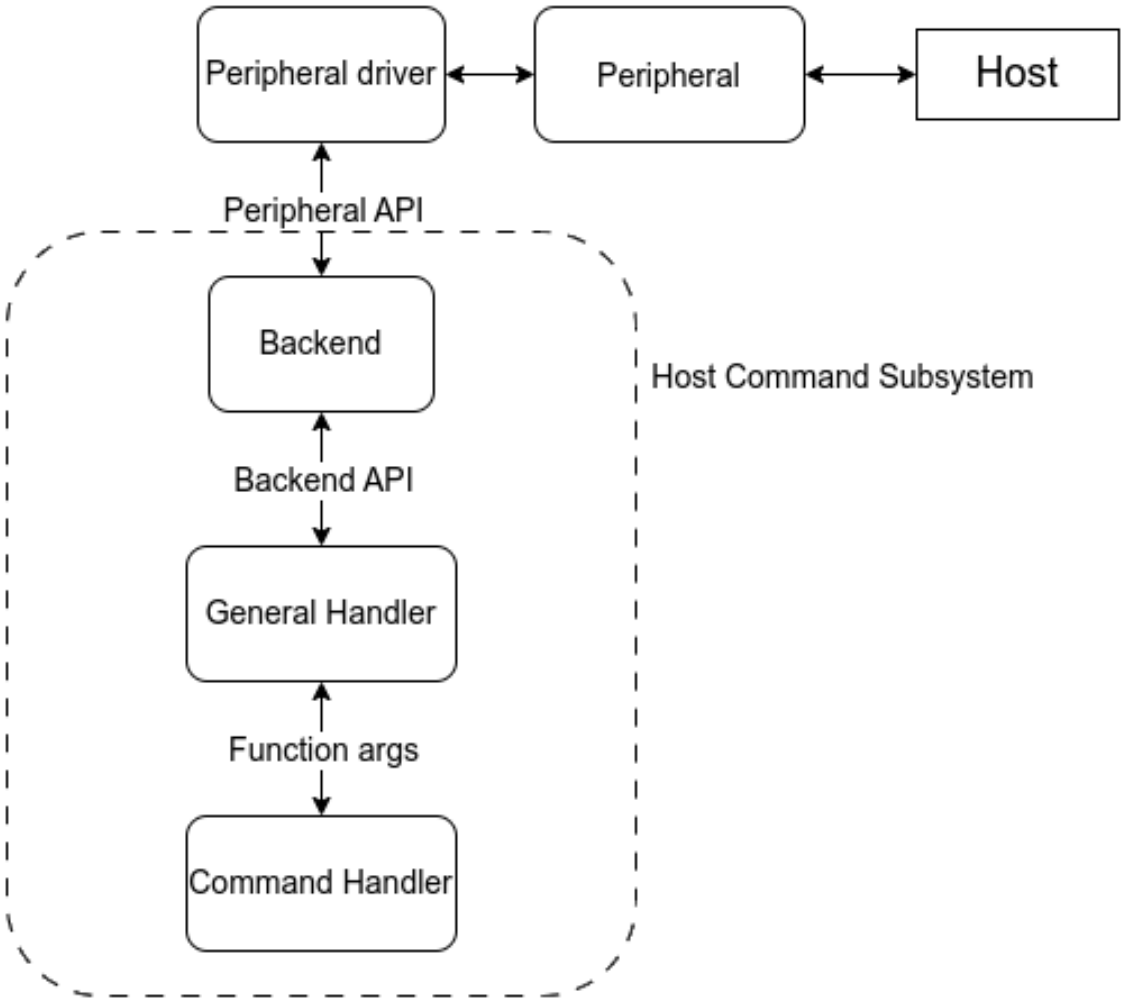
Architecture

The Host Command subsystem contains a few components:

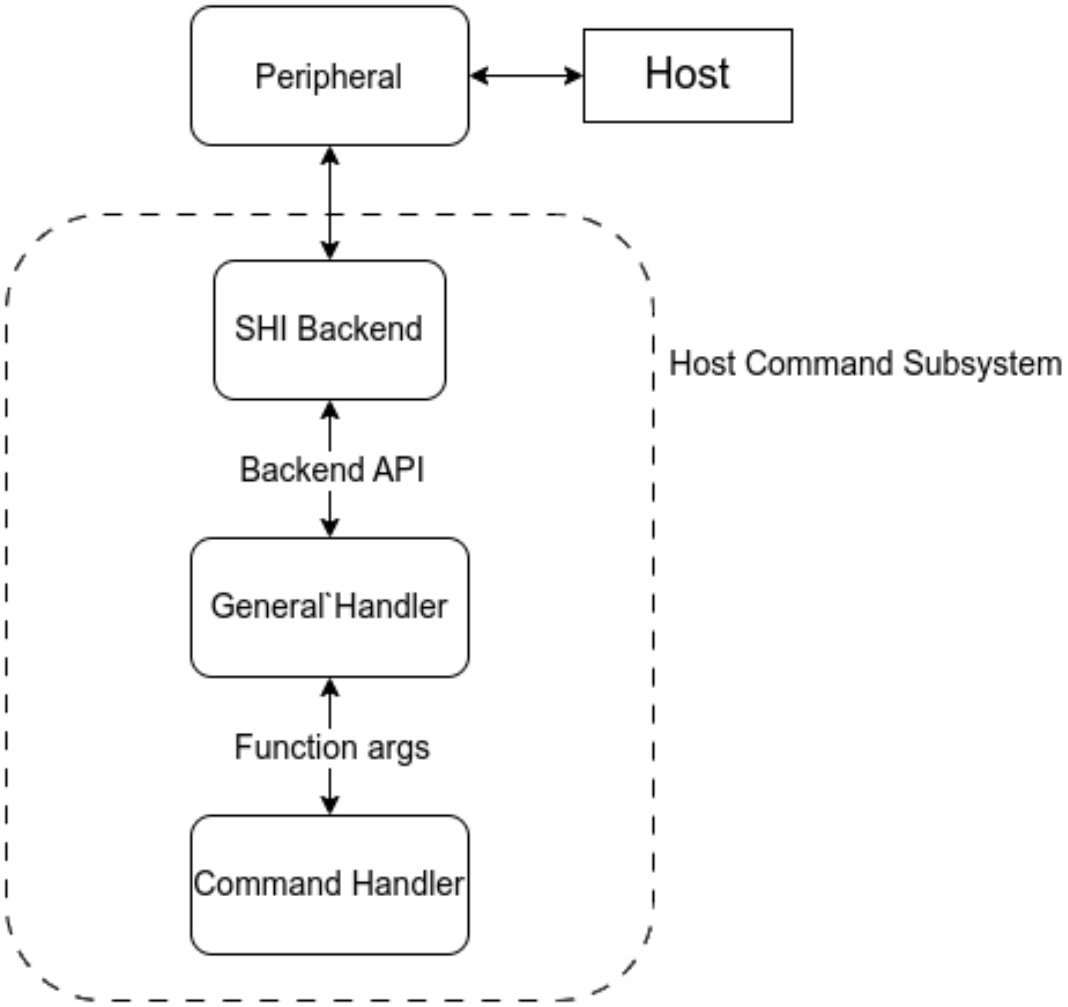
- Backend
- General handler
- Command handler

The backend is a layer between a peripheral driver and the general handler. It is responsible for sending and receiving commands via chosen peripheral.

The general handler validates data from the backend e.g. check sizes, checksum, etc. If the command is valid and the user has provided a handler for a received command id, the command handler is called.



SHI (Serial Host Interface) is different to this because it is used only for communication with a host. SHI does not have API itself, thus the backend and peripheral driver layers are combined into one backend layer.



Another case is SPI. Unfortunately, the current SPI API can't be used to handle the host commands communication. The main issues are unknown command size sent by the host (the SPI transaction sends/receives specific number of bytes) and need to constant sending status byte (the SPI module is enabled and disabled per transaction). It forces implementing the SPI driver within a backend, as it is done for SHI. That means a SPI backend has to be implemented per chip family. However, it can be changed in the future once the SPI API is extended to host command needs. Please check the discussion.

That approach requires configuring the SPI dts node in a special way. The main compatible string of a SPI node has changed to use the Host Command version of a SPI driver. The rest of the properties should be configured as usual. Example of the SPI node for STM32:

```
&spi1 {
        /* Change the compatible string to use the Host Command version of the
         * STM32 SPI driver
         */
        compatible = "st,stm32-spi-host-cmd";
        status = "okay";

        dmas = <&dma2 3 3 0x38440 0x03>,
             <&dma2 0 3 0x38480 0x03>;
        dma-names = "tx", "rx";
        /* This field is used to point at our CS pin */
        cs-gpios = <&gpioa 4 (GPIO_ACTIVE_LOW | GPIO_PULL_UP)>;
};
```

The STM32 SPI host command backend driver supports the `st,stm32h7-spi` and `st,stm32-spi-fifo` variant implementations. To enable these variants, append the corresponding compatible string. For example, to enable FIFO support and support for the STM32H7 SoCs, modify the compatible string as shown.

```
&spi1 {
    compatible = "st,stm32h7-spi", "st,stm32-spi-fifo", "st,stm32-spi-host-cmd";
    ...
};
```

The chip that runs Zephyr is a SPI slave and the *cs-gpios* property is used to point our CS pin. For the SPI, it is required to set the backend chosen node `zephyr,host-cmd-spi-backend`.

The supported backend and peripheral drivers:

- Simulator
- SHI - ITE and NPCX
- eSPI - any eSPI slave driver that support `CONFIG_ESPI_PERIPHERAL_EC_HOST_CMD` and `CONFIG_ESPI_PERIPHERAL_CUSTOM_OPCODE`
- UART - any UART driver that supports the asynchronous API
- SPI - STM32

Initialization

If the application configures one of the following backend chosen nodes and `CONFIG_EC_HOST_CMD_INITIALIZE_AT_BOOT` is set, then the corresponding backend initializes the host command subsystem by calling *ec_host_cmd_init()*:

- `zephyr,host-cmd-espi-backend`
- `zephyr,host-cmd-shi-backend`
- `zephyr,host-cmd-uart-backend`
- `zephyr,host-cmd-spi-backend`

If no backend chosen node is configured, the application must call the *ec_host_cmd_init()* function directly. This way of initialization is useful if a backend is chosen in runtime based on e.g. GPIO state.

Buffers

The host command communication requires buffers for rx and tx. The buffers are be provided by the general handler if `CONFIG_EC_HOST_CMD_HANDLER_RX_BUFFER_SIZE` > 0 for rx buffer and `CONFIG_EC_HOST_CMD_HANDLER_TX_BUFFER_SIZE` > 0 for the tx buffer. The shared buffers are useful for applications that use multiple backends. Defining separate buffers by every backend would increase the memory usage. However, some buffers can be defined by a peripheral driver e.g. eSPI. These ones should be reused as much as possible.

Logging

The host command has an embedded logging system of the ongoing communication. The are a few logging levels:

- *LOG_INF* is used to log a command id of a new command and not success responses. Repeats of the same command are not logged
- *LOG_DBG* logs every command, even repeats

- *LOG_DBG* + CONFIG_EC_HOST_CMD_LOG_DBG_BUFFERS logs every command and responses with the data buffers

API Reference

group **ec_host_cmd_interface**

EC Host Command Interface.

Since
2.4

Version
0.1.0

Defines

EC_HOST_CMD_HANDLER(_id, _function, _version_mask, _request_type, _response_type)

Statically define and register a host command handler.

Helper macro to statically define and register a host command handler that has a compile-time-fixed sizes for its both request and response structures.

Parameters

- **_id** – Id of host command to handle request for.
- **_function** – Name of handler function.
- **_version_mask** – The bitfield of all versions that the *_function* supports. E.g. *BIT(0)* corresponds to version 0.
- **_request_type** – The datatype of the request parameters for *_function*.
- **_response_type** – The datatype of the response parameters for *_function*.

EC_HOST_CMD_HANDLER_UNBOUND(_id, _function, _version_mask)

Statically define and register a host command handler without sizes.

Helper macro to statically define and register a host command handler whose request or response structure size is not known as compile time.

Parameters

- **_id** – Id of host command to handle request for.
- **_function** – Name of handler function.
- **_version_mask** – The bitfield of all versions that the *_function* supports. E.g. *BIT(0)* corresponds to version 0.

Typedefs

typedef int (***ec_host_cmd_backend_api_init**)(const struct ec_host_cmd_backend *backend, struct *ec_host_cmd_rx_ctx* *rx_ctx, struct *ec_host_cmd_tx_buf* *tx)

Initialize a host command backend.

This routine initializes a host command backend. It includes initialization a device used to communication and setting up buffers. This function is called by the ec_host_cmd_init function.

Param backend
[in] Pointer to the backend structure for the driver instance.

Param rx_ctx
[inout] Pointer to the receive context object. These objects are used to receive data from the driver when the host sends data. The buf member can be assigned by the backend.

Param tx
[inout] Pointer to the transmit buffer object. The buf and len_max members can be assigned by the backend. These objects are used to send data by the backend with the ec_host_cmd_backend_api_send function.

Retval 0
if successful

typedef int (***ec_host_cmd_backend_api_send**)(const struct ec_host_cmd_backend *backend)

Sends data to the host.

Sends data from tx buf that was passed via ec_host_cmd_backend_api_init function.

Param backend
Pointer to the backed to send data.

Retval 0
if successful.

typedef void (***ec_host_cmd_user_cb_t**)(const struct *ec_host_cmd_rx_ctx* *rx_ctx, void *user_data)

typedef enum *ec_host_cmd_status* (***ec_host_cmd_in_progress_cb_t**)(void *user_data)

typedef enum *ec_host_cmd_status* (***ec_host_cmd_handler_cb**)(struct *ec_host_cmd_handler_args* *args)

Enums

enum **ec_host_cmd_status**

Host command response codes (16-bit).

Values:

enumerator **EC_HOST_CMD_SUCCESS** = 0

Host command was successful.

enumerator **EC_HOST_CMD_INVALID_COMMAND** = 1

The specified command id is not recognized or supported.

enumerator **EC_HOST_CMD_ERROR** = 2

Generic Error.

enumerator **EC_HOST_CMD_INVALID_PARAM** = 3

One of more of the input request parameters is invalid.

enumerator EC_HOST_CMD_ACCESS_DENIED = 4
Host command is not permitted.

enumerator EC_HOST_CMD_INVALID_RESPONSE = 5
Response was invalid (e.g.
not version 3 of header).

enumerator EC_HOST_CMD_INVALID_VERSION = 6
Host command id version unsupported.

enumerator EC_HOST_CMD_INVALID_CHECKSUM = 7
Checksum did not match.

enumerator EC_HOST_CMD_IN_PROGRESS = 8
A host command is currently being processed.

enumerator EC_HOST_CMD_UNAVAILABLE = 9
Requested information is currently unavailable.

enumerator EC_HOST_CMD_TIMEOUT = 10
Timeout during processing.

enumerator EC_HOST_CMD_OVERFLOW = 11
Data or table overflow.

enumerator EC_HOST_CMD_INVALID_HEADER = 12
Header is invalid or unsupported (e.g.
not version 3 of header).

enumerator EC_HOST_CMD_REQUEST_TRUNCATED = 13
Did not receive all expected request data.

enumerator EC_HOST_CMD_RESPONSE_TOO_BIG = 14
Response was too big to send within one response packet.

enumerator EC_HOST_CMD_BUS_ERROR = 15
Error on underlying communication bus.

enumerator EC_HOST_CMD_BUSY = 16
System busy.
Should retry later.

enumerator EC_HOST_CMD_INVALID_HEADER_VERSION = 17
Header version invalid.

enumerator EC_HOST_CMD_INVALID_HEADER_CRC = 18
Header CRC invalid.

enumerator EC_HOST_CMD_INVALID_DATA_CRC = 19
Data CRC invalid.

enumerator EC_HOST_CMD_DUP_UNAVAILABLE = 20
Can't resend response.

enumerator EC_HOST_CMD_MAX = UINT16_MAX

enum ec_host_cmd_log_level
Values:

enumerator EC_HOST_CMD_DEBUG_OFF

enumerator EC_HOST_CMD_DEBUG_NORMAL

enumerator EC_HOST_CMD_DEBUG_EVERY

enumerator EC_HOST_CMD_DEBUG_PARAMS

enumerator EC_HOST_CMD_DEBUG_MODES

enum ec_host_cmd_state
Values:

enumerator EC_HOST_CMD_STATE_DISABLED = 0

enumerator EC_HOST_CMD_STATE_RECEIVING

enumerator EC_HOST_CMD_STATE_PROCESSING

enumerator EC_HOST_CMD_STATE_SENDING

Functions

struct ec_host_cmd_backend *ec_host_cmd_backend_get_espi(const struct *device* *dev)
Get the eSPI Host Command backend pointer.

Get the eSPI pointer backend and pass a pointer to eSPI device instance that will be used for the Host Command communication.

Parameters

- dev – Pointer to eSPI device instance.

Return values
The – eSPI backend pointer.

struct ec_host_cmd_backend *ec_host_cmd_backend_get_shi_npcx(void)
Get the SHI NPCX Host Command backend pointer.

Return values
the – SHI NPCX backend pointer

struct ec_host_cmd_backend *ec_host_cmd_backend_get_shi_ite(void)

Get the SHI ITE Host Command backend pointer.

Return values

the – SHI ITE backend pointer

struct ec_host_cmd_backend *ec_host_cmd_backend_get_uart(const struct *device* *dev)

Get the UART Host Command backend pointer.

Get the UART pointer backend and pass a pointer to UART device instance that will be used for the Host Command communication.

Parameters

- dev – Pointer to UART device instance.

Return values

The – UART backend pointer.

struct ec_host_cmd_backend *ec_host_cmd_backend_get_spi(struct *gpio_dt_spec* *cs)

Get the SPI Host Command backend pointer.

Get the SPI pointer backend and pass a chip select pin that will be used for the Host Command communication.

Parameters

- cs – Chip select pin..

Return values

The – SPI backend pointer.

int ec_host_cmd_init(struct ec_host_cmd_backend *backend)

Initialize the host command subsystem.

This routine initializes the host command subsystem. It includes initialization of a backend and the handler. When the application configures the zephyr,host-cmd-espi-backend/zephyr,host-cmd-shi-backend/ zephyr,host-cmd-uart-backend chosen node and CONFIG_EC_HOST_CMD_INITIALIZE_AT_BOOT is set, the chosen backend automatically calls this routine at CONFIG_EC_HOST_CMD_INIT_PRIORITY . Applications that require a run-time selection of the backend must set CONFIG_EC_HOST_CMD_INITIALIZE_AT_BOOT to n and must explicitly call this routine.

Parameters

- backend – **[in]** Pointer to the backend structure to initialize.

Return values

0 – if successful

int ec_host_cmd_send_response(enum *ec_host_cmd_status* status, const struct *ec_host_cmd_handler_args* *args)

Send the host command response.

This routine sends the host command response. It should be used to send IN_PROGRESS status or if the host command handler doesn't return e.g. reboot command.

Parameters

- status – **[in]** Host command status to be sent.
- args – **[in]** Pointer of a structure passed to the handler.

Return values

0 – if successful.

void **ec_host_cmd_rx_notify**(void)

Signal a new host command.

Signal that a new host command has been received. The function should be called by a backend after copying data to the rx buffer and setting the length.

void **ec_host_cmd_set_user_cb**(*ec_host_cmd_user_cb_t* cb, void *user_data)

Install a user callback for receiving a host command.

It allows installing a custom procedure needed by a user after receiving a command.

Parameters

- **cb** – **[in]** A callback to be installed.
- **user_data** – **[in]** User data to be passed to the callback.

const struct *ec_host_cmd* ***ec_host_cmd_get_hc**(void)

Get the main ec host command structure.

This routine returns a pointer to the main host command structure. It allows the application code to get inside information for any reason e.g. the host command thread id.

Return values

A – pointer to the main host command structure

FUNC_NORETURN void **ec_host_cmd_task**(void)

The thread function for Host Command subsystem.

This routine calls the Host Command thread entry function. If CONFIG_EC_HOST_CMD_DEDICATED_THREAD is not defined, a new thread is not created, and this function has to be called by application code. It doesn't return.

int **ec_host_cmd_add_suppressed**(uint16_t cmd_id)

Add a suppressed command.

Suppressed commands are not logged. Add a command to be suppressed.

Parameters

- **cmd_id** – **[in]** A command id to be suppressed.

Return values

0 – if successful, -EIO if exceeded max number of suppressed commands.

struct **ec_host_cmd_rx_ctx**

#include <backend.h> Context for host command backend and handler to pass rx data.

Public Members

uint8_t ***buf**

Buffer to hold received data.

The buffer is provided by the handler if CONFIG_EC_HOST_CMD_HANDLER_RX_BUFFER_SIZE > 0. Otherwise, the backend should provide the buffer on its own and overwrites *buf* pointer and *len_max* in the init function.

size_t **len**

Number of bytes written to *buf* by backend.

size_t **len_max**

Maximum number of bytes to receive with one request packet.

struct **ec_host_cmd_tx_buf**

#include <backend.h> Context for host command backend and handler to pass tx data.

Public Members

void ***buf**

Data to write to the host The buffer is provided by the handler if CONFIG_EC_HOST_CMD_HANDLER_TX_BUFFER_SIZE > 0.

Otherwise, the backend should provide the buffer on its own and overwrites *buf* pointer and *len_max* in the init function.

size_t **len**

Number of bytes to write from *buf*.

size_t **len_max**

Maximum number of bytes to send with one response packet.

struct **ec_host_cmd_backend_api**

#include <backend.h>

struct **ec_host_cmd**

#include <ec_host_cmd.h>

Public Members

struct k_sem **rx_ready**

The backend gives rx_ready (by calling the ec_host_cmd_send_receive function), when data in rx_ctx are ready.

The handler takes rx_ready to read data in rx_ctx.

enum *ec_host_cmd_status* **rx_status**

Status of the rx data checked in the ec_host_cmd_send_received function.

ec_host_cmd_user_cb_t **user_cb**

User callback after receiving a command.

It is called by the ec_host_cmd_send_received function.

struct **ec_host_cmd_handler_args**

#include <ec_host_cmd.h> Arguments passed into every installed host command handler.

Public Members

void *reserved

Reserved for compatibility.

uint16_t command

Command identifier.

uint8_t version

The version of the host command that is being requested.

This will be a value that has been static registered as valid for the handler.

const void *input_buf

The incoming data that can be cast to the handlers request type.

uint16_t input_buf_size

The number of valid bytes that can be read from *input_buf*.

void *output_buf

The data written to this buffer will be send to the host.

uint16_t output_buf_max

Maximum number of bytes that can be written to the *output_buf*.

uint16_t output_buf_size

Number of bytes of *output_buf* to send to the host.

struct ec_host_cmd_handler

#include <ec_host_cmd.h> Structure use for statically registering host command handlers.

Public Members

ec_host_cmd_handler_cb handler

Callback routine to process commands that match *id*.

uint16_t id

The numerical command id used as the lookup for commands.

uint16_t version_mask

The bitfield of all versions that the *handler* supports, where each bit value represents that the *handler* supports that version.

E.g. *BIT(0)* corresponds to version 0.

uint16_t min_rqt_size

The minimum *input_buf_size* enforced by the framework before passing to the handler.

uint16_t **min_rsp_size**

The minimum *output_buf_size* enforced by the framework before passing to the handler.

struct **ec_host_cmd_request_header**

#include <ec_host_cmd.h> Header for requests from host to embedded controller.

Represent the over-the-wire header in LE format for host command requests. This represent version 3 of the host command header. The requests are always sent from host to embedded controller.

Public Members

uint8_t **prtcl_ver**

Should be 3.

The EC will return EC_HOST_CMD_INVALID_HEADER if it receives a header with a version it doesn't know how to parse.

uint8_t **checksum**

Checksum of response and data; sum of all bytes including checksum.

Should total to 0.

uint16_t **cmd_id**

Id of command that is being sent.

uint8_t **cmd_ver**

Version of the specific *cmd_id* being requested.

Valid versions start at 0.

uint8_t **reserved**

Unused byte in current protocol version; set to 0.

uint16_t **data_len**

Length of data which follows this header.

struct **ec_host_cmd_response_header**

#include <ec_host_cmd.h> Header for responses from embedded controller to host.

Represent the over-the-wire header in LE format for host command responses. This represent version 3 of the host command header. Responses are always sent from embedded controller to host.

Public Members

uint8_t **prtcl_ver**

Should be 3.

uint8_t **checksum**

Checksum of response and data; sum of all bytes including checksum.

Should total to 0.

uint16_t **result**

A *ec_host_cmd_status* response code for specific command.

uint16_t **data_len**

Length of data which follows this header.

uint16_t **reserved**

Unused bytes in current protocol version; set to 0.

4.5.10 SMP Groups

Zephyr Management Group

Zephyr management group defines the following commands:

Command ID	Command description
0	Erase storage

Erase storage command Erase storage command allows clearing the `storage_partition` flash partition on a device, generally this is used when switching to a new application build if the application uses storage that should be cleared (application dependent).

Erase storage request Erase storage request header fields:

OP	Group ID	Command ID
2	63	0

The command sends sends empty CBOR map as data.

Erase storage response Read setting response header fields:

OP	Group ID	Command ID
3	63	0

The command sends an empty CBOR map as data if successful. In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1

```
{
    (str)"rc"       : (int)
}
```

where:

"err" -> "group"	*mcumgr_group_t* group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	*mcumgr_err_t* only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

4.6 Digital Signal Processing (DSP)

- *Using zDSP*
- *Optimizing for your architecture*
- *API Reference*

The DSP API provides an architecture agnostic way for signal processing. Currently, the API will work on any architecture but will likely not be optimized. The status of the various architectures can be found below:

Architecture	Status
ARC	Optimized
ARM	Optimized
ARM64	Optimized
MIPS	Unoptimized
NIOS2	Unoptimized
POSIX	Unoptimized
RISCV	Unoptimized
RISCV64	Unoptimized
SPARC	Unoptimized
X86	Unoptimized
XTENSA	Unoptimized

4.6.1 Using zDSP

zDSP provides various backend options which are selected automatically for the application. By default, including the CMSIS module will enable all architectures to use the zDSP APIs. This can

be done by setting:

```
CONFIG_CMSIS_DSP=y
```

If your application requires some additional customization, it's possible to enable `CONFIG_DSP_BACKEND_CUSTOM` which means that the application is responsible for providing the implementation of the zDSP library.

4.6.2 Optimizing for your architecture

If your architecture is showing as `Unoptimized`, it's possible to add a new zDSP backend to better support it. To do that, a new Kconfig option should be added to `subsys/dsp/Kconfig` along with the required dependencies and the `default` set for `DSP_BACKEND` Kconfig choice.

Next, the implementation should be added at `subsys/dsp/<backend>/` and linked in at `subsys/dsp/CMakeLists.txt`. To add architecture-specific attributes, its corresponding Kconfig option should be added to `subsys/dsp/Kconfig` and use them to update `DSP_DATA` and `DSP_STATIC_DATA` in `include/zephyr/dsp/dsp.h`.

4.6.3 API Reference

group **math_dsp**

DSP Interface.

Since
3.3

Version
0.1.0

Typedefs

typedef int8_t **q7_t**

8-bit fractional data type in 1.7 format.

typedef int16_t **q15_t**

16-bit fractional data type in 1.15 format.

typedef int32_t **q31_t**

32-bit fractional data type in 1.31 format.

typedef int64_t **q63_t**

64-bit fractional data type in 1.63 format.

typedef __fp16 **float16_t**

16-bit floating point type definition.

typedef float **float32_t**

32-bit floating-point type definition.

typedef double **float64_t**

64-bit floating-point type definition.

4.7 File Systems

Zephyr RTOS Virtual Filesystem Switch (VFS) allows applications to mount multiple file systems at different mount points (e.g., `/fatfs` and `/lfs`). The mount point data structure contains all the necessary information required to instantiate, mount, and operate on a file system. The File system Switch decouples the applications from directly accessing an individual file system's specific API or internal functions by introducing file system registration mechanisms.

In Zephyr, any file system implementation or library can be plugged into or pulled out through a file system registration API. Each file system implementation must have a globally unique integer identifier; use `FS_TYPE_EXTERNAL_BASE` to avoid clashes with in-tree identifiers.

```
int fs_register(int type, const struct fs_file_system_t *fs);

int fs_unregister(int type, const struct fs_file_system_t *fs);
```

Zephyr RTOS supports multiple instances of a file system by making use of the mount point as the disk volume name, which is used by the file system library while formatting or mounting a disk.

A file system is declared as:

```
static struct fs_mount_t mp = {
.type = FS_FATFS,
.mnt_point = FATFS_MNTP,
.fs_data = &fat_fs,
};
```

where

- `FS_FATFS` is the file system type like FATFS or LittleFS.
- `FATFS_MNTP` is the mount point where the file system will be mounted.
- `fat_fs` is the file system data which will be used by fs_mount() API.

4.7.1 Samples

Samples for the VFS are mainly supplied in `samples/subsys/fs`, although various examples of the VFS usage are provided as important functionalities in samples for different subsystems. Here is the list of samples worth looking at:

- `samples/subsys/fs/fat_fs` is an example of FAT file system usage with SDHC media;
- **`samples/subsys/shell/fs` is an example of Shell fs subsystem, using internal flash partition**
  formatted to LittleFS;
- **`samples/subsys/usb/mass/` example of USB Mass Storage device that uses FAT FS driver with RAM**
  or SPI connected FLASH, or LittleFS in flash, depending on the sample configuration.

4.7.2 API Reference

Related code samples

File system manipulation
Use file system API with various filesystems and storage devices.

File system shell
Access a LittleFS file system partition in flash using the file system shell.

Format filesystem
Format different storage devices for different file systems.

LittleFS filesystem
Use file system API over LittleFS.

USB Mass Storage
Expose board's RAM or FLASH as a USB disk using USB Mass Storage driver.

group `file_system_api`

File System APIs.

Since
1.5

Version
1.0.0

fs_open open and creation mode flags

`FS_O_READ`
Open for read flag.

`FS_O_WRITE`
Open for write flag.

`FS_O_RDWR`
Open for read-write flag combination.

`FS_O_MODE_MASK`
Bitmask for read and write flags.

`FS_O_CREATE`
Create file if it does not exist.

`FS_O_APPEND`
Open/create file for append.

`FS_O_TRUNC`
Truncate the file while opening.

FS_O_FLAGS_MASK
: Bitmask for open/create flags.

FS_O_MASK
: Bitmask for open flags.

fs_seek whence parameter values

FS_SEEK_SET
: Seek from the beginning of file.

FS_SEEK_CUR
: Seek from a current position.

FS_SEEK_END
: Seek from the end of file.

Defines

FS_MOUNT_FLAG_NO_FORMAT
: Flag prevents formatting device if requested file system not found.

FS_MOUNT_FLAG_READ_ONLY
: Flag makes mounted file system read-only.

FS_MOUNT_FLAG_AUTOMOUNT
: Flag used in pre-defined mount structures that are to be mounted on startup.

  This flag has no impact in user-defined mount structures.

FS_MOUNT_FLAG_USE_DISK_ACCESS
: Flag requests file system driver to use Disk Access API.

  When the flag is set to the *fs_mount_t.flags* prior to fs_mount call, a file system needs to use the Disk Access API, otherwise mount callback for the driver should return -ENOSUP; when the flag is not set the file system driver should use Flash API by default, unless it only supports Disc Access API. When file system will use Disk Access API and the flag is not set, the mount callback for the file system should set the flag on success.

FSTAB_ENTRY_DT_MOUNT_FLAGS(node_id)
: Get the common mount flags for an fstab entry.

  Parameters

  - **node_id** – the node identifier for a child entry in a zephyr,fstab node.

  Returns
  a value suitable for initializing an *fs_mount_t* flags member.

FS_FSTAB_ENTRY(node_id)
: The name under which a zephyr,fstab entry mount structure is defined.

  Parameters

- **node_id** – the node identifier for a child entry in a zephyr,fstab node.

FS_FSTAB_DECLARE_ENTRY(node_id)

Generate a declaration for the externally defined fstab entry.

This will evaluate to the name of a struct *fs_mount_t* object.

Parameters

- **node_id** – the node identifier for a child entry in a zephyr,fstab node.

Enums

enum **fs_dir_entry_type**

Enumeration for directory entry types.

Values:

enumerator **FS_DIR_ENTRY_FILE** = 0

Identifier for file entry.

enumerator **FS_DIR_ENTRY_DIR**

Identifier for directory entry.

Enumeration to uniquely identify file system types.

Zephyr supports in-tree file systems and external ones. Each requires a unique identifier used to register the file system implementation and to associate a mount point with the file system type. This anonymous enum defines global identifiers for the in-tree file systems.

External file systems should be registered using unique identifiers starting at FS_TYPE_EXTERNAL_BASE. It is the responsibility of applications that use external file systems to ensure that these identifiers are unique if multiple file system implementations are used by the application.

Values:

enumerator **FS_FATFS** = 0

Identifier for in-tree FatFS file system.

enumerator **FS_LITTLEFS**

Identifier for in-tree LittleFS file system.

enumerator **FS_EXT2**

Identifier for in-tree Ext2 file system.

enumerator **FS_TYPE_EXTERNAL_BASE**

Base identifier for external file systems.

Functions

static inline void **fs_file_t_init**(struct *fs_file_t* *zfp)

Initialize *fs_file_t* object.

Initializes the *fs_file_t* object; the function needs to be invoked on object before first use with fs_open.

Parameters

- **zfp** – Pointer to file object

static inline void **fs_dir_t_init**(struct *fs_dir_t* *zdp)

Initialize *fs_dir_t* object.

Initializes the *fs_dir_t* object; the function needs to be invoked on object before first use with fs_opendir.

Parameters

- **zdp** – Pointer to file object

int **fs_open**(struct *fs_file_t* *zfp, const char *file_name, fs_mode_t flags)

Open or create file.

Opens or possibly creates a file and associates a stream with it. Successfully opened file, when no longer in use, should be closed with *fs_close()*.

`flags` can be 0 or a binary combination of one or more of the following identifiers:

- `FS_O_READ` open for read
- `FS_O_WRITE` open for write
- `FS_O_RDWR` open for read/write (`FS_O_READ | FS_O_WRITE`)
- `FS_O_CREATE` create file if it does not exist
- `FS_O_APPEND` move to end of file before each write
- `FS_O_TRUNC` truncate the file

Warning

If `flags` are set to 0 the function will open file, if it exists and is accessible, but you will have no read/write access to it.

Parameters

- **zfp** – Pointer to a file object
- **file_name** – The name of a file to open
- **flags** – The mode flags

Return values

- **0** – on success;
- **-EBUSY** – when zfp is already used;
- **-EINVAL** – when a bad file name is given;
- **-EROFS** – when opening read-only file for write, or attempting to create a file on a system that has been mounted with the FS_MOUNT_FLAG_READ_ONLY flag;
- **-ENOENT** – when the file does not exist at the path;
- **-ENOTSUP** – when not implemented by underlying file system driver;
- **-EACCES** – when trying to truncate a file without opening it for write.

- `<0` – an other negative errno code, depending on a file system back-end.

int `fs_close`(struct *fs_file_t* *zfp)

Close file.

Flushes the associated stream and closes the file.

Parameters

- `zfp` – Pointer to the file object

Return values

- `0` – on success;
- `-ENOTSUP` – when not implemented by underlying file system driver;
- `<0` – a negative errno code on error.

int `fs_unlink`(const char *path)

Unlink file.

Deletes the specified file or directory

Parameters

- `path` – Path to the file or directory to delete

Return values

- `0` – on success;
- `-EINVAL` – when a bad file name is given;
- `-EROFS` – if file is read-only, or when file system has been mounted with the FS_MOUNT_FLAG_READ_ONLY flag;
- `-ENOTSUP` – when not implemented by underlying file system driver;
- `<0` – an other negative errno code on error.

int `fs_rename`(const char *from, const char *to)

Rename file or directory.

Performs a rename and / or move of the specified source path to the specified destination. The source path can refer to either a file or a directory. All intermediate directories in the destination path must already exist. If the source path refers to a file, the destination path must contain a full filename path, rather than just the new parent directory. If an object already exists at the specified destination path, this function causes it to be unlinked prior to the rename (i.e., the destination gets clobbered).

Note

Current implementation does not allow moving files between mount points.

Parameters

- `from` – The source path
- `to` – The destination path

Return values

- `0` – on success;
- `-EINVAL` – when a bad file name is given, or when rename would cause move between mount points;

- **-EROFS** – if file is read-only, or when file system has been mounted with the FS_MOUNT_FLAG_READ_ONLY flag;
- **-ENOTSUP** – when not implemented by underlying file system driver;
- **<0** – an other negative errno code on error.

ssize_t **fs_read**(struct *fs_file_t* *zfp, void *ptr, size_t size)

Read file.

Reads up to `size` bytes of data to `ptr` pointed buffer, returns number of bytes read. A returned value may be lower than `size` if there were fewer bytes available than requested.

Parameters

- **zfp** – Pointer to the file object
- **ptr** – Pointer to the data buffer
- **size** – Number of bytes to be read

Return values

- **>=0** – a number of bytes read, on success;
- **-EBADF** – when invoked on zfp that represents unopened/closed file;
- **-ENOTSUP** – when not implemented by underlying file system driver;
- **<0** – a negative errno code on error.

ssize_t **fs_write**(struct *fs_file_t* *zfp, const void *ptr, size_t size)

Write file.

Attempts to write `size` number of bytes to the specified file. If a negative value is returned from the function, the file pointer has not been advanced. If the function returns a non-negative number that is lower than `size`, the global `errno` variable should be checked for an error code, as the device may have no free space for data.

Parameters

- **zfp** – Pointer to the file object
- **ptr** – Pointer to the data buffer
- **size** – Number of bytes to be written

Return values

- **>=0** – a number of bytes written, on success;
- **-EBADF** – when invoked on zfp that represents unopened/closed file;
- **-ENOTSUP** – when not implemented by underlying file system driver;
- **<0** – an other negative errno code on error.

int **fs_seek**(struct *fs_file_t* *zfp, off_t offset, int whence)

Seek file.

Moves the file position to a new location in the file. The `offset` is added to file position based on the `whence` parameter.

Parameters

- **zfp** – Pointer to the file object
- **offset** – Relative location to move the file pointer to
- **whence** – Relative location from where offset is to be calculated.

- FS_SEEK_SET for the beginning of the file;
- FS_SEEK_CUR for the current position;
- FS_SEEK_END for the end of the file.

Return values

- `0` – on success;
- `-EBADF` – when invoked on zfp that represents unopened/closed file;
- `-ENOTSUP` – if not supported by underlying file system driver;
- `<0` – an other negative errno code on error.

off_t **fs_tell**(struct *fs_file_t* *zfp)

Get current file position.

Retrieves and returns the current position in the file stream.

The current revision does not validate the file object.

Parameters

- `zfp` – Pointer to the file object

Return values

- `>=` – 0 a current position in file;
- `-EBADF` – when invoked on zfp that represents unopened/closed file;
- `-ENOTSUP` – if not supported by underlying file system driver;
- `<0` – an other negative errno code on error.

int **fs_truncate**(struct *fs_file_t* *zfp, off_t length)

Truncate or extend an open file to a given size.

Truncates the file to the new length if it is shorter than the current size of the file. Expands the file if the new length is greater than the current size of the file. The expanded region would be filled with zeroes.

Note

In the case of expansion, if the volume got full during the expansion process, the function will expand to the maximum possible length and return success. Caller should check if the expanded size matches the requested length.

Parameters

- `zfp` – Pointer to the file object
- `length` – New size of the file in bytes

Return values

- `0` – on success;
- `-EBADF` – when invoked on zfp that represents unopened/closed file;
- `-ENOTSUP` – when not implemented by underlying file system driver;
- `<0` – an other negative errno code on error.

int `fs_sync`(struct *fs_file_t* *zfp)

Flush cached write data buffers of an open file.

The function flushes the cache of an open file; it can be invoked to ensure data gets written to the storage media immediately, e.g. to avoid data loss in case if power is removed unexpectedly.

> **Note**
>
> Closing a file will cause caches to be flushed correctly so the function need not be called when the file is being closed.

Parameters

- `zfp` – Pointer to the file object

Return values

- `0` – on success;
- `-EBADF` – when invoked on zfp that represents unopened/closed file;
- `-ENOTSUP` – when not implemented by underlying file system driver;
- `<0` – a negative errno code on error.

int `fs_mkdir`(const char *path)

Directory create.

Creates a new directory using specified path.

Parameters

- `path` – Path to the directory to create

Return values

- `0` – on success;
- `-EEXIST` – if entry of given name exists;
- `-EROFS` – if `path` is within read-only directory, or when file system has been mounted with the FS_MOUNT_FLAG_READ_ONLY flag;
- `-ENOTSUP` – when not implemented by underlying file system driver;
- `<0` – an other negative errno code on error

int `fs_opendir`(struct *fs_dir_t* *zdp, const char *path)

Directory open.

Opens an existing directory specified by the path.

Parameters

- `zdp` – Pointer to the directory object
- `path` – Path to the directory to open

Return values

- `0` – on success;
- `-EINVAL` – when a bad directory path is given;
- `-EBUSY` – when zdp is already used;
- `-ENOTSUP` – when not implemented by underlying file system driver;
- `<0` – a negative errno code on error.

int `fs_readdir`(struct *fs_dir_t* *zdp, struct *fs_dirent* *entry)

Directory read entry.

Reads directory entries of an open directory. In end-of-dir condition, the function will return 0 and set the `entry->name[0]` to 0.

> **Note**
>
> : Most existing underlying file systems do not generate POSIX special directory entries "." or "..". For consistency the abstraction layer will remove these from lower layer results so higher layers see consistent results.

Parameters

- `zdp` – Pointer to the directory object
- `entry` – Pointer to zfs_dirent structure to read the entry into

Return values

- `0` – on success or end-of-dir;
- `-ENOENT` – when no such directory found;
- `-ENOTSUP` – when not implemented by underlying file system driver;
- `<0` – a negative errno code on error.

int `fs_closedir`(struct *fs_dir_t* *zdp)

Directory close.

Closes an open directory.

Parameters

- `zdp` – Pointer to the directory object

Return values

- `0` – on success;
- `-ENOTSUP` – when not implemented by underlying file system driver;
- `<0` – a negative errno code on error.

int `fs_mount`(struct *fs_mount_t* *mp)

Mount filesystem.

Perform steps needed for mounting a file system like calling the file system specific mount function and adding the mount point to mounted file system list.

> **Note**
>
> Current implementation of ELM FAT driver allows only following mount points: "/RAM:","/NAND:","/CF:","/SD:","/SD2:","/USB:","/USB2:","/USB3:" or mount points that consist of single digit, e.g: "/0:", "/1:" and so forth.

Parameters

- `mp` – Pointer to the *fs_mount_t* structure. Referenced object is not changed if the mount operation failed. A reference is captured in the fs infrastructure if the mount operation succeeds, and the application must not mutate the structure contents until fs_unmount is successfully invoked on the same pointer.

Return values

- `0` – on success;
- `-ENOENT` – when file system type has not been registered;
- `-ENOTSUP` – when not supported by underlying file system driver; when `FS_MOUNT_FLAG_USE_DISK_ACCESS` is set but driver does not support it.
- `-EROFS` – if system requires formatting but `FS_MOUNT_FLAG_READ_ONLY` has been set;
- `<0` – an other negative errno code on error.

int `fs_unmount`(struct *fs_mount_t* *mp)

Unmount filesystem.

Perform steps needed to unmount a file system like calling the file system specific unmount function and removing the mount point from mounted file system list.

Parameters

- `mp` – Pointer to the *fs_mount_t* structure

Return values

- `0` – on success;
- `-EINVAL` – if no system has been mounted at given mount point;
- `-ENOTSUP` – when not supported by underlying file system driver;
- `<0` – an other negative errno code on error.

int `fs_readmount`(int *index, const char **name)

Get path of mount point at index.

This function iterates through the list of mount points and returns the directory name of the mount point at the given `index`. On success `index` is incremented and `name` is set to the mount directory name. If a mount point with the given `index` does not exist, `name` will be set to `NULL`.

Parameters

- `index` – Pointer to mount point index
- `name` – Pointer to pointer to path name

Return values

- `0` – on success;
- `-ENOENT` – if there is no mount point with given index.

int `fs_stat`(const char *path, struct *fs_dirent* *entry)

File or directory status.

Checks the status of a file or directory specified by the `path`.

> **Note**
>
> The file on a storage device may not be updated until it is closed.

Parameters

- `path` – Path to the file or directory
- `entry` – Pointer to the zfs_dirent structure to fill if the file or directory exists.

Return values

- `0` – on success;
- `-EINVAL` – when a bad directory or file name is given;
- `-ENOENT` – when no such directory or file is found;
- `-ENOTSUP` – when not supported by underlying file system driver;
- `<0` – negative errno code on error.

int `fs_statvfs`(const char *path, struct *fs_statvfs* *stat)

Retrieves statistics of the file system volume.

Returns the total and available space in the file system volume.

Parameters

- `path` – Path to the mounted directory
- `stat` – Pointer to the zfs_statvfs structure to receive the fs statistics.

Return values

- `0` – on success;
- `-EINVAL` – when a bad path to a directory, or a file, is given;
- `-ENOTSUP` – when not implemented by underlying file system driver;
- `<0` – an other negative errno code on error.

int `fs_mkfs`(int fs_type, uintptr_t dev_id, void *cfg, int flags)

Create fresh file system.

Parameters

- `fs_type` – Type of file system to create.
- `dev_id` – Id of storage device.
- `cfg` – Backend dependent init object. If NULL then default configuration is used.
- `flags` – Additional flags for file system implementation.

Return values

- `0` – on success;
- `<0` – negative errno code on error.

int `fs_register`(int type, const struct *fs_file_system_t* *fs)

Register a file system.

Register file system with virtual file system. Number of allowed file system types to be registered is controlled with the CONFIG_FILE_SYSTEM_MAX_TYPES Kconfig option.

Parameters

- `type` – Type of file system (ex: `FS_FATFS`)
- `fs` – Pointer to File system

Return values

- `0` – on success;
- `-EALREADY` – when a file system of a given type has already been registered;

- `-ENOSCP` – when there is no space left, in file system registry, to add this file system type.

int `fs_unregister`(int type, const struct *fs_file_system_t* *fs)

Unregister a file system.

Unregister file system from virtual file system.

Parameters

- `type` – Type of file system (ex: `FS_FATFS`)
- `fs` – Pointer to File system

Return values

- `0` – on success;
- `-EINVAL` – when file system of a given type has not been registered.

struct `fs_mount_t`

#include <fs.h> File system mount info structure.

Public Members

sys_dnode_t `node`

Entry for the fs_mount_list list.

int `type`

File system type.

const char *`mnt_point`

Mount point directory name (ex: “/fatfs”)

void *`fs_data`

Pointer to file system specific data.

void *`storage_dev`

Pointer to backend storage device.

size_t `mountp_len`

Length of Mount point string.

const struct *fs_file_system_t* *`fs`

Pointer to File system interface of the mount point.

uint8_t `flags`

Mount flags.

struct `fs_dirent`

#include <fs.h> Structure to receive file or directory information.

Used in functions that read the directory entries to get file or directory information.

Public Members

enum *fs_dir_entry_type* **type**
File/directory type (FS_DIR_ENTRY_FILE or FS_DIR_ENTRY_DIR)

char **name**[MAX_FILE_NAME + 1]
Name of file or directory.

size_t **size**
Size of file (0 if directory).

struct **fs_statvfs**
#include <fs.h> Structure to receive volume statistics.

Used to retrieve information about total and available space in the volume.

Public Members

unsigned long **f_bsize**
Optimal transfer block size.

unsigned long **f_frsize**
Allocation unit size.

unsigned long **f_blocks**
Size of FS in f_frsize units.

unsigned long **f_bfree**
Number of free blocks.

struct **fs_file_t**
#include <fs_interface.h> File object representing an open file.

The object needs to be initialized with *fs_file_t_init()*.

Public Members

void ***filep**
Pointer to file object structure.

const struct *fs_mount_t* ***mp**
Pointer to mount point structure.

fs_mode_t **flags**
Open/create flags.

struct **fs_dir_t**
#include <fs_interface.h> Directory object representing an open directory.

The object needs to be initialized with *fs_dir_t_init()*.

Public Members

void *dirp
: Pointer to directory object structure.

const struct *fs_mount_t* *mp
: Pointer to mount point structure.

struct fs_file_system_t
: *#include <fs_sys.h>* File System interface structure.

File operations

int (*open)(struct *fs_file_t* *filp, const char *fs_path, fs_mode_t flags)
: Opens or creates a file, depending on flags given.
  Param filp
  File to open/create.
  Param fs_path
  Path to the file.
  Param flags
  Flags for opening/creating the file.
  Return
  0 on success, negative errno code on fail.

ssize_t (*read)(struct *fs_file_t* *filp, void *dest, size_t nbytes)
: Reads nbytes number of bytes.
  Param filp
  File to read from.
  Param dest
  Destination buffer.
  Param nbytes
  Number of bytes to read.
  Return
  Number of bytes read on success, negative errno code on fail.

ssize_t (*write)(struct *fs_file_t* *filp, const void *src, size_t nbytes)
: Writes nbytes number of bytes.
  Param filp
  File to write to.
  Param src
  Source buffer.
  Param nbytes
  Number of bytes to write.
  Return
  Number of bytes written on success, negative errno code on fail.

int (*lseek)(struct *fs_file_t* *filp, off_t off, int whence)
: Moves the file position to a new location in the file.
  Param filp
  File to move.
  Param off
  Relative offset from the position specified by whence.

Param whence
Position in the file. Possible values: SEEK_CUR, SEEK_SET, SEEK_END.
Return
New position in the file or negative errno code on fail.

off_t (*`tell`)(struct *fs_file_t* *filp)

Retrieves the current position in the file.
Param filp
File to get the current position from.
Return
Current position in the file or negative errno code on fail.

int (*`truncate`)(struct *fs_file_t* *filp, off_t length)

Truncates/expands the file to the new length.
Param filp
File to truncate/expand.
Param length
New length of the file.
Return
0 on success, negative errno code on fail.

int (*`sync`)(struct *fs_file_t* *filp)

Flushes the cache of an open file.
Param filp
File to flush.
Return
0 on success, negative errno code on fail.

int (*`close`)(struct *fs_file_t* *filp)

Flushes the associated stream and closes the file.
Param filp
File to close.
Return
0 on success, negative errno code on fail.

Directory operations

int (*`opendir`)(struct *fs_dir_t* *dirp, const char *fs_path)

Opens an existing directory specified by the path.
Param dirp
Directory to open.
Param fs_path
Path to the directory.
Return
0 on success, negative errno code on fail.

int (*`readdir`)(struct *fs_dir_t* *dirp, struct *fs_dirent* *entry)

Reads directory entries of an open directory.
Param dirp
Directory to read from.
Param entry
Next directory entry in the dirp directory.
Return
0 on success, negative errno code on fail.

int (*`closedir`)(struct *fs_dir_t* *dirp)

Closes an open directory.

Param dirp
Directory to close.

Return
0 on success, negative errno code on fail.

File system level operations

int (*`mount`)(struct *fs_mount_t* *mountp)

Mounts a file system.

Param mountp
Mount point.

Return
0 on success, negative errno code on fail.

int (*`unmount`)(struct *fs_mount_t* *mountp)

Unmounts a file system.

Param mountp
Mount point.

Return
0 on success, negative errno code on fail.

int (*`unlink`)(struct *fs_mount_t* *mountp, const char *name)

Deletes the specified file or directory.

Param mountp
Mount point.

Param name
Path to the file or directory to delete.

Return
0 on success, negative errno code on fail.

int (*`rename`)(struct *fs_mount_t* *mountp, const char *from, const char *to)

Renames a file or directory.

Param mountp
Mount point.

Param from
Path to the file or directory to rename.

Param to
New name of the file or directory.

Return
0 on success, negative errno code on fail.

int (*`mkdir`)(struct *fs_mount_t* *mountp, const char *name)

Creates a new directory using specified path.

Param mountp
Mount point.

Param name
Path to the directory to create.

Return
0 on success, negative errno code on fail.

int (***stat**)(struct *fs_mount_t* *mountp, const char *path, struct *fs_dirent* *entry)

Checks the status of a file or directory specified by the path.

Param mountp
Mount point.
Param path
Path to the file or directory.
Param entry
Directory entry.
Return
0 on success, negative errno code on fail.

int (***statvfs**)(struct *fs_mount_t* *mountp, const char *path, struct *fs_statvfs* *stat)

Returns the total and available space on the file system volume.

Param mountp
Mount point.
Param path
Path to the file or directory.
Param stat
File system statistics.
Return
0 on success, negative errno code on fail.

int (***mkfs**)(uintptr_t dev_id, void *cfg, int flags)

Formats a device to specified file system type.

Available only if `CONFIG_FILE_SYSTEM_MKFS` is enabled.

Note

This operation destroys existing data on the target device.

Param dev_id
Device identifier.
Param cfg
File system configuration.
Param flags
Formatting flags.
Return
0 on success, negative errno code on fail.

4.8 Formatted Output

Applications as well as Zephyr itself requires infrastructure to format values for user consumption. The standard C99 library `*printf()` functionality fulfills this need for streaming output devices or memory buffers, but in an embedded system devices may not accept streamed data and memory may not be available to store the formatted output.

Internal Zephyr API traditionally provided this both for `printk()` and for Zephyr's internal minimal libc, but with separate internal interfaces. Logging, tracing, shell, and other applications made use of either these APIs or standard libc routines based on build options.

The *cbprintf()* public APIs convert C99 format strings and arguments, providing output produced one character at a time through a callback mechanism, replacing the original internal functions and providing support for almost all C99 format specifications. Existing use of

s*printf() C libraries in Zephyr can be converted to *snprintfcb()* to avoid pulling in libc implementations.

Several Kconfig options control the set of features that are enabled, allowing some control over features and memory usage:

- CONFIG_CBPRINTF_FULL_INTEGRAL or CONFIG_CBPRINTF_REDUCED_INTEGRAL
- CONFIG_CBPRINTF_FP_SUPPORT
- CONFIG_CBPRINTF_FP_A_SUPPORT
- CONFIG_CBPRINTF_FP_ALWAYS_A
- CONFIG_CBPRINTF_N_SPECIFIER

CONFIG_CBPRINTF_LIBC_SUBSTS can be used to provide functions that behave like standard libc functions but use the selected cbprintf formatter rather than pulling in another formatter from libc.

In addition CONFIG_CBPRINTF_NANO can be used to revert back to the very space-optimized but limited formatter used for printk() before this capability was added.

4.8.1 Cbprintf Packaging

Typically, strings are formatted synchronously when a function from printf family is called. However, there are cases when it is beneficial that formatting is deferred. In that case, a state (format string and arguments) must be captured. Such state forms a self-contained package which contains format string and arguments. Additionally, package may contain copies of strings which are part of a format string (format string or any %s argument). Package primary content resembles va_list stack frame thus standard formatting functions are used to process a package. Since package contains data which is processed as va_list frame, strict alignment must be maintained. Due to required padding, size of the package depends on alignment. When package is copied, it should be copied to a memory block with the same alignment as origin.

Package can have following variants:

- **Self-contained** - non read-only strings appended to the package. String can be formatted from such package as long as there is access to read-only string locations. Package may contain information where read-only strings are located within the package. That information can be used to convert packet to fully self-contained package.
- **Fully self-contained** - all strings are appended to the package. String can be formatted from such package without any external data.
- **Transient**- only arguments are stored. Package contain information where pointers to non read-only strings are located within the package. Optionally, it may contain read-only string location information. String can be formatted from such package as long as non read-only strings are still valid and read-only strings are accessible. Alternatively, package can be converted to **self-contained** package or **fully self-contained** if information about read-only string locations is present in the package.

Package can be created using two methods:

- runtime - using *cbprintf_package()* or *cbvprintf_package()*. This method scans format string and based on detected format specifiers builds the package.
- static - types of arguments are detected at compile time by the preprocessor and package is created as simple assignments to a provided memory. This method is significantly faster than runtime (more than 15 times) but has following limitations: requires _Generic keyword (C11 feature) to be supported by the compiler and cannot distinguish between %p and %s if char pointer is used. It treats all (unsigned) char pointers as %s thus it will attempt to append string to a package. It can be handled correctly during conversion from **transient** package to **self-contained** package using CBPRINTF_PACKAGE_CONVERT_PTR_CHECK

flag. However, it requires access to the format string and it is not always possible thus it is recommended to cast char pointers used for %p to `void *`. There is a logging warning generated by *cbprintf_package_convert()* called with CBPRINTF_PACKAGE_CONVERT_PTR_CHECK flag when char pointer is used with %p.

Several Kconfig options control behavior of the packaging:

- CONFIG_CBPRINTF_PACKAGE_LONGDOUBLE
- CONFIG_CBPRINTF_STATIC_PACKAGE_CHECK_ALIGNMENT

Cbprintf package conversion

It is possible to convert package to a variant which contains more information, e.g **transient** package can be converted to **self-contained**. Conversion to **fully self-contained** package is possible if CBPRINTF_PACKAGE_ADD_RO_STR_POS flag was used when package was created.

cbprintf_package_copy() is used to calculate space needed for the new package and to copy and convert a package.

Cbprintf package format

Format of the package contains paddings which are platform specific. Package consists of header which contains size of package (excluding appended strings) and number of appended strings. It is followed by the arguments which contains alignment paddings and resembles *va_list* stack frame. It is followed by data associated with character pointer arguments used by the string which are not appended to the string (but may be appended later by cbprinf_package_convert()). Finally, package, optionally, contains appended strings. Each string contains 1 byte header which contains index of the location where address argument is stored. During packaging address is set to null and before string formatting it is updated to point to the current string location within the package. Updating address argument must happen just before string formatting since address changes whenever package is copied.

Header sizeof(void *)	1 byte: Argument list size including header and *fmt* (in 32 bit words)
	1 byte: Number of strings appended to the package
	1 byte: Number of read-only string argument locations
	1 byte: Number of transient string argument locations
	platform specific padding to sizeof(void *)
Arguments	Pointer to *fmt* (or null if *fmt* is appended to the package)
	(optional padding for platform specific alignment)
	argument 0
	(optional padding for platform specific alignment)
	argument 1
	...
String location information (optional)	Indexes of words within the package where read-only strings are located
	Pairs of argument index and argument location index where transient strings are located
Appended strings (optional)	1 byte: Index within the package to the location of associated argument
	Null terminated string
	...

Warning

If CONFIG_MINIMAL_LIBC is selected in combination with CONFIG_CBPRINTF_NANO formatting with C standard library functions like printf or snprintf is limited. Among other things the %n specifier, most format flags, precision control, and floating point are not supported.

Limitations and recommendations

- C11 _Generic support is required by the compiler to use static (fast) packaging.
- It is recommended to cast any character pointer used with %p format specifier to other pointer type (e.g. void *). If format string is not accessible then only static packaging is possible and it will append all detected strings. Character pointer used for %p will be considered as string pointer. Copying from unexpected location can have serious consequences (e.g., memory fault or security violation).

4.8.2 API Reference

group cbprintf_apis

Defines

CBPRINTF_PACKAGE_ALIGNMENT

Required alignment of the buffer used for packaging.

CBPRINTF_MUST_RUNTIME_PACKAGE(flags, ...)

Determine if string must be packaged in run time.

Static packaging can be applied if size of the package can be determined at compile time. In general, package size can be determined at compile time if there are no string arguments which might be copied into package body if they are considered transient.

Note

By default any char pointers are considered to be pointing at transient strings. This can be narrowed down to non const pointers by using CBPRINTF_PACKAGE_CONST_CHAR_RO.

Parameters

- ... – String with arguments.
- flags – option flags. See Package flags.

Return values

- 1 – if string must be packaged in run time.
- 0 – string can be statically packaged.

CBPRINTF_STATIC_PACKAGE(packaged, inlen, outlen, align_offset, flags, ...)

Statically package string.

Build string package from formatted string. It assumes that formatted string is in the read only memory.

If _Generic is not supported then runtime packaging is performed.

Parameters

- **packaged** – pointer to where the packaged data can be stored. Pass a null pointer to skip packaging but still calculate the total space required. The data stored here is relocatable, that is it can be moved to another contiguous block of memory. It must be aligned to the size of the longest argument. It is recommended to use CBPRINTF_PACKAGE_ALIGNMENT for alignment.
- **inlen** – set to the number of bytes available at packaged. If packaged is NULL the value is ignored.
- **outlen** – variable updated to the number of bytes required to completely store the packed information. If input buffer was too small it is set to -ENOSPC.
- **align_offset** – input buffer alignment offset in bytes. Where offset 0 means that buffer is aligned to CBPRINTF_PACKAGE_ALIGNMENT. Xtensa requires that packaged is aligned to CBPRINTF_PACKAGE_ALIGNMENT so it must be multiply of CBPRINTF_PACKAGE_ALIGNMENT or 0.
- **flags** – option flags. See Package flags.
- **...** – formatted string with arguments. Format string must be constant.

Typedefs

typedef int (*cbprintf_cb)()

Signature for a cbprintf callback function.

This function expects two parameters:

- c a character to output. The output behavior should be as if this was cast to an unsigned char.
- ctx a pointer to an object that provides context for the output operation.

The declaration does not specify the parameter types. This allows a function like fputc to be used without requiring all context pointers to be to a FILE object.

Return

the value of c cast to an unsigned char then back to int, or a negative error code that will be returned from *cbprintf()*.

typedef int (*cbprintf_cb_local)(int c, void *ctx)

typedef int (*cbprintf_convert_cb)(const void *buf, size_t len, void *ctx)

Signature for a cbprintf multibyte callback function.

return Amount of copied data or negative error code.

Param buf
data.

Param len
data length.

Param ctx
a pointer to an object that provides context for the operation.

typedef int (*`cbvprintf_external_formatter_func`)(*cbprintf_cb* out, void *ctx, const char *fmt, va_list ap)

Signature for a external formatter function identical to cbvprintf.

This function expects the following parameters:

Param out
the function used to emit each generated character.

Param ctx
a pointer to an object that provides context for the external formatter.

Param fmt
a standard ISO C format string with characters and conversion specifications.

Param ap
captured stack arguments corresponding to the conversion specifications found within `fmt`.

Return
vprintf like return values: the number of characters printed, or a negative error value returned from external formatter.

Functions

int `cbprintf_package`(void *packaged, size_t len, uint32_t flags, const char *format, ...)

Capture state required to output formatted data later.

Like *cbprintf()* but instead of processing the arguments and emitting the formatted results immediately all arguments are captured so this can be done in a different context, e.g. when the output function can block.

In addition to the values extracted from arguments this will ensure that copies are made of the necessary portions of any string parameters that are not confirmed to be stored in read-only memory (hence assumed to be safe to refer to directly later).

Parameters

- `packaged` – pointer to where the packaged data can be stored. Pass a null pointer to store nothing but still calculate the total space required. The data stored here is relocatable, that is it can be moved to another contiguous block of memory. However, under condition that alignment is maintained. It must be aligned to at least the size of a pointer.
- `len` – this must be set to the number of bytes available at `packaged` if it is not null. If `packaged` is null then it indicates hypothetical buffer alignment offset in bytes compared to CBPRINTF_PACKAGE_ALIGNMENT alignment. Buffer alignment offset impacts returned size of the package. Xtensa requires that buffer is always aligned to CBPRINTF_PACKAGE_ALIGNMENT so it must be multiply of CBPRINTF_PACKAGE_ALIGNMENT or 0 when `packaged` is null.
- `flags` – option flags. See Package flags.

- **format** – a standard ISO C format string with characters and conversion specifications.
- **...** – arguments corresponding to the conversion specifications found within format.

Return values

- **nonegative** – the number of bytes successfully stored at packaged. This will not exceed len.
- **-EINVAL** – if format is not acceptable
- **-EFAULT** – if packaged alignment is not acceptable
- **-ENOSPC** – if packaged was not null and the space required to store exceed len.

int **cbvprintf_package**(void *packaged, size_t len, uint32_t flags, const char *format, va_list ap)

Capture state required to output formatted data later.

Like *cbprintf()* but instead of processing the arguments and emitting the formatted results immediately all arguments are captured so this can be done in a different context, e.g. when the output function can block.

In addition to the values extracted from arguments this will ensure that copies are made of the necessary portions of any string parameters that are not confirmed to be stored in read-only memory (hence assumed to be safe to refer to directly later).

Parameters

- **packaged** – pointer to where the packaged data can be stored. Pass a null pointer to store nothing but still calculate the total space required. The data stored here is relocatable, that is it can be moved to another contiguous block of memory. The pointer must be aligned to a multiple of the largest element in the argument list.
- **len** – this must be set to the number of bytes available at packaged. Ignored if packaged is NULL.
- **flags** – option flags. See Package flags.
- **format** – a standard ISO C format string with characters and conversion specifications.
- **ap** – captured stack arguments corresponding to the conversion specifications found within format.

Return values

- **nonegative** – the number of bytes successfully stored at packaged. This will not exceed len.
- **-EINVAL** – if format is not acceptable
- **-ENOSPC** – if packaged was not null and the space required to store exceed len.

int **cbprintf_package_convert**(void *in_packaged, size_t in_len, *cbprintf_convert_cb* cb, void *ctx, uint32_t flags, uint16_t *strl, size_t strl_len)

Convert a package.

Converting may include appending strings used in the package to the package body. If input package was created with CBPRINTF_PACKAGE_ADD_RO_STR_POS or CBPRINTF_PACKAGE_ADD_RW_STR_POS, it contains information where strings are located within the package. This information can be used to copy strings during the conversion.

cb is called with portions of the output package. At the end of the conversion cb is called with null buffer.

Parameters

- **in_packaged** – Input package.
- **in_len** – Input package length. If 0 package length will be retrieved from the in_packaged
- **cb** – callback called with portions of the converted package. If null only length of the output package is calculated.
- **ctx** – Context provided to the cb.
- **flags** – Flags. See Package convert flags.
- **strl** – **[inout]** if packaged is null, it is a pointer to the array where strl_len first string lengths will is stored. If packaged is not null, it contains lengths of first strl_len strings. It can be used to optimize copying so that string length is calculated only once (at length calculation phase when packaged is null.)
- **strl_len** – Number of elements in strl array.

Return values

- **Positive** – output package size.
- **-ENOSPC** – if packaged was not null and the space required to store exceed len.

static inline int **cbprintf_package_copy**(void *in_packaged, size_t in_len, void *packaged, size_t len, uint32_t flags, uint16_t *strl, size_t strl_len)

Copy package with optional appending of strings.

cbprintf_package_convert is used to convert and store converted package in the new location.

Parameters

- **in_packaged** – Input package.
- **in_len** – Input package length. If 0 package length will be retrieved from the in_packaged
- **packaged** – **[out]** Output package. If null only length of the output package is calculated.
- **len** – Available space in the location pointed by packaged. Not used when packaged is null.
- **flags** – Flags. See Package convert flags.
- **strl** – **[inout]** if packaged is null, it is a pointer to the array where strl_len first string lengths will is stored. If packaged is not null, it contains lengths of first strl_len strings. It can be used to optimize copying so that string length is calculated only once (at length calculation phase when packaged is null.)
- **strl_len** – Number of elements in strl array.

Return values

- **Positive** – Output package size.
- **-ENOSPC** – if packaged was not null and the space required to store exceed len.

static inline int **cbprintf_fsc_package**(void *in_packaged, size_t in_len, void *packaged, size_t len)

Convert package to fully self-contained (fsc) package.

Package may not be self contain since strings by default are stored by address. Package may be partially self-contained when transient (not read only) strings are appended to the package. Such package can be decoded only when there is an access to read-only strings.

Fully self-contained has (fsc) contains all strings used in the package. A package can be converted to fsc package if it was create with CBPRINTF_PACKAGE_ADD_RO_STR_POS flag. Such package will contain necessary data to find read only strings in the package and copy them into the package body.

Parameters

- **in_packaged** – pointer to original package created with CBPRINTF_PACKAGE_ADD_RO_STR_POS.
- **in_len** – `in_packaged` length.
- **packaged** – pointer to location where fully self-contained version of the input package will be written. Pass a null pointer to calculate space required.
- **len** – must be set to the number of bytes available at `packaged`. Not used if `packaged` is null.

Return values

- **nonegative** – the number of bytes successfully stored at `packaged`. This will not exceed `len`. If `packaged` is null, calculated length.
- **-ENOSPC** – if `packaged` was not null and the space required to store exceed `len`.
- **-EINVAL** – if `in_packaged` is null.

int **cbpprintf_external**(*cbprintf_cb* out, *cbvprintf_external_formatter_func* formatter, void *ctx, void *packaged)

Generate the output for a previously captured format operation using an external formatter.

> **Note**
>
> Memory indicated by `packaged` will be modified in a non-destructive way, meaning that it could still be reused with this function again.

Parameters

- **out** – the function used to emit each generated character.
- **formatter** – external formatter function.
- **ctx** – a pointer to an object that provides context for the external formatter.
- **packaged** – the data required to generate the formatted output, as captured by *cbprintf_package()* or *cbvprintf_package()*. The alignment requirement on this data is the same as when it was initially created.

Returns

printf like return values: the number of characters printed, or a negative error value returned from external formatter.

int **cbprintf**(*cbprintf_cb* out, void *ctx, const char *format, ...)

*printf-like output through a callback.

This is essentially printf() except the output is generated character-by-character using the provided `out` function. This allows formatting text of unbounded length without incurring the cost of a temporary buffer.

All formatting specifiers of C99 are recognized, and most are supported if the functionality is enabled.

> **Note**
>
> The functionality of this function is significantly reduced when `CONFIG_CBPRINTF_NANO` is selected.

Parameters

- **out** – the function used to emit each generated character.
- **ctx** – context provided when invoking out
- **format** – a standard ISO C format string with characters and conversion specifications.
- **...** – arguments corresponding to the conversion specifications found within `format`.

Returns

the number of characters printed, or a negative error value returned from invoking `out`.

static inline int **cbvprintf**(*cbprintf_cb* out, void *ctx, const char *format, va_list ap)

varargs-aware *printf-like output through a callback.

This is essentially vsprintf() except the output is generated character-by-character using the provided `out` function. This allows formatting text of unbounded length without incurring the cost of a temporary buffer.

> **Note**
>
> This function is available only when `CONFIG_CBPRINTF_LIBC_SUBSTS` is selected.

> **Note**
>
> The functionality of this function is significantly reduced when `CONFIG_CBPRINTF_NANO` is selected.

Parameters

- **out** – the function used to emit each generated character.
- **ctx** – context provided when invoking out
- **format** – a standard ISO C format string with characters and conversion specifications.
- **ap** – a reference to the values to be converted.

Returns
the number of characters generated, or a negative error value returned from invoking `out`.

static inline int **cbvprintf_tagged_args**(*cbprintf_cb* out, void *ctx, const char *format, va_list ap)

varargs-aware *printf-like output through a callback with tagged arguments.

This is essentially vsprintf() except the output is generated character-by-character using the provided `out` function. This allows formatting text of unbounded length without incurring the cost of a temporary buffer.

Note that the argument list `ap` are tagged.

Note

This function is available only when `CONFIG_CBPRINTF_LIBC_SUBSTS` is selected.

Note

The functionality of this function is significantly reduced when `CONFIG_CBPRINTF_NANO` is selected.

Parameters

- **out** – the function used to emit each generated character.
- **ctx** – context provided when invoking out
- **format** – a standard ISO C format string with characters and conversion specifications.
- **ap** – a reference to the values to be converted.

Returns
the number of characters generated, or a negative error value returned from invoking `out`.

static inline int **cbpprintf**(*cbprintf_cb* out, void *ctx, void *packaged)

Generate the output for a previously captured format operation.

Note

Memory indicated by `packaged` will be modified in a non-destructive way, meaning that it could still be reused with this function again.

Parameters

- **out** – the function used to emit each generated character.
- **ctx** – context provided when invoking out
- **packaged** – the data required to generate the formatted output, as captured by *cbprintf_package()* or *cbvprintf_package()*. The alignment requirement on this data is the same as when it was initially created.

Returns
the number of characters printed, or a negative error value returned from invoking `out`.

int **fprintfcb**(FILE *stream, const char *format, ...)
fprintf using Zephyrs cbprintf infrastructure.

return The number of characters printed.

> **Note**
>
> This function is available only when CONFIG_CBPRINTF_LIBC_SUBSTS is selected.

> **Note**
>
> The functionality of this function is significantly reduced when CONFIG_CBPRINTF_NANO is selected.

Parameters

- **stream** – the stream to which the output should be written.
- **format** – a standard ISO C format string with characters and conversion specifications.
- ... – arguments corresponding to the conversion specifications found within format.

int **vfprintfcb**(FILE *stream, const char *format, va_list ap)
vfprintf using Zephyrs cbprintf infrastructure.

> **Note**
>
> This function is available only when CONFIG_CBPRINTF_LIBC_SUBSTS is selected.

> **Note**
>
> The functionality of this function is significantly reduced when CONFIG_CBPRINTF_NANO is selected.

Parameters

- **stream** – the stream to which the output should be written.
- **format** – a standard ISO C format string with characters and conversion specifications.
- **ap** – a reference to the values to be converted.

Returns
The number of characters printed.

int **printfcb**(const char *format, ...)
printf using Zephyrs cbprintf infrastructure.

> **Note**
>
> This function is available only when CONFIG_CBPRINTF_LIBC_SUBSTS is selected.

Note

The functionality of this function is significantly reduced when CONFIG_CBPRINTF_NANO is selected.

Parameters

- **format** – a standard ISO C format string with characters and conversion specifications.
- **...** – arguments corresponding to the conversion specifications found within format.

Returns
The number of characters printed.

int **vprintfcb**(const char *format, va_list ap)
vprintf using Zephyrs cbprintf infrastructure.

Note

This function is available only when CONFIG_CBPRINTF_LIBC_SUBSTS is selected.

Note

The functionality of this function is significantly reduced when CONFIG_CBPRINTF_NANO is selected.

Parameters

- **format** – a standard ISO C format string with characters and conversion specifications.
- **ap** – a reference to the values to be converted.

Returns
The number of characters printed.

int **snprintfcb**(char *str, size_t size, const char *format, ...)
snprintf using Zephyrs cbprintf infrastructure.

Note

This function is available only when CONFIG_CBPRINTF_LIBC_SUBSTS is selected.

Note

The functionality of this function is significantly reduced when CONFIG_CBPRINTF_NANO is selected.

Parameters

- **str** – where the formatted content should be written

- **size** – maximum number of chaacters for the formatted output, including the terminating null byte.
- **format** – a standard ISO C format string with characters and conversion specifications.
- ... – arguments corresponding to the conversion specifications found within format.

Returns
The number of characters that would have been written to str, excluding the terminating null byte. This is greater than the number actually written if size is too small.

int **vsnprintfcb**(char *str, size_t size, const char *format, va_list ap)
vsnprintf using Zephyrs cbprintf infrastructure.

> **Note**
> This function is available only when CONFIG_CBPRINTF_LIBC_SUBSTS is selected.

> **Note**
> The functionality of this function is significantly reduced when CONFIG_CBPRINTF_NANO is selected.

Parameters

- **str** – where the formatted content should be written
- **size** – maximum number of chaacters for the formatted output, including the terminating null byte.
- **format** – a standard ISO C format string with characters and conversion specifications.
- **ap** – a reference to the values to be converted.

Returns
The number of characters that would have been written to str, excluding the terminating null byte. This is greater than the number actually written if size is too small.

4.9 Input

The input subsystem provides an API for dispatching input events from input devices to the application.

4.9.1 Input Events

The subsystem is built around the *input_event* structure. An input event represents a change in an individual event entity, for example the state of a single button, or a movement in a single axis.

The *input_event* structure describes the specific event, and includes a synchronization bit to indicate that the device reached a stable state, for example when the events corresponding to multiple axes of a multi-axis device have been reported.

4.9.2 Input Devices

An input device can report input events directly using *input_report()* or any related function; for example buttons or other on-off input entities would use *input_report_key()*.

Complex devices may use a combination of multiple events, and set the sync bit once the output is stable.

The input_report* functions take a *device* pointer, which is used to indicate which device reported the event and can be used by subscribers to only receive events from a specific device. If there's no actual device associated with the event, it can be set to NULL, in which case only subscribers with no device filter will receive the event.

4.9.3 Application API

An application can register a callback using the *INPUT_CALLBACK_DEFINE* macro. If a device node is specified, the callback is only invoked for events from the specific device, otherwise the callback will receive all the events in the system. This is the only type of filtering supported, any more complex filtering logic has to be implemented in the callback itself.

The subsystem can operate synchronously or by using an event queue, depending on the CONFIG_INPUT_MODE option. If the input thread is used, all the events are added to a queue and executed in a common input thread. If the thread is not used, the callback are invoked directly in the input driver context.

The synchronous mode can be used in a simple application to keep a minimal footprint, or in a complex application with an existing event model, where the callback is just a wrapper to pipe back the event in a more complex application specific event system.

4.9.4 HID code mapping

A common use case for input devices is to use them to generate HID reports. For this purpose, the *input_to_hid_code()* and *input_to_hid_modifier()* functions can be used to map input codes to HID codes and modifiers.

4.9.5 Kscan Compatibility

Input devices generating X/Y/Touch events can be used in existing applications based on the *Keyboard Scan* API by enabling both CONFIG_INPUT and CONFIG_KSCAN, defining a zephyr,kscan-input node as a child node of the corresponding input device and pointing the zephyr,keyboard-scan chosen node to the compatibility device node, for example:

```
chosen {
    zephyr,keyboard-scan = &kscan_input;
};

ft5336@38 {
    ...
    kscan_input: kscan-input {
        compatible = "zephyr,kscan-input";
    };
};
```

4.9.6 Driver Documentation

GPIO Keyboard Matrix

The `gpio-kbd-matrix` driver supports a large variety of keyboard matrix hardware configurations and has numerous options to change its behavior. This is an overview of some common setups and how they can be supported by the driver.

The conventional configuration for all of these is that the driver reads on the row GPIOs (inputs) and selects on the columns GPIOs (output).

Base use case, no isolation diodes, interrupt capable GPIOs This is the common configuration found on consumer keyboards with membrane switches and flexible circuit boards, no isolation diodes, requires ghosting detection (which is enabled by default).

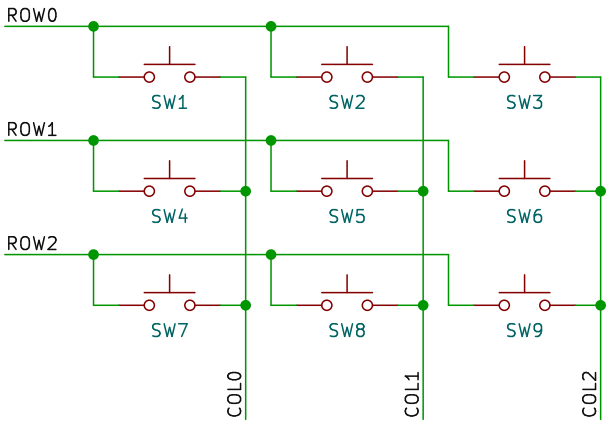


Fig. 1: A 3x3 matrix, no diodes

The system must support GPIO interrupts, and the interrupt can be enabled on all row GPIOs at the same time.

```
kbd-matrix {
     compatible = "gpio-kbd-matrix";
     row-gpios = <&gpio0 0 (GPIO_PULL_UP | GPIO_ACTIVE_LOW)>,
                 <&gpio0 1 (GPIO_PULL_UP | GPIO_ACTIVE_LOW)>,
                 <&gpio0 2 (GPIO_PULL_UP | GPIO_ACTIVE_LOW)>;
     col-gpios = <&gpio0 3 GPIO_ACTIVE_LOW>,
                 <&gpio0 4 GPIO_ACTIVE_LOW>,
                 <&gpio0 5 GPIO_ACTIVE_LOW>;
};
```

In this configuration the matrix scanning library enters idle mode once all keys are released, and the keyboard matrix thread only wakes up when a key has been pressed.

GPIOs for columns that are not currently selected are configured in high impedance mode. This means that the row state may need some time to settle to avoid misreading the key state from a column to the following one. The settle time can be tweaked by changing the `settle-time-us` property.

Isolation diodes If the matrix has isolation diodes for every key, then it's possible to:

- disable ghosting detection, allowing any key combination to be detected
- configuring the driver to drive unselected columns GPIO to inactive state rather than high impedance, this allows to reduce the settle time (potentially down to 0), and use the more efficient port wide GPIO read APIs (happens automatically if the GPIO pins are sequential)

Matrixes with diodes going from rows to columns must use pull-ups on rows and active low columns.

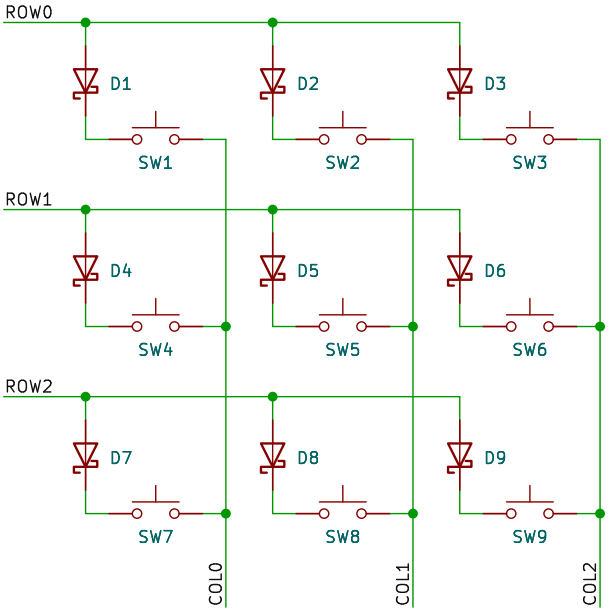


Fig. 2: A 3x3 matrix with row to column isolation diodes.

```
kbd-matrix {
     compatible = "gpio-kbd-matrix";
     row-gpios = <&gpio0 0 (GPIO_PULL_UP | GPIO_ACTIVE_LOW)>,
                 <&gpio0 1 (GPIO_PULL_UP | GPIO_ACTIVE_LOW)>,
                 <&gpio0 2 (GPIO_PULL_UP | GPIO_ACTIVE_LOW)>;
     col-gpios = <&gpio0 3 GPIO_ACTIVE_LOW>,
                 <&gpio0 4 GPIO_ACTIVE_LOW>,
                 <&gpio0 5 GPIO_ACTIVE_LOW>;
     col-drive-inactive;
     settle-time-us = <0>;
     no-ghostkey-check;
};
```

Matrixes with diodes going from columns to rows must use pull-downs on rows and active high columns.

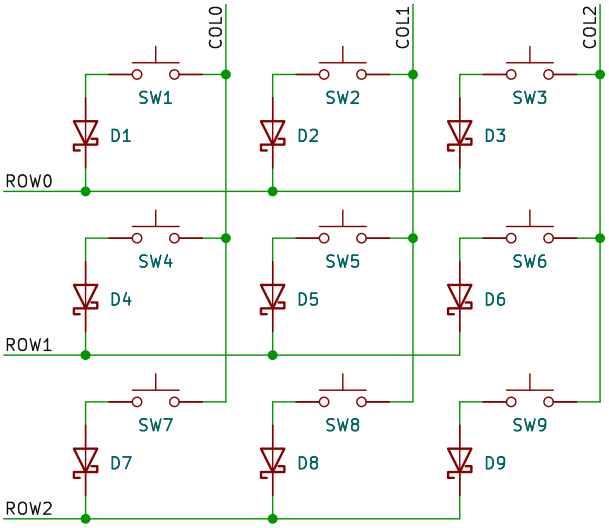


Fig. 3: A 3x3 matrix with column to row isolation diodes.

```
kbd-matrix {
     compatible = "gpio-kbd-matrix";
     row-gpios = <&gpio0 0 (GPIO_PULL_DOWN | GPIO_ACTIVE_HIGH)>,
                 <&gpio0 1 (GPIO_PULL_DOWN | GPIO_ACTIVE_HIGH)>,
                 <&gpio0 2 (GPIO_PULL_DOWN | GPIO_ACTIVE_HIGH)>;
     col-gpios = <&gpio0 3 GPIO_ACTIVE_HIGH>,
                 <&gpio0 4 GPIO_ACTIVE_HIGH>,
                 <&gpio0 5 GPIO_ACTIVE_HIGH>;
     col-drive-inactive;
     settle-time-us = <0>;
     no-ghostkey-check;
};
```

GPIO with no interrupt support Some GPIO controllers have limitations on GPIO interrupts, and may not support enabling interrupts on all row GPIOs at the same time.

In this case, the driver can be configured to not use interrupt at all, and instead idle by selecting all columns and keep polling on the row GPIOs, which is a single GPIO API operation if the pins are sequential.

This configuration can be enabled by setting the `idle-mode` property to `poll`:

```
kbd-matrix {
     compatible = "gpio-kbd-matrix";
     ...
     idle-mode = "poll";
};
```

GPIO multiplexer In more extreme cases, such as if the columns are using a multiplexer and it's impossible to select all of them at the same time, the driver can be configured to scan continuously.

This can be done by setting `idle-mode` to `scan` and `poll-timeout-ms` to `0`.

```
kbd-matrix {
     compatible = "gpio-kbd-matrix";
     ...
     poll-timeout-ms = <0>;
     idle-mode = "scan";
};
```

Row and column GPIO selection If the row GPIOs are sequential and on the same gpio controller, the driver automatically switches API to read from the whole GPIO port rather than the individual pins. This is particularly useful if the GPIOs are not memory mapped, for example on an I2C or SPI port expander, as this significantly reduces the number of transactions on the corresponding bus.

The same is true for column GPIOs, but only if the matrix is configured for `col-drive-inactive`, so that is only usable for matrixes with isolation diodes.

16-bit row support The driver uses an 8-bit datatype to store the row state by default, which limits the matrix row size to 8. This can be increased to 16 by enabling the `CONFIG_INPUT_KBD_MATRIX_16_BIT_ROW` option.

Actual key mask configuration If the key matrix is not complete, a map of the keys that are actually populated can be specified using the *actual-key-mask* property. This allows the matrix state to be filtered to remove keys that are not present before ghosting detection, potentially allowing key combinations that would otherwise be blocked by it.

For example for a 3x3 matrix missing a key:

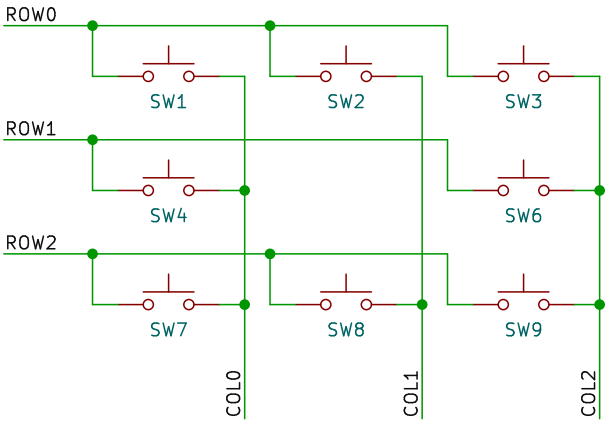


Fig. 4: A 3x3 matrix missing a key.

```
kbd-matrix {
     compatible = "gpio-kbd-matrix";
     ...
     actual-key-mask = <0x07 0x05 0x07>;
};
```

This would allow, for example, to detect pressing `SW1`, `SW2` and `SW4` at the same time without triggering anti ghosting.

The actual key mask can be changed at runtime by enabling `CONFIG_INPUT_KBD_ACTUAL_KEY_MASK_DYNAMIC` and the using the *input_kbd_matrix_actual_key_mask_set()* API.

Keymap configuration Keyboard matrix devices report a series of x/y/touch events. These can be mapped to normal key events using the `input-keymap` driver.

For example, the following would setup a `keymap` device that take the x/y/touch events as an input and generate corresponding key events as an output:

```
kbd {
    ...
    keymap {
        compatible = "input-keymap";
        keymap = <
            MATRIX_KEY(0, 0, INPUT_KEY_1)
            MATRIX_KEY(0, 1, INPUT_KEY_2)
            MATRIX_KEY(0, 2, INPUT_KEY_3)
            MATRIX_KEY(1, 0, INPUT_KEY_4)
            MATRIX_KEY(1, 1, INPUT_KEY_5)
            MATRIX_KEY(1, 2, INPUT_KEY_6)
            MATRIX_KEY(2, 0, INPUT_KEY_7)
            MATRIX_KEY(2, 1, INPUT_KEY_8)
            MATRIX_KEY(2, 2, INPUT_KEY_9)
        >;
        row-size = <3>;
        col-size = <3>;
    };
};
```

Keyboard matrix shell commands The shell command `kbd_matrix_state_dump` can be used to test the functionality of any keyboard matrix driver implemented using the keyboard matrix library. Once enabled it logs the state of the matrix every time it changes, and once disabled it prints an or-mask of any key that has been detected, which can be used to set the `actual-key-mask` property.

The command can be enabled using the `CONFIG_INPUT_SHELL_KBD_MATRIX_STATE`.

Example usage:

```
uart:~$ device list
devices:
- kbd-matrix (READY)
uart:~$ input kbd_matrix_state_dump kbd-matrix
Keyboard state logging enabled for kbd-matrix
[00:01:41.678,466] <inf> input: kbd-matrix state [01 -- -- --] (1)
[00:01:41.784,912] <inf> input: kbd-matrix state [-- -- -- --] (0)
...
press more buttons
...
uart:~$ input kbd_matrix_state_dump off
Keyboard state logging disabled
[00:01:47.967,651] <inf> input: kbd-matrix key-mask [07 05 07 --] (8)
```

Keyboard matrix library The GPIO keyboard matrix driver is based on a generic keyboard matrix library, which implements the core functionalities such as scanning delays, debouncing, idle mode etc. This can be reused to implement other keyboard matrix drivers, potentially application specific.

group `input_kbd_matrix`

Keyboard Matrix API.

Defines

`INPUT_KBD_MATRIX_COLUMN_DRIVE_NONE`

Special drive_column argument for not driving any column.

`INPUT_KBD_MATRIX_COLUMN_DRIVE_ALL`

Special drive_column argument for driving all the columns.

`INPUT_KBD_MATRIX_SCAN_OCURRENCES`

Number of tracked scan cycles.

`PRIkbdrow`

`INPUT_KBD_ACTUAL_KEY_MASK_CONST`

`INPUT_KBD_MATRIX_ROW_BITS`

Maximum number of rows.

`INPUT_KBD_MATRIX_DATA_NAME`(node_id, name)

INPUT_KBD_MATRIX_DT_DEFINE_ROW_COL(node_id, _row_size, _col_size)

Defines the common keyboard matrix support data from devicetree, specify row and col count.

INPUT_KBD_MATRIX_DT_DEFINE(node_id)

Defines the common keyboard matrix support data from devicetree.

INPUT_KBD_MATRIX_DT_INST_DEFINE_ROW_COL(inst, row_size, col_size)

Defines the common keyboard matrix support data from devicetree instance, specify row and col count.

Parameters

- `inst` – Instance.
- `row_size` – The matrix row count.
- `col_size` – The matrix column count.

INPUT_KBD_MATRIX_DT_INST_DEFINE(inst)

Defines the common keyboard matrix support data from devicetree instance.

Parameters

- `inst` – Instance.

INPUT_KBD_MATRIX_DT_COMMON_CONFIG_INIT_ROW_COL(node_id, _api, _row_size, _col_size)

Initialize common keyboard matrix config from devicetree, specify row and col count.

Parameters

- `node_id` – The devicetree node identifier.
- `_api` – Pointer to a *input_kbd_matrix_api* structure.
- `_row_size` – The matrix row count.
- `_col_size` – The matrix column count.

INPUT_KBD_MATRIX_DT_COMMON_CONFIG_INIT(node_id, api)

Initialize common keyboard matrix config from devicetree.

Parameters

- `node_id` – The devicetree node identifier.
- `api` – Pointer to a *input_kbd_matrix_api* structure.

INPUT_KBD_MATRIX_DT_INST_COMMON_CONFIG_INIT_ROW_COL(inst, api, row_size, col_size)

Initialize common keyboard matrix config from devicetree instance, specify row and col count.

Parameters

- `inst` – Instance.
- `api` – Pointer to a *input_kbd_matrix_api* structure.
- `row_size` – The matrix row count.
- `col_size` – The matrix column count.

INPUT_KBD_MATRIX_DT_INST_COMMON_CONFIG_INIT(inst, api)

Initialize common keyboard matrix config from devicetree instance.

Parameters

- `inst` – Instance.
- `api` – Pointer to a *input_kbd_matrix_api* structure.

INPUT_KBD_STRUCT_CHECK(config, data)

Validate the offset of the common data structures.

Parameters

- **config** – Name of the config structure.
- **data** – Name of the data structure.

Typedefs

typedef uint8_t **kbd_row_t**

Row entry data type.

Functions

int **input_kbd_matrix_actual_key_mask_set**(const struct *device* *dev, uint8_t row, uint8_t col, bool enabled)

Enables or disables a specific row, column combination in the actual key mask.

This allows enabling or disabling specific row, column combination in the actual key mask in runtime. It can be useful if some of the keys are not present in some configuration, and the specific configuration is determined in runtime. Requires CONFIG_INPUT_KBD_ACTUAL_KEY_MASK_DYNAMIC to be enabled.

Parameters

- **dev** – Pointer to the keyboard matrix device.
- **row** – The matrix row to enable or disable.
- **col** – The matrix column to enable or disable.
- **enabled** – Whether the specified row, col has to be enabled or disabled.

Return values

- **0** – If the change is successful.
- **-errno** – Negative errno if row or col are out of range for the device.

void **input_kbd_matrix_poll_start**(const struct *device* *dev)

Start scanning the keyboard matrix.

Starts the keyboard matrix scanning cycle, this should be called in reaction of a press event, after the device has been put in detect mode.

Parameters

- **dev** – Keyboard matrix device instance.

void **input_kbd_matrix_drive_column_hook**(const struct *device* *dev, int col)

Drive column hook.

This can be implemented by the application to handle column selection quirks. Called after the driver specific drive_column function. Requires CONFIG_INPUT_KBD_DRIVE_COLUMN_HOOK to be enabled.

Parameters

- **dev** – Keyboard matrix device instance.
- **col** – The column to drive, or *INPUT_KBD_MATRIX_COLUMN_DRIVE_NONE* or *INPUT_KBD_MATRIX_COLUMN_DRIVE_ALL*.

int `input_kbd_matrix_common_init`(const struct *device* *dev)

Common function to initialize a keyboard matrix device at init time.

This function must be called at the end of the device init function.

Parameters

- `dev` – Keyboard matrix device instance.

Return values

- `0` – If initialized successfully.
- `-errno` – Negative errno in case of failure.

struct `input_kbd_matrix_api`

#include <*input_kbd_matrix.h*> Keyboard matrix internal APIs.

Public Members

void (*`drive_column`)(const struct *device* *dev, int col)

Request to drive a specific column.

Request to drive a specific matrix column, or none, or all.

Param dev
Pointer to the keyboard matrix device.

Param col
The column to drive, or *INPUT_KBD_MATRIX_COLUMN_DRIVE_NONE* or *INPUT_KBD_MATRIX_COLUMN_DRIVE_ALL*.

kbd_row_t (*`read_row`)(const struct *device* *dev)

Read the matrix row.

Param dev
Pointer to the keyboard matrix device.

void (*`set_detect_mode`)(const struct *device* *dev, bool enabled)

Request to put the matrix in detection mode.

Request to put the driver in detection mode, this is called after a request to drive all the column and typically involves reenabling interrupts row pin changes.

Param dev
Pointer to the keyboard matrix device.

Param enable
Whether detection mode has to be enabled or disabled.

struct `input_kbd_matrix_common_config`

#include <*input_kbd_matrix.h*> Common keyboard matrix config.

This structure **must** be placed first in the driver's config structure.

struct `input_kbd_matrix_common_data`

#include <*input_kbd_matrix.h*> Common keyboard matrix data.

This structure **must** be placed first in the driver's data structure.

4.9.7 API Reference

Related code samples

LVGL basic sample
Display a ”Hello World” and react to user input using LVGL.

USB HID keyboard
Implement a basic HID keyboard device.

USB HID mouse
Implement a basic HID mouse device.

group `input_interface`

Input Interface.

Since
3.4

Version
0.1.0

Defines

`INPUT_CALLBACK_DEFINE`(_dev, _callback)

Register a callback structure for input events.

The `_dev` field can be used to only invoke callback for events generated by a specific device. Setting dev to NULL causes callback to be invoked for every event.

Parameters

- `_dev` – *device* pointer or NULL.
- `_callback` – The callback function.

Functions

int `input_report`(const struct *device* *dev, uint8_t type, uint16_t code, int32_t value, bool sync, *k_timeout_t* timeout)

Report a new input event.

This causes all the callbacks for the specified device to be executed, either synchronously or through the input thread if utilized.

Parameters

- `dev` – Device generating the event or NULL.
- `type` – Event type (see *INPUT_EV_CODES*).
- `code` – Event code (see *INPUT_KEY_CODES*, *INPUT_BTN_CODES*, *INPUT_ABS_CODES*, *INPUT_REL_CODES*, *INPUT_MSC_CODES*).
- `value` – Event value.
- `sync` – Set the synchronization bit for the event.

- **timeout** – Timeout for reporting the event, ignored if CONFIG_INPUT_MODE_SYNCHRONOUS is used.

Return values

- **0** – if the message has been processed.
- **negative** – if CONFIG_INPUT_MODE_THREAD is enabled and the message failed to be enqueued.

static inline int **input_report_key**(const struct *device* *dev, uint16_t code, int32_t value, bool sync, *k_timeout_t* timeout)

Report a new *INPUT_EV_KEY* input event, note that value is converted to either 0 or 1.

See also

input_report() for more details.

static inline int **input_report_rel**(const struct *device* *dev, uint16_t code, int32_t value, bool sync, *k_timeout_t* timeout)

Report a new *INPUT_EV_REL* input event.

See also

input_report() for more details.

static inline int **input_report_abs**(const struct *device* *dev, uint16_t code, int32_t value, bool sync, *k_timeout_t* timeout)

Report a new *INPUT_EV_ABS* input event.

See also

input_report() for more details.

bool **input_queue_empty**(void)

Returns true if the input queue is empty.

This can be used to batch input event processing until the whole queue has been emptied. Always returns true if CONFIG_INPUT_MODE_SYNCHRONOUS is enabled.

int16_t **input_to_hid_code**(uint16_t input_code)

Convert an input code to HID code.

Takes an input code as input and returns the corresponding HID code as output. The return value is -1 if the code is not found, if found it can safely be casted to a uint8_t type.

Parameters

- **input_code** – Event code (see *INPUT_KEY_CODES*).

Return values

- **the** – HID code corresponding to the input code.

- `-1` – if there's no HID code for the specified input code.

uint8_t **input_to_hid_modifier**(uint16_t input_code)

Convert an input code to HID modifier.

Takes an input code as input and returns the corresponding HID modifier as output or 0.

Parameters

- `input_code` – Event code (see *INPUT_KEY_CODES*).

Return values

- `the` – HID modifier corresponding to the input code.
- `0` – if there's no HID modifier for the specified input code.

struct **input_event**

#include <*input.h*> Input event structure.

This structure represents a single input event, for example a key or button press for a single button, or an absolute or relative coordinate for a single axis.

Public Members

const struct *device* ***dev**

Device generating the event or NULL.

uint8_t **sync**

Sync flag.

uint8_t **type**

Event type (see *INPUT_EV_CODES*).

uint16_t **code**

Event code (see *INPUT_KEY_CODES*, *INPUT_BTN_CODES*, *INPUT_ABS_CODES*, *INPUT_REL_CODES*, *INPUT_MSC_CODES*).

int32_t **value**

Event value.

struct **input_callback**

#include <*input.h*> Input callback structure.

Public Members

const struct *device* ***dev**

device pointer or NULL.

void (***callback**)(struct *input_event* *evt)

The callback function.

4.9.8 Input Event Definitions

Related code samples

Input dump
Print all input events.

group `input_events`

Input event types.

`INPUT_EV_KEY`
Key event.

`INPUT_EV_REL`
Relative coordinate event.

`INPUT_EV_ABS`
Absolute coordinate event.

`INPUT_EV_MSC`
Miscellaneous event.

`INPUT_EV_VENDOR_START`
Vendor specific event start.

`INPUT_EV_VENDOR_STOP`
Vendor specific event stop.

Input event KEY codes.

`INPUT_KEY_RESERVED`
Reserved, do not use.

`INPUT_KEY_0`
0 Key

`INPUT_KEY_1`
1 Key

`INPUT_KEY_2`
2 Key

INPUT_KEY_3
: 3 Key

INPUT_KEY_4
: 4 Key

INPUT_KEY_5
: 5 Key

INPUT_KEY_6
: 6 Key

INPUT_KEY_7
: 7 Key

INPUT_KEY_8
: 8 Key

INPUT_KEY_9
: 9 Key

INPUT_KEY_A
: A Key.

INPUT_KEY_APOSTROPHE
: Apostrophe Key.

INPUT_KEY_B
: B Key.

INPUT_KEY_BACK
: Back Key.

INPUT_KEY_BACKSLASH
: Backslash Key.

INPUT_KEY_BACKSPACE
: Backspace Key.

INPUT_KEY_BLUETOOTH
: Bluetooth Key.

INPUT_KEY_BRIGHTNESSDOWN
: Brightness Up Key.

INPUT_KEY_BRIGHTNESSUP
: Brightneess Down Key.

INPUT_KEY_C
: C Key.

INPUT_KEY_CAPSLOCK
: Caps Lock Key.

INPUT_KEY_COFFEE
: Screen Saver Key.

INPUT_KEY_COMMA
: Comma Key.

INPUT_KEY_COMPOSE
: Compose Key.

INPUT_KEY_CONNECT
: Connect Key.

INPUT_KEY_D
: D Key.

INPUT_KEY_DELETE
: Delete Key.

INPUT_KEY_DOT
: Dot Key.

INPUT_KEY_DOWN
: Down Key.

INPUT_KEY_E
: E Key.

INPUT_KEY_END
: End Key.

INPUT_KEY_ENTER
: Enter Key.

INPUT_KEY_EQUAL
: Equal Key.

INPUT_KEY_ESC
: Escape Key.

INPUT_KEY_F
: F Key.

`INPUT_KEY_F1`
F1 Key.

`INPUT_KEY_F10`
F10 Key.

`INPUT_KEY_F11`
F11 Key.

`INPUT_KEY_F12`
F12 Key.

`INPUT_KEY_F13`
F13 Key.

`INPUT_KEY_F14`
F14 Key.

`INPUT_KEY_F15`
F15 Key.

`INPUT_KEY_F16`
F16 Key.

`INPUT_KEY_F17`
F17 Key.

`INPUT_KEY_F18`
F18 Key.

`INPUT_KEY_F19`
F19 Key.

`INPUT_KEY_F2`
F2 Key.

`INPUT_KEY_F20`
F20 Key.

`INPUT_KEY_F21`
F21 Key.

`INPUT_KEY_F22`
F22 Key.

`INPUT_KEY_F23`
F23 Key.

INPUT_KEY_F24
: F24 Key.

INPUT_KEY_F3
: F3 Key.

INPUT_KEY_F4
: F4 Key.

INPUT_KEY_F5
: F5 Key.

INPUT_KEY_F6
: F6 Key.

INPUT_KEY_F7
: F7 Key.

INPUT_KEY_F8
: F8 Key.

INPUT_KEY_F9
: F9 Key.

INPUT_KEY_FASTFORWARD
: Fast Forward Key.

INPUT_KEY_FORWARD
: Forward Key.

INPUT_KEY_G
: G Key.

INPUT_KEY_GRAVE
: Grave (backtick) Key.

INPUT_KEY_H
: H Key.

INPUT_KEY_HOME
: Home Key.

INPUT_KEY_I
: I Key.

INPUT_KEY_INSERT
: Insert Key.

INPUT_KEY_J
: J Key.

INPUT_KEY_K
: K Key.

INPUT_KEY_KP0
: Keypad 0 Key.

INPUT_KEY_KP1
: Keypad 1 Key.

INPUT_KEY_KP2
: Keypad 2 Key.

INPUT_KEY_KP3
: Keypad 3 Key.

INPUT_KEY_KP4
: Keypad 4 Key.

INPUT_KEY_KP5
: Keypad 5 Key.

INPUT_KEY_KP6
: Keypad 6 Key.

INPUT_KEY_KP7
: Keypad 7 Key.

INPUT_KEY_KP8
: Keypad 8 Key.

INPUT_KEY_KP9
: Keypad 9 Key.

INPUT_KEY_KPASTERISK
: Keypad Asterisk Key.

INPUT_KEY_KPCOMMA
: Keypad Comma Key.

INPUT_KEY_KPDOT
: Keypad Dot Key.

INPUT_KEY_KPENTER
: Keypad Enter Key.

INPUT_KEY_KPEQUAL
: Keypad Equal Key.

INPUT_KEY_KPMINUS
: Keypad Minus Key.

INPUT_KEY_KPPLUS
: Keypad Plus Key.

INPUT_KEY_KPPLUSMINUS
: Keypad Plus Key.

INPUT_KEY_KPSLASH
: Keypad Slash Key.

INPUT_KEY_L
: L Key.

INPUT_KEY_LEFT
: Left Key.

INPUT_KEY_LEFTALT
: Left Alt Key.

INPUT_KEY_LEFTBRACE
: Left Brace Key.

INPUT_KEY_LEFTCTRL
: Left Ctrl Key.

INPUT_KEY_LEFTMETA
: Left Meta Key.

INPUT_KEY_LEFTSHIFT
: Left Shift Key.

INPUT_KEY_M
: M Key.

INPUT_KEY_MENU
: Menu Key.

INPUT_KEY_MINUS
: Minus Key.

INPUT_KEY_MUTE
: Mute Key.

INPUT_KEY_N
: N Key.

INPUT_KEY_NUMLOCK
: Num Lock Key.

INPUT_KEY_O
: O Key.

INPUT_KEY_P
: P Key.

INPUT_KEY_PAGEDOWN
: Page Down Key.

INPUT_KEY_PAGEUP
: Page UpKey.

INPUT_KEY_PAUSE
: Pause Key.

INPUT_KEY_PLAY
: Play Key.

INPUT_KEY_POWER
: Power Key.

INPUT_KEY_PRINT
: Print Key.

INPUT_KEY_Q
: Q Key.

INPUT_KEY_R
: R Key.

INPUT_KEY_RIGHT
: Right Key.

INPUT_KEY_RIGHTALT
: Right Alt Key.

INPUT_KEY_RIGHTBRACE
: Right Brace Key.

INPUT_KEY_RIGHTCTRL
: Right Ctrl Key.

INPUT_KEY_RIGHTMETA
: Right Meta Key.

INPUT_KEY_RIGHTSHIFT
: Right Shift Key.

INPUT_KEY_S
: S Key.

INPUT_KEY_SCALE
: Scale Key.

INPUT_KEY_SCROLLLOCK
: Scroll Lock Key.

INPUT_KEY_SEMICOLON
: Semicolon Key.

INPUT_KEY_SLASH
: Slash Key.

INPUT_KEY_SLEEP
: System Sleep Key.

INPUT_KEY_SPACE
: Space Key.

INPUT_KEY_SYSRQ
: SysReq Key.

INPUT_KEY_T
: T Key.

INPUT_KEY_TAB
: Tab Key.

INPUT_KEY_U
: U Key.

INPUT_KEY_UP
: Up Key.

INPUT_KEY_UWB
: Ultra-Wideband Key.

INPUT_KEY_V
: V Key.

INPUT_KEY_VOLUMEDOWN
: Volume Down Key.

INPUT_KEY_VOLUMEUP
: Volume Up Key.

INPUT_KEY_W
: W Key.

INPUT_KEY_WAKEUP
: System Wake Up Key.

INPUT_KEY_WLAN
: Wireless LAN Key.

INPUT_KEY_X
: X Key.

INPUT_KEY_Y
: Y Key.

INPUT_KEY_Z
: Z Key.

Input event BTN codes.

INPUT_BTN_0
: 0 button

INPUT_BTN_1
: 1 button

INPUT_BTN_2
: 2 button

INPUT_BTN_3
: 3 button

INPUT_BTN_4
: 4 button

INPUT_BTN_5
: 5 button

INPUT_BTN_6
: 6 button

INPUT_BTN_7
: 7 button

INPUT_BTN_8
: 8 button

INPUT_BTN_9
: 9 button

INPUT_BTN_A
: A button.

INPUT_BTN_B
: B button.

INPUT_BTN_BACK
: Back button.

INPUT_BTN_C
: C button.

INPUT_BTN_DPAD_DOWN
: Directional pad Down.

INPUT_BTN_DPAD_LEFT
: Directional pad Left.

INPUT_BTN_DPAD_RIGHT
: Directional pad Right.

INPUT_BTN_DPAD_UP
: Directional pad Up.

INPUT_BTN_EAST
: East button.

INPUT_BTN_EXTRA
: Extra button.

INPUT_BTN_FORWARD
: Forward button.

INPUT_BTN_GEAR_DOWN
: Gear Up button.

INPUT_BTN_GEAR_UP
: Gear Down button.

`INPUT_BTN_LEFT`
: Left button.

`INPUT_BTN_MIDDLE`
: Middle button.

`INPUT_BTN_MODE`
: Mode button.

`INPUT_BTN_NORTH`
: North button.

`INPUT_BTN_RIGHT`
: Right button.

`INPUT_BTN_SELECT`
: Select button.

`INPUT_BTN_SIDE`
: Side button.

`INPUT_BTN_SOUTH`
: South button.

`INPUT_BTN_START`
: Start button.

`INPUT_BTN_TASK`
: Task button.

`INPUT_BTN_THUMBL`
: Left thumbstick button.

`INPUT_BTN_THUMBR`
: Right thumbstick button.

`INPUT_BTN_TL`
: Left trigger (L1)

`INPUT_BTN_TL2`
: Left trigger 2 (L2)

`INPUT_BTN_TOUCH`
: Touchscreen touch.

`INPUT_BTN_TR`
: Right trigger (R1)

INPUT_BTN_TR2
: Right trigger 2 (R2)

INPUT_BTN_WEST
: West button.

INPUT_BTN_X
: X button.

INPUT_BTN_Y
: Y button.

INPUT_BTN_Z
: Z button.

Input event ABS codes.

INPUT_ABS_BRAKE
: Absolute brake position.

INPUT_ABS_GAS
: Absolute gas position.

INPUT_ABS_MT_SLOT
: Absolute multitouch slot identifier.

INPUT_ABS_RUDDER
: Absolute rudder position.

INPUT_ABS_RX
: Absolute rotation around X axis.

INPUT_ABS_RY
: Absolute rotation around Y axis.

INPUT_ABS_RZ
: Absolute rotation around Z axis.

INPUT_ABS_THROTTLE
: Absolute throttle position.

INPUT_ABS_WHEEL
: Absolute wheel position.

INPUT_ABS_X
: Absolute X coordinate.

INPUT_ABS_Y
: Absolute Y coordinate.

INPUT_ABS_Z
: Absolute Z coordinate.

Input event REL codes.

INPUT_REL_DIAL
: Relative dial coordinate.

INPUT_REL_HWHEEL
: Relative horizontal wheel coordinate.

INPUT_REL_MISC
: Relative misc coordinate.

INPUT_REL_RX
: Relative rotation around X axis.

INPUT_REL_RY
: Relative rotation around Y axis.

INPUT_REL_RZ
: Relative rotation around Z axis.

INPUT_REL_WHEEL
: Relative wheel coordinate.

INPUT_REL_X
: Relative X coordinate.

INPUT_REL_Y
: Relative Y coordinate.

INPUT_REL_Z
: Relative Z coordinate.

Input event MSC codes.

INPUT_MSC_SCAN
: Scan code.

4.9.9 Analog Axis API Reference

group `input_analog_axis`

Analog axis API.

Typedefs

typedef void (*`analog_axis_raw_data_t`)(const struct *device* *dev, int channel, int16_t raw_val)

Analog axis raw data callback.

Param dev
Analog axis device.

Param channel
Channel number.

Param raw_val
Raw value for the channel.

Functions

void `analog_axis_set_raw_data_cb`(const struct *device* *dev, *analog_axis_raw_data_t* cb)

Set a raw data callback.

Set a callback to receive raw data for the specified analog axis device. This is meant to be use in the application to acquire the data to use for calibration. Set cb to NULL to disable the callback.

Parameters

- `dev` – Analog axis device.
- `cb` – An analog_axis_raw_data_t callback to use, NULL disable.

int `analog_axis_num_axes`(const struct *device* *dev)

Get the number of defined axes.

Return values
`n` – The number of defined axes for dev.

int `analog_axis_calibration_get`(const struct *device* *dev, int channel, struct *analog_axis_calibration* *cal)

Get the axis calibration data.

Parameters

- `dev` – Analog axis device.
- `channel` – Channel number.
- `cal` – Pointer to an *analog_axis_calibration* structure that is going to get set with the current calibration data.

Return values

- `0` – If successful.
- `-EINVAL` – If the specified channel is not valid.

int **analog_axis_calibration_set**(const struct *device* *dev, int channel, struct *analog_axis_calibration* *cal)

Set the axis calibration data.

Parameters

- **dev** – Analog axis device.
- **channel** – Channel number.
- **cal** – Pointer to an *analog_axis_calibration* structure with the new calibration data

Return values

- **0** – If successful.
- **-EINVAL** – If the specified channel is not valid.

int **analog_axis_calibration_save**(const struct *device* *dev)

Save the calibration data.

Save the calibration data permanently on the specified device, requires the *Settings* subsystem to be configured and initialized.

Parameters

- **dev** – Analog axis device.

Return values

- **0** – If successful.
- **-errno** – In case of any other error.

struct **analog_axis_calibration**

#include <*input_analog_axis.h*> Analog axis calibration data structure.

Holds the calibration data for a single analog axis. Initial values are set from the devicetree and can be changed by the applicatoin in runtime using *analog_axis_calibration_set* and *analog_axis_calibration_get*.

Public Members

int16_t **in_min**

Input value that corresponds to the minimum output value.

int16_t **in_max**

Input value that corresponds to the maximum output value.

uint16_t **in_deadzone**

Input value center deadzone.

4.10 Interprocessor Communication (IPC)

4.10.1 IPC service

- *Overview*
- *Simple data exchange*
- *Data exchange using the no-copy API*
  - *Backends*
  - *API Reference*
- *IPC service API*
- *IPC service backend API*

The IPC service API provides an interface to exchange data between two domains or CPUs.

Overview

An IPC service communication channel consists of one instance and one or several endpoints associated with the instance.

An instance is the external representation of a physical communication channel between two domains or CPUs. The actual implementation and internal representation of the instance is peculiar to each backend.

An individual instance is not used to send data between domains/CPUs. To send and receive the data, the user must create (register) an endpoint in the instance. This allows for the connection of the two domains of interest.

It is possible to have zero or multiple endpoints for one single instance, possibly with different priorities, and to use each to exchange data. Endpoint prioritization and multi-instance ability highly depend on the backend used.

The endpoint is an entity the user must use to send and receive data between two domains (connected by the instance). An endpoint is always associated to an instance.

The creation of the instances is left to the backend, usually at init time. The registration of the endpoints is left to the user, usually at run time.

The API does not mandate a way for the backend to create instances but it is strongly recommended to use the devicetree to retrieve the configuration parameters for an instance. Currently, each backend defines its own DT-compatible configuration that is used to configure the interface at boot time.

The following usage scenarios are supported:

- Simple data exchange.
- Data exchange using the no-copy API.

Simple data exchange

To send data between domains or CPUs, an endpoint must be registered onto an instance.

See the following example:

Note

Before registering an endpoint, the instance must be opened using the `ipc_service_open_instance()` function.

```
#include <zephyr/ipc/ipc_service.h>

static void bound_cb(void *priv)
{
   /* Endpoint bounded */
}

static void recv_cb(const void *data, size_t len, void *priv)
{
   /* Data received */
}

static struct ipc_ept_cfg ept0_cfg = {
   .name = "ept0",
   .cb = {
      .bound    = bound_cb,
      .received = recv_cb,
   },
};

int main(void)
{
   const struct device *inst0;
   struct ipc_ept ept0;
   int ret;

   inst0 = DEVICE_DT_GET(DT_NODELABEL(ipc0));
   ret = ipc_service_open_instance(inst0);
   ret = ipc_service_register_endpoint(inst0, &ept0, &ept0_cfg);

   /* Wait for endpoint bound (bound_cb called) */

   unsigned char message[] = "hello world";
   ret = ipc_service_send(&ept0, &message, sizeof(message));
}
```

Data exchange using the no-copy API

If the backend supports the no-copy API you can use it to directly write and read to and from shared memory regions.

See the following example:

```
#include <zephyr/ipc/ipc_service.h>
#include <stdint.h>
#include <string.h>

static struct ipc_ept ept0;

static void bound_cb(void *priv)
{
   /* Endpoint bounded */
}

static void recv_cb_nocopy(const void *data, size_t len, void *priv)
{
   int ret;

   ret = ipc_service_hold_rx_buffer(&ept0, (void *)data);
   /* Process directly or put the buffer somewhere else and release. */
```

(continues on next page)

(continued from previous page)

```
    ret = ipc_service_release_rx_buffer(&ept0, (void *)data);
}

static struct ipc_ept_cfg ept0_cfg = {
    .name = "ept0",
    .cb = {
        .bound    = bound_cb,
        .received = recv_cb,
    },
};

int main(void)
{
    const struct device *inst0;
    int ret;

    inst0 = DEVICE_DT_GET(DT_NODELABEL(ipc0));
    ret = ipc_service_open_instance(inst0);
    ret = ipc_service_register_endpoint(inst0, &ept0, &ept0_cfg);

    /* Wait for endpoint bound (bound_cb called) */
    void *data;
    unsigned char message[] = "hello world";
    uint32_t len = sizeof(message);

    ret = ipc_service_get_tx_buffer(&ept0, &data, &len, K_FOREVER);

    memcpy(data, message, len);

    ret = ipc_service_send_nocopy(&ept0, data, sizeof(message));
}
```

Backends The requirements needed for implementing backends give flexibility to the IPC service. These allow for the addition of dedicated backends having only a subsets of features for specific use cases.

The backend must support at least the following:

- The init-time creation of instances.
- The run-time registration of an endpoint in an instance.

Additionally, the backend can also support the following:

- The run-time deregistration of an endpoint from the instance.
- The run-time closing of an instance.
- The no-copy API.

Each backend can have its own limitations and features that make the backend unique and dedicated to a specific use case. The IPC service API can be used with multiple backends simultaneously, combining the pros and cons of each backend.

ICMsg backend The inter core messaging backend (ICMsg) is a lighter alternative to the heavier RPMsg static vrings backend. It offers a minimal feature set in a small memory footprint. The ICMsg backend is build on top of *Single Producer Single Consumer Packet Buffer*.

Overview The ICMsg backend uses shared memory and MBOX devices for exchanging data. Shared memory is used to store the data, MBOX devices are used to signal that the data has been

written.

The backend supports the registration of a single endpoint on a single instance. If the application requires more than one communication channel, you must define multiple instances, each having its own dedicated endpoint.

Configuration The backend is configured via Kconfig and devicetree. When configuring the backend, do the following:

- Define two memory regions and assign them to `tx-region` and `rx-region` of an instance. Ensure that the memory regions used for data exchange are unique (not overlapping any other region) and accessible by both domains (or CPUs).
- Define MBOX devices which are used to send the signal that informs the other domain (or CPU) that data has been written. Ensure that the other domain (or CPU) is able to receive the signal.

See the following configuration example for one of the instances:

```
reserved-memory {
   tx: memory@20070000 {
      reg = <0x20070000 0x0800>;
   };

   rx: memory@20078000 {
      reg = <0x20078000 0x0800>;
   };
};

   ipc {
      ipc0: ipc0 {
         compatible = "zephyr,ipc-icmsg";
         tx-region = <&tx>;
         rx-region = <&rx>;
         mboxes = <&mbox 0>, <&mbox 1>;
         mbox-names = "tx", "rx";
         status = "okay";
      };
   };
};
```

You must provide a similar configuration for the other side of the communication (domain or CPU) but you must swap the MBOX channels and memory regions (`tx-region` and `rx-region`).

Bonding When the endpoint is registered, the following happens on each domain (or CPU) connected through the IPC instance:

1. The domain (or CPU) writes a magic number to its `tx-region` of the shared memory. #. It then sends a signal to the other domain or CPU, informing that the data has been written. Sending the signal to the other domain or CPU is repeated with timeout specified by `CONFIG_IPC_SERVICE_ICMSG_BOND_NOTIFY_REPEAT_TO_MS` option. #. When the signal from the other domain or CPU is received, the magic number is read from `rx-region`. If it is correct, the bonding process is finished and the backend informs the application by calling *`ipc_service_cb.bound`* callback.

Samples

- ipc-icmsg

ICMsg with dynamically allocated buffers backend This backend is built on top of the *ICMsg backend*. Data transferred over this backend travels in dynamically allocated buffers on shared memory. The ICMsg just sends references to the buffers. It also supports multiple endpoints.

This architecture allows for overcoming some common problems with other backends (mostly related to multithread access and zero-copy). This backend provides an alternative with no significant limitations.

Overview The shared memory is divided into two parts. One is reserved for the ICMsg and the other contains equal-sized blocks. The number of blocks is configured in the devicetree.

The data sending process is following:

- The sender allocates one or more blocks. If there are not enough sequential blocks, it waits using the timeout provided in the parameter that also includes K_FOREVER and K_NO_WAIT.
- The allocated blocks are filled with data. For the zero-copy case, this is done by the caller, otherwise, it is copied automatically. During this time other threads are not blocked in any way as long as there are enough free blocks for them. They can allocate, send data and receive data.
- A message containing the block index is sent over ICMsg to the receiver. The size of the ICMsg queue is large enough to hold messages for all blocks, so it will never overflow.
- The receiver can hold the data as long as desired. Again, other threads are not blocked as long as there are enough free blocks for them.
- When data is no longer needed, the backend sends a release message over ICMsg.
- When the backend receives this message, it deallocates all blocks. It is done internally by the backend and it is invisible to the caller.

Configuration The backend is configured using Kconfig and devicetree. When configuring the backend, do the following:

- Define two memory regions and assign them to `tx-region` and `rx-region` of an instance. Ensure that the memory regions used for data exchange are unique (not overlapping any other region) and accessible by both domains (or CPUs).
- Define the number of allocable blocks for each region with `tx-blocks` and `rx-blocks`.
- Define MBOX devices for sending a signal that informs the other domain (or CPU) of the written data. Ensure that the other domain (or CPU) can receive the signal.

See the following configuration example for one of the instances:

```
reserved-memory {
   tx: memory@20070000 {
      reg = <0x20070000 0x0800>;
   };

   rx: memory@20078000 {
      reg = <0x20078000 0x0800>;
   };
};

ipc {
   ipc0: ipc0 {
      compatible = "zephyr,ipc-icbmsg";
      tx-region = <&tx>;
      rx-region = <&rx>;
      tx-blocks = <16>;
```

(continues on next page)

(continued from previous page)

```
        rx-blocks = <32>;
        mboxes = <&mbox 0>, <&mbox 1>;
        mbox-names = "tx", "rx";
        status = "okay";
    };
};
```

You must provide a similar configuration for the other side of the communication (domain or CPU). Swap the MBOX channels, memory regions (`tx-region` and `rx-region`), and block count (`tx-blocks` and `rx-blocks`).

Samples

- ipc_multi_endpoint_sample

API Reference

IPC service API

group **ipc_service_api**

IPC Service API.

Functions

int **ipc_service_open_instance**(const struct *device* *instance)

Open an instance.

Function to be used to open an instance before being able to register a new endpoint on it.

Parameters

- **instance** – **[in]** Instance to open.

Return values

- **-EINVAL** – when instance configuration is invalid.
- **-EIO** – when no backend is registered.
- **-EALREADY** – when the instance is already opened (or being opened).
- **0** – on success or when not implemented on the backend (not needed).
- **other** – errno codes depending on the implementation of the backend.

int **ipc_service_close_instance**(const struct *device* *instance)

Close an instance.

Function to be used to close an instance. All bounded endpoints must be deregistered using ipc_service_deregister_endpoint before this is called.

Parameters

- **instance** – **[in]** Instance to close.

Return values

- **-EINVAL** – when instance configuration is invalid.

- `-EIO` – when no backend is registered.
- `-EALREADY` – when the instance is not already opened.
- `-EBUSY` – when an endpoint exists that hasn't been deregistered
- `0` – on success or when not implemented on the backend (not needed).
- `other` – errno codes depending on the implementation of the backend.

int `ipc_service_register_endpoint`(const struct *device* *instance, struct *ipc_ept* *ept, const struct *ipc_ept_cfg* *cfg)

Register IPC endpoint onto an instance.

Registers IPC endpoint onto an instance to enable communication with a remote device.

The same function registers endpoints for both host and remote devices.

Note

Keep the variable pointed by `cfg` alive when endpoint is in use.

Parameters

- `instance` – **[in]** Instance to register the endpoint onto.
- `ept` – **[in]** Endpoint object.
- `cfg` – **[in]** Endpoint configuration.

Return values

- `-EIO` – when no backend is registered.
- `-EINVAL` – when instance, endpoint or configuration is invalid.
- `-EBUSY` – when the instance is busy.
- `0` – on success.
- `other` – errno codes depending on the implementation of the backend.

int `ipc_service_deregister_endpoint`(struct *ipc_ept* *ept)

Deregister an IPC endpoint from its instance.

Deregisters an IPC endpoint from its instance.

The same function deregisters endpoints for both host and remote devices.

Parameters

- `ept` – **[in]** Endpoint object.

Return values

- `-EIO` – when no backend is registered.
- `-EINVAL` – when instance, endpoint or configuration is invalid.
- `-ENOENT` – when the endpoint is not registered with the instance.
- `-EBUSY` – when the instance is busy.
- `0` – on success.
- `other` – errno codes depending on the implementation of the backend.

int **ipc_service_send**(struct *ipc_ept* *ept, const void *data, size_t len)

Send data using given IPC endpoint.

Parameters

- **ept** – **[in]** Registered endpoint by *ipc_service_register_endpoint*.
- **data** – **[in]** Pointer to the buffer to send.
- **len** – **[in]** Number of bytes to send.

Return values

- **-EIO** – when no backend is registered or send hook is missing from backend.
- **-EINVAL** – when instance or endpoint is invalid.
- **-ENOENT** – when the endpoint is not registered with the instance.
- **-EBADMSG** – when the data is invalid (i.e. invalid data format, invalid length, ...)
- **-EBUSY** – when the instance is busy.
- **-ENOMEM** – when no memory / buffers are available.
- **bytes** – number of bytes sent.
- **other** – errno codes depending on the implementation of the backend.

int **ipc_service_get_tx_buffer_size**(struct *ipc_ept* *ept)

Get the TX buffer size.

Get the maximal size of a buffer which can be obtained by *ipc_service_get_tx_buffer*

Parameters

- **ept** – **[in]** Registered endpoint by *ipc_service_register_endpoint*.

Return values

- **-EIO** – when no backend is registered or send hook is missing from backend.
- **-EINVAL** – when instance or endpoint is invalid.
- **-ENOENT** – when the endpoint is not registered with the instance.
- **-ENOTSUP** – when the operation is not supported by backend.
- **size** – TX buffer size on success.
- **other** – errno codes depending on the implementation of the backend.

int **ipc_service_get_tx_buffer**(struct *ipc_ept* *ept, void **data, uint32_t *size, *k_timeout_t* wait)

Get an empty TX buffer to be sent using *ipc_service_send_nocopy*.

This function can be called to get an empty TX buffer so that the application can directly put its data into the sending buffer without copy from an application buffer.

It is the application responsibility to correctly fill the allocated TX buffer with data and passing correct parameters to *ipc_service_send_nocopy* function to perform data no-copy-send mechanism.

The size parameter can be used to request a buffer with a certain size:

- if the size can be accommodated the function returns no errors and the buffer is allocated

- if the requested size is too big, the function returns -ENOMEM and the the buffer is not allocated.
- if the requested size is '0' the buffer is allocated with the maximum allowed size.

In all the cases on return the size parameter contains the maximum size for the returned buffer.

When the function returns no errors, the buffer is intended as allocated and it is released under two conditions: (1) when sending the buffer using *ipc_service_send_nocopy* (and in this case the buffer is automatically released by the backend), (2) when using *ipc_service_drop_tx_buffer* on a buffer not sent.

Parameters

- `ept` – **[in]** Registered endpoint by *ipc_service_register_endpoint*.
- `data` – **[out]** Pointer to the empty TX buffer.
- `size` – **[inout]** Pointer to store the requested TX buffer size. If the function returns -ENOMEM, this parameter returns the maximum allowed size.
- `wait` – **[in]** Timeout waiting for an available TX buffer.

Return values

- `-EIO` – when no backend is registered or send hook is missing from backend.
- `-EINVAL` – when instance or endpoint is invalid.
- `-ENOENT` – when the endpoint is not registered with the instance.
- `-ENOTSUP` – when the operation or the timeout is not supported by backend.
- `-ENOBUFS` – when there are no TX buffers available.
- `-EALREADY` – when a buffer was already claimed and not yet released.
- `-ENOMEM` – when the requested size is too big (and the size parameter contains the maximum allowed size).
- `0` – on success.
- `other` – errno codes depending on the implementation of the backend.

int `ipc_service_drop_tx_buffer`(struct *ipc_ept* *ept, const void *data)

Drop and release a TX buffer.

Drop and release a TX buffer. It is possible to drop only TX buffers obtained by using *ipc_service_get_tx_buffer*.

Parameters

- `ept` – **[in]** Registered endpoint by *ipc_service_register_endpoint*.
- `data` – **[in]** Pointer to the TX buffer.

Return values

- `-EIO` – when no backend is registered or send hook is missing from backend.
- `-EINVAL` – when instance or endpoint is invalid.
- `-ENOENT` – when the endpoint is not registered with the instance.
- `-ENOTSUP` – when this is not supported by backend.
- `-EALREADY` – when the buffer was already dropped.

- `-ENXIO` – when the buffer was not obtained using *ipc_service_get_tx_buffer*
- `0` – on success.
- `other` – errno codes depending on the implementation of the backend.

int **ipc_service_send_nocopy**(struct *ipc_ept* *ept, const void *data, size_t len)

Send data in a TX buffer reserved by *ipc_service_get_tx_buffer* using the given IPC endpoint.

This is equivalent to *ipc_service_send* but in this case the TX buffer has been obtained by using *ipc_service_get_tx_buffer*.

The application has to take the responsibility for getting the TX buffer using *ipc_service_get_tx_buffer* and filling the TX buffer with the data.

After the *ipc_service_send_nocopy* function is issued the TX buffer is no more owned by the sending task and must not be touched anymore unless the function fails and returns an error.

If this function returns an error, *ipc_service_drop_tx_buffer* can be used to drop the TX buffer.

Parameters

- `ept` – **[in]** Registered endpoint by *ipc_service_register_endpoint*.
- `data` – **[in]** Pointer to the buffer to send obtained by *ipc_service_get_tx_buffer*.
- `len` – **[in]** Number of bytes to send.

Return values

- `-EIO` – when no backend is registered or send hook is missing from backend.
- `-EINVAL` – when instance or endpoint is invalid.
- `-ENOENT` – when the endpoint is not registered with the instance.
- `-EBADMSG` – when the data is invalid (i.e. invalid data format, invalid length, ...)
- `-EBUSY` – when the instance is busy.
- `bytes` – number of bytes sent.
- `other` – errno codes depending on the implementation of the backend.

int **ipc_service_hold_rx_buffer**(struct *ipc_ept* *ept, void *data)

Holds the RX buffer for usage outside the receive callback.

Calling this function prevents the receive buffer from being released back to the pool of shmem buffers. This function can be called in the receive callback when the user does not want to copy the message out in the callback itself.

After the message is processed, the application must release the buffer using the *ipc_service_release_rx_buffer* function.

Parameters

- `ept` – **[in]** Registered endpoint by *ipc_service_register_endpoint*.
- `data` – **[in]** Pointer to the RX buffer to hold.

Return values

- `-EIO` – when no backend is registered or release hook is missing from backend.

- `-EINVAL` – when instance or endpoint is invalid.
- `-ENOENT` – when the endpoint is not registered with the instance.
- `-EALREADY` – when the buffer data has been hold already.
- `-ENOTSUP` – when this is not supported by backend.
- `0` – on success.
- `other` – errno codes depending on the implementation of the backend.

int `ipc_service_release_rx_buffer`(struct *ipc_ept* *ept, void *data)

Release the RX buffer for future reuse.

When supported by the backend, this function can be called after the received message has been processed and the buffer can be marked as reusable again.

It is possible to release only RX buffers on which *ipc_service_hold_rx_buffer* was previously used.

Parameters

- `ept` – **[in]** Registered endpoint by *ipc_service_register_endpoint*.
- `data` – **[in]** Pointer to the RX buffer to release.

Return values

- `-EIO` – when no backend is registered or release hook is missing from backend.
- `-EINVAL` – when instance or endpoint is invalid.
- `-ENOENT` – when the endpoint is not registered with the instance.
- `-EALREADY` – when the buffer data has been already released.
- `-ENOTSUP` – when this is not supported by backend.
- `-ENXIO` – when the buffer was not hold before using *ipc_service_hold_rx_buffer*
- `0` – on success.
- `other` – errno codes depending on the implementation of the backend.

struct `ipc_service_cb`

#include <*ipc_service.h*> Event callback structure.

It is registered during endpoint registration. This structure is part of the endpoint configuration.

Public Members

void (*`bound`)(void *priv)

Bind was successful.

This callback is called when the endpoint binding is successful.

Param priv

[in] Private user data.

void (*`received`)(const void *data, size_t len, void *priv)

New packet arrived.

This callback is called when new data is received.

> **Note**
>
> When *ipc_service_hold_rx_buffer* is not used, the data buffer is to be considered released and available again only when this callback returns.

Param data
[in] Pointer to data buffer.
Param len
[in] Length of *data*.
Param priv
[in] Private user data.

void (*`error`)(const char *message, void *priv)

An error occurred.
Param message
[in] Error message.
Param priv
[in] Private user data.

struct `ipc_ept`

#include <ipc_service.h> Endpoint instance.

Token is not important for user of the API. It is implemented in a specific backend.

Public Members

const struct *device* *`instance`

Instance this endpoint belongs to.

void *`token`

Backend-specific token used to identify an endpoint in an instance.

struct `ipc_ept_cfg`

#include <ipc_service.h> Endpoint configuration structure.

Public Members

const char *`name`

Name of the endpoint.

int `prio`

Endpoint priority.

If the backend supports priorities.

struct *ipc_service_cb* **cb**
Event callback structure.

void ***priv**
Private user data.

IPC service backend API

group **ipc_service_backend**
IPC service backend.

struct **ipc_service_backend**
#include <*ipc_service_backend.h*> IPC backend configuration structure.

This structure is used for configuration backend during registration.

Public Members

int (***open_instance**)(const struct *device* *instance)
Pointer to the function that will be used to open an instance.
Param instance
[in] Instance pointer.
Retval -EALREADY
when the instance is already opened.
Retval 0
on success
Retval other
errno codes depending on the implementation of the backend.

int (***close_instance**)(const struct *device* *instance)
Pointer to the function that will be used to close an instance.
Param instance
[in] Instance pointer.
Retval -EALREADY
when the instance is not already inited.
Retval 0
on success
Retval other
errno codes depending on the implementation of the backend.

int (***send**)(const struct *device* *instance, void *token, const void *data, size_t len)
Pointer to the function that will be used to send data to the endpoint.
Param instance
[in] Instance pointer.
Param token
[in] Backend-specific token.
Param data
[in] Pointer to the buffer to send.
Param len
[in] Number of bytes to send.

Retval -EINVAL
when instance is invalid.
Retval -ENOENT
when the endpoint is not registered with the instance.
Retval -EBADMSG
when the message is invalid.
Retval -EBUSY
when the instance is busy or not ready.
Retval -ENOMEM
when no memory / buffers are available.
Retval bytes
number of bytes sent.
Retval other
errno codes depending on the implementation of the backend.

int (*`register_endpoint`)(const struct *device* *instance, void **token, const struct *ipc_ept_cfg* *cfg)

Pointer to the function that will be used to register endpoints.

Param instance
[in] Instance to register the endpoint onto.
Param token
[out] Backend-specific token.
Param cfg
[in] Endpoint configuration.
Retval -EINVAL
when the endpoint configuration or instance is invalid.
Retval -EBUSY
when the instance is busy or not ready.
Retval 0
on success
Retval other
errno codes depending on the implementation of the backend.

int (*`deregister_endpoint`)(const struct *device* *instance, void *token)

Pointer to the function that will be used to deregister endpoints.

Param instance
[in] Instance from which to deregister the endpoint.
Param token
[in] Backend-specific token.
Retval -EINVAL
when the endpoint configuration or instance is invalid.
Retval -ENOENT
when the endpoint is not registered with the instance.
Retval -EBUSY
when the instance is busy or not ready.
Retval 0
on success
Retval other
errno codes depending on the implementation of the backend.

int (*`get_tx_buffer_size`)(const struct *device* *instance, void *token)

Pointer to the function that will return the TX buffer size.

Param instance
[in] Instance pointer.
Param token
[in] Backend-specific token.

Retval -EINVAL
when instance is invalid.
Retval -ENOENT
when the endpoint is not registered with the instance.
Retval -ENOTSUP
when the operation is not supported.
Retval size
TX buffer size on success.
Retval other
errno codes depending on the implementation of the backend.

int (*`get_tx_buffer`)(const struct *device* *instance, void *token, void **data, uint32_t *len, *k_timeout_t* wait)

Pointer to the function that will return an empty TX buffer.
Param instance
[in] Instance pointer.
Param token
[in] Backend-specific token.
Param data
[out] Pointer to the empty TX buffer.
Param len
[inout] Pointer to store the TX buffer size.
Param wait
[in] Timeout waiting for an available TX buffer.
Retval -EINVAL
when instance is invalid.
Retval -ENOENT
when the endpoint is not registered with the instance.
Retval -ENOTSUP
when the operation or the timeout is not supported.
Retval -ENOBUFS
when there are no TX buffers available.
Retval -EALREADY
when a buffer was already claimed and not yet released.
Retval -ENOMEM
when the requested size is too big (and the size parameter contains the maximum allowed size).
Retval 0
on success
Retval other
errno codes depending on the implementation of the backend.

int (*`drop_tx_buffer`)(const struct *device* *instance, void *token, const void *data)

Pointer to the function that will drop a TX buffer.
Param instance
[in] Instance pointer.
Param token
[in] Backend-specific token.
Param data
[in] Pointer to the TX buffer.
Retval -EINVAL
when instance is invalid.
Retval -ENOENT
when the endpoint is not registered with the instance.
Retval -ENOTSUP
when this function is not supported.
Retval -EALREADY

when the buffer was already dropped.
Retval 0
on success
Retval other
errno codes depending on the implementation of the backend.

int (*`send_nocopy`)(const struct *device* *instance, void *token, const void *data, size_t len)

Pointer to the function that will be used to send data to the endpoint when the TX buffer has been obtained using *ipc_service_get_tx_buffer*.
Param instance
[in] Instance pointer.
Param token
[in] Backend-specific token.
Param data
[in] Pointer to the buffer to send.
Param len
[in] Number of bytes to send.
Retval -EINVAL
when instance is invalid.
Retval -ENOENT
when the endpoint is not registered with the instance.
Retval -EBADMSG
when the data is invalid (i.e. invalid data format, invalid length, ...)
Retval -EBUSY
when the instance is busy or not ready.
Retval bytes
number of bytes sent.
Retval other
errno codes depending on the implementation of the backend.

int (*`hold_rx_buffer`)(const struct *device* *instance, void *token, void *data)

Pointer to the function that will hold the RX buffer.
Param instance
[in] Instance pointer.
Param token
[in] Backend-specific token.
Param data
[in] Pointer to the RX buffer to hold.
Retval -EINVAL
when instance is invalid.
Retval -ENOENT
when the endpoint is not registered with the instance.
Retval -EALREADY
when the buffer data has been already hold.
Retval -ENOTSUP
when this function is not supported.
Retval 0
on success
Retval other
errno codes depending on the implementation of the backend.

int (*`release_rx_buffer`)(const struct *device* *instance, void *token, void *data)

Pointer to the function that will release the RX buffer.
Param instance
[in] Instance pointer.

Param token
[in] Backend-specific token.
Param data
[in] Pointer to the RX buffer to release.
Retval -EINVAL
when instance is invalid.
Retval -ENOENT
when the endpoint is not registered with the instance.
Retval -EALREADY
when the buffer data has been already released.
Retval -ENOTSUP
when this function is not supported.
Retval 0
on success
Retval other
errno codes depending on the implementation of the backend.

4.11 Linkable Loadable Extensions (LLEXT)

The LLEXT subsystem provides a toolbox for extending the functionality of an application at runtime with linkable loadable code.

Extensions are precompiled executables in ELF format that can be verified, loaded, and linked with the main Zephyr binary. Extensions can be manipulated and introspected to some degree, as well as unloaded when no longer needed.

4.11.1 Configuration

The following Kconfig options are available for the LLEXT subsystem:

Heap size

The LLEXT subsystem needs a static heap to be allocated for extension related data. The following option controls this allocation.

`CONFIG_LLEXT_HEAP_SIZE`

Size of the LLEXT heap in kilobytes.

Note

When *user mode* is enabled, the heap size must be large enough to allow the extension sections to be allocated with the alignment required by the architecture.

ELF object type

The LLEXT subsystem supports loading different types of extensions; the type can be set by choosing among the following Kconfig options:

`CONFIG_LLEXT_TYPE_ELF_OBJECT`

Build and expect relocatable files as binary object type for the LLEXT subsystem. A single compiler invocation is used to generate the object file.

`CONFIG_LLEXT_TYPE_ELF_RELOCATABLE`

Build and expect relocatable (partially linked) files as the binary object type for the LLEXT subsystem. These object files are generated by the linker by combining multiple object files into a single one.

`CONFIG_LLEXT_TYPE_ELF_SHAREDLIB`

Build and expect shared libraries as binary object type for the LLEXT subsystem. The standard linking process is used to generate the shared library from multiple object files.

> **Note**
>
> This is not currently supported on ARM architectures.

Minimize allocations

The LLEXT subsystem loading mechanism, by default, uses a seek/read abstraction and copies all data into allocated memory; this is done to allow the extension to be loaded from any storage medium. Sometimes, however, data is already in a buffer in RAM and copying it is not necessary. The following option allows the LLEXT subsystem to optimize memory footprint in this case.

`CONFIG_LLEXT_STORAGE_WRITABLE`

Allow the extension to be loaded by directly referencing section data into the ELF buffer. To be effective, this requires the use of an ELF loader that supports the peek functionality, such as the *`llext_buf_loader`*.

> **Warning**
>
> The application must ensure that the buffer used to load the extension remains allocated until the extension is unloaded.

> **Note**
>
> This will directly modify the contents of the buffer during the link phase. Once the extension is unloaded, the buffer must be reloaded before it can be used again in a call to *`llext_load()`*.

> **Note**
>
> This is currently required by the Xtensa architecture. Further information on this topic is available on GitHub issue #75341.

Using SLID for symbol lookups

When an extension is loaded, the LLEXT subsystem must find the address of all the symbols residing in the main application that the extension references. To this end, the main binary contains a LLEXT-dedicated symbol table, filled with one symbol-name-to-address mapping entry for each symbol exported by the main application to extensions. This table can then be searched into by the LLEXT linker at extension load time. This process is pretty slow due to the nature of

string comparisons, and the size consumed by the table can become significant as the number of exported symbols increases.

`CONFIG_LLEXT_EXPORT_BUILTINS_BY_SLID`

Perform an extra processing step on the Zephyr binary and on all extensions being built, converting every string in the symbol tables to a pointer-sized hash called *Symbol Link Identifier* (SLID), which is stored in the binary.

This speeds up the symbol lookup process by allowing usage of integer-based comparisons rather than string-based ones. Another benefit of SLID-based linking is that storing symbol names in the binary is no longer necessary, which provides a significant decrease in symbol table size.

Note

This option is not currently compatible with the *LLEXT EDK*.

Note

Using a different value for this option in the main binary and in extensions is not supported. For example, if the main application is built with `CONFIG_LLEXT_EXPORT_BUILTINS_BY_SLID=y`, it is forbidden to load an extension that was compiled with `CONFIG_LLEXT_EXPORT_BUILTINS_BY_SLID=n`.

EDK configuration

Options influencing the generation and behavior of the LLEXT EDK are described in *LLEXT EDK Kconfig options*.

4.11.2 Building extensions

The LLEXT subsystem allows for the creation of extensions that can be loaded into a running Zephyr application. When building these extensions, it's very often useful to have access to the headers and compiler flags used by the main Zephyr application.

The easiest path to achieve this is to build the extension as part of the Zephyr application, using the *native Zephyr CMake features*. This will result in a single build providing both the main Zephyr application and the extension(s), which will all automatically be built with the same parameters.

In some cases, involving the full Zephyr build system may not be feasible or convenient; maybe the extension is built using a different compiler suite or as part of a different project altogether. In this case, the extension developer needs to export the headers and compiler flags used by the main Zephyr application. This can be done using the *LLEXT Extension Development Kit*.

Using the Zephyr CMake features

The Zephyr build system provides a set of features that can be used to build extensions as part of the Zephyr application. This is the simplest way to build extensions, as it requires minimal additions to an application build system.

Building the extension An extension can be defined in the app's `CMakeLists.txt` by invoking the `add_llext_target` function, providing the target name, the output and the source files. Usage is similar to the standard `add_custom_target` CMake function:

```
add_llext_target(
    <target_name>
    OUTPUT <ext_file.llext>
    SOURCES <src1> [<src2>...]
)
```

where:

- `<target_name>` is the name of the final CMake target that will result in the LLEXT binary being created;
- `<ext_file.llext>` is the name of the output file that will contain the packaged extension;
- `<src1> [<src2>...]` is the list of source files that will be compiled to create the extension.

The exact steps of the extension building process depend on the currently selected *ELF object format*.

The following custom properties of `<target_name>` are defined and can be retrieved using the `get_target_property()` CMake function:

`lib_target`

Target name for the source compilation and/or link step.

`lib_output`

The binary file resulting from compilation and/or linking steps.

`pkg_input`

The file to be used as input for the packaging step.

`pkg_output`

The final extension file name.

Tweaking the build process The following CMake functions can be used to modify the build system behavior during the extension build process to a fine degree. Each of the below functions takes the LLEXT target name as its first argument; it is otherwise functionally equivalent to the common Zephyr `target_*` version.

- `llext_compile_definitions`
- `llext_compile_features`
- `llext_compile_options`
- `llext_include_directories`
- `llext_link_options`

Custom build steps The `add_llext_command` CMake function can be used to add custom build steps that will be executed during the extension build process. The command will be run at the specified build step and can refer to the properties of the target for build-specific details.

The function signature is:

```
add_llext_command(
    TARGET <target_name>
    [PRE_BUILD | POST_BUILD | POST_PKG]
    COMMAND <command> [args...]
)
```

The different build steps are:

PRE_BUILD

: Before the extension code is linked, if the architecture uses dynamic libraries. This step can access *lib_target* and its own properties.

POST_BUILD

: After the extension code is built, but before packaging it in an `.llext` file. This step is expected to create a *pkg_input* file by reading the contents of *lib_output*.

POST_PKG

: After the extension output file has been created. The command can operate on the final llext file *pkg_output*.

Anything else after `COMMAND` will be passed to `add_custom_command()` as-is (including multiple commands and other options).

LLEXT Extension Development Kit (EDK)

When building extensions as a standalone project, outside of the main Zephyr build system, it's important to have access to the same set of generated headers and compiler flags used by the main Zephyr application, since they have a direct impact on how Zephyr headers are interpreted and the extension is compiled in general.

This can be achieved by asking Zephyr to generate an Extension Development Kit (EDK) from the build artifacts of the main Zephyr application, by running the following command which uses the `llext-edk` target:

```
west build -t llext-edk
```

The generated EDK can be found in the build directory under the `zephyr` directory. It's a tarball that contains the headers and compile flags needed to build extensions. The extension developer can then include the headers and use the compile flags in their build system to build the extension.

Compile flags The EDK includes the convenience files `cmake.cflags` (for CMake-based projects) and `Makefile.cflags` (for Make-based ones), which define a set of variables that contain the compile flags needed by the project. The full list of flags needed to build an extension is provided by `LLEXT_CFLAGS`. Also provided is a more granular set of flags that can be used in support of different use cases, such as when building mocks for unit tests:

LLEXT_INCLUDE_CFLAGS

: Compiler flags to add directories containing non-autogenerated headers to the compiler's include search paths.

LLEXT_GENERATED_INCLUDE_CFLAGS

: Compiler flags to add directories containing autogenerated headers to the compiler's include search paths.

LLEXT_ALL_INCLUDE_CFLAGS

: Compiler flags to add all directories containing headers used in the build to the compiler's include search paths. This is a combination of `LLEXT_INCLUDE_CFLAGS` and `LLEXT_GENERATED_INCLUDE_CFLAGS`.

LLEXT_GENERATED_IMACROS_CFLAGS

: Compiler flags for autogenerated headers that must be included in the build via `-imacros`.

`LLEXT_BASE_CFLAGS`

Other compiler flags that control code generation for the target CPU. None of these flags are included in the above lists.

`LLEXT_CFLAGS`

All flags required to build an extension. This is a combination of `LLEXT_ALL_INCLUDE_CFLAGS`, `LLEXT_GENERATED_IMACROS_CFLAGS` and `LLEXT_BASE_CFLAGS`.

LLEXT EDK Kconfig options The LLEXT EDK can be configured using the following Kconfig options:

`CONFIG_LLEXT_EDK_NAME`
The name of the generated EDK tarball.

`CONFIG_LLEXT_EDK_USERSPACE_ONLY`
If set, the EDK will include headers that do not contain code to route syscalls to the kernel. This is useful when building extensions that will run exclusively in user mode.

EDK Sample Refer to llext-edk for an example of how to use the LLEXT EDK.

4.11.3 Loading extensions

Once an extension is built and the ELF file is available, it can be loaded into the Zephyr application using the LLEXT API, which provides a way to load the extension into memory, access its symbols and call its functions.

Loading an extension

An extension may be loaded using any implementation of a *`llext_loader`* which has a set of function pointers that provide the necessary functionality to read the ELF data. A loader also provides some minimal context (memory) needed by the *`llext_load()`* function. An implementation over a buffer containing an ELF in addressable memory in memory is available as *`llext_buf_loader`*.

The extensions are loaded with a call to the *`llext_load()`* function, passing in the extension name and the configured loader. Once that completes successfully, the extension is loaded into memory and is ready to be used.

Note

When *User Mode* is enabled, the extension will not be included in any user memory domain. To allow access from user mode, the *`llext_add_domain()`* function must be called.

Accessing code and data

To interact with the newly loaded extension, the host application must use the *`llext_find_sym()`* function to get the address of the exported symbol. The returned `void *` can then be cast to the appropriate type and used.

A wrapper for calling a function with no arguments is provided in *`llext_call_fn()`*.

Cleaning up after use

The *llext_unload()* function must be called to free the memory used by the extension once it is no longer required. After this call completes, all pointers to symbols in the extension that were obtained will be invalid.

4.11.4 Troubleshooting

This feature is being actively developed and as such it is possible that some issues may arise. Since linking does modify the binary code, in case of errors the results are difficult to predict. Some common issues may be:

- Results from *llext_find_sym()* point to an invalid address;
- Constants and variables defined in the extension do not have the expected values;
- Calling a function defined in an extension results in a hard fault, or memory in the main application is corrupted after returning from it.

If any of this happens, the following tips may help understand the issue:

- Make sure `CONFIG_LLEXT_LOG_LEVEL` is set to `DEBUG`, then obtain a log of the *llext_load()* invocation.
- If possible, disable memory protection (MMU/MPU) and see if this results in different behavior.
- Try to simplify the extension to the minimum possible code that reproduces the issue.
- Use a debugger to inspect the memory and registers to try to understand what is happening.

> **Note**
>
> When using GDB, the `add_symbol_file` command may be used to load the debugging information and symbols from the ELF file. Make sure to specify the proper offset (usually the start of the `.text` section, reported as `region 0` in the debug logs.)

If the issue persists, please open an issue in the GitHub repository, including all the above information.

4.11.5 API Reference

group `llext_apis`

Since
3.5

Version
0.1.0

Defines

`LLEXT_LOAD_PARAM_DEFAULT`
Default initializer for *llext_load_param*.

Enums

enum `llext_mem`

List of memory regions stored or referenced in the LLEXT subsystem.

This enum lists the different types of memory regions that are used by the LLEXT subsystem. The names match common ELF file section names; but note that at load time multiple ELF sections with similar flags may be merged together into a single memory region.

Values:

enumerator `LLEXT_MEM_TEXT`

Executable code.

enumerator `LLEXT_MEM_DATA`

Initialized data.

enumerator `LLEXT_MEM_RODATA`

Read-only data.

enumerator `LLEXT_MEM_BSS`

Uninitialized data.

enumerator `LLEXT_MEM_EXPORT`

Exported symbol table.

enumerator `LLEXT_MEM_SYMTAB`

Symbol table.

enumerator `LLEXT_MEM_STRTAB`

Symbol name strings.

enumerator `LLEXT_MEM_SHSTRTAB`

Section name strings.

enumerator `LLEXT_MEM_COUNT`

Number of regions managed by LLEXT.

Functions

struct *llext* *`llext_by_name`(const char *name)

Find an llext by name.

Parameters

- `name` – **[in]** String name of the llext

Returns

a pointer to the *llext*, or `NULL` if not found

int `llext_iterate`(int (*fn)(struct *llext* *ext, void *arg), void *arg)

Iterate over all loaded extensions.

Calls a provided callback function for each registered extension or until the callback function returns a non-0 value.

Parameters

- `fn` – **[in]** callback function
- `arg` – **[in]** a private argument to be provided to the callback function

Return values

`0` – if no extensions are registered

Returns

the value returned by the last callback invocation

int `llext_load`(struct *llext_loader* *loader, const char *name, struct *llext* **ext, struct *llext_load_param* *ldr_parm)

Load and link an extension.

Loads relevant ELF data into memory and provides a structure to work with it.

Parameters

- `loader` – **[in]** An extension loader that provides input data and context
- `name` – **[in]** A string identifier for the extension
- `ext` – **[out]** Pointer to the newly allocated *llext* structure
- `ldr_parm` – **[in]** Optional advanced load parameters (may be `NULL`)

Return values

- `-ENOMEM` – Not enough memory
- `-ENOEXEC` – Invalid ELF stream
- `-ENOTSUP` – Unsupported ELF features

Returns

the previous extension use count on success, or a negative error code.

int `llext_unload`(struct *llext* **ext)

Unload an extension.

Parameters

- `ext` – **[in]** Extension to unload

const void *`llext_find_sym`(const struct *llext_symtable* *sym_table, const char *sym_name)

Find the address for an arbitrary symbol.

Searches for a symbol address, either in the list of symbols exported by the main Zephyr binary or in an extension's symbol table.

Parameters

- `sym_table` – **[in]** Symbol table to lookup symbol in, or `NULL` to search in the main Zephyr symbol table
- `sym_name` – **[in]** Symbol name to find

Returns

the address of symbol in memory, or `NULL` if not found

int `llext_call_fn`(struct *llext* *ext, const char *sym_name)

Call a function by name.

Expects a symbol representing a `void fn(void)` style function exists and may be called.

Parameters

- `ext` – **[in]** Extension to call function in
- `sym_name` – **[in]** Function name (exported symbol) in the extension

Return values

- `0` – Success
- `-ENOENT` – Symbol name not found

int `llext_add_domain`(struct *llext* *ext, struct *k_mem_domain* *domain)

Add an extension to a memory domain.

Allows an extension to be executed in user mode threads when memory protection hardware is enabled by adding memory partitions covering the extension's memory regions to a memory domain.

Parameters

- `ext` – **[in]** Extension to add to a domain
- `domain` – **[in]** Memory domain to add partitions to

Return values

`-ENOSYS` – `CONFIG_USERSPACE` is not enabled or supported

Returns

0 on success, or a negative error code.

int `arch_elf_relocate`(elf_rela_t *rel, uintptr_t loc, uintptr_t sym_base_addr, const char *sym_name, uintptr_t load_bias)

Architecture specific opcode update function.

ELF files include sections describing a series of *relocations*, which are instructions on how to rewrite opcodes given the actual placement of some symbolic data such as a section, function, or object. These relocations are architecture specific and each architecture supporting LLEXT must implement this.

Parameters

- `rel` – **[in]** Relocation data provided by ELF
- `loc` – **[in]** Address of opcode to rewrite
- `sym_base_addr` – **[in]** Address of symbol referenced by relocation
- `sym_name` – **[in]** Name of symbol referenced by relocation
- `load_bias` – **[in]** `.text` load address

Return values

- `0` – Success
- `-ENOTSUP` – Unsupported relocation
- `-ENOEXEC` – Invalid relocation

ssize_t `llext_find_section`(struct *llext_loader* *loader, const char *search_name)

Locates an ELF section in the file.

Searches for a section by name in the ELF file and returns its offset.

Parameters

- **loader** – Extension loader data and context
- **search_name** – Section name to search for

Returns
the section offset or a negative error code

void **arch_elf_relocate_local**(struct *llext_loader* *loader, struct *llext* *ext, const elf_rela_t *rel, const elf_sym_t *sym, size_t got_offset)

Architecture specific function for updating addresses via relocation table.

Parameters

- **loader** – **[in]** Extension loader data and context
- **ext** – **[in]** Extension to call function in
- **rel** – **[in]** Relocation data provided by elf
- **sym** – **[in]** Corresponding symbol table entry
- **got_offset** – **[in]** Offset within a relocation table

struct **llext**

#include <*llext.h*> Structure describing a linkable loadable extension.

This structure holds the data for a loaded extension. It is created by the *llext_load* function and destroyed by the *llext_unload* function.

Public Members

char **name**[16]

Name of the llext.

void ***mem**[*LLEXT_MEM_COUNT*]

Lookup table of memory regions.

bool **mem_on_heap**[*LLEXT_MEM_COUNT*]

Is the memory for this region allocated on heap?

size_t **mem_size**[*LLEXT_MEM_COUNT*]

Size of each stored region.

size_t **alloc_size**

Total llext allocation size.

struct *llext_symtable* **sym_tab**

Table of all global symbols in the extension; used internally as part of the linking process.

E.g. if the extension is built out of several files, if any symbols are referenced between files, this table will be used to link them.

struct *llext_symtable* **exp_tab**

Table of symbols exported by the llext via *LL_EXTENSION_SYMBOL*.

This can be used in the main Zephyr binary to find symbols in the extension.

unsigned int **use_count**

Extension use counter, prevents unloading while in use.

struct **llext_load_param**

#include <*llext.h*> Advanced llext_load parameters.

This structure contains advanced parameters for *llext_load*.

Public Members

bool **relocate_local**

Perform local relocation.

bool **pre_located**

Use the virtual symbol addresses from the ELF, not addresses within the memory buffer, when calculating relocation targets.

group **llext_symbols**

Defines

EXPORT_SYMBOL(x)

Export a constant symbol to extensions.

Takes a symbol (function or object) by symbolic name and adds the name and address of the symbol to a table of symbols that may be referenced by extensions.

Parameters

- x – Symbol to export to extensions

LL_EXTENSION_SYMBOL(x)

Exports a symbol from an extension to the base image.

This macro can be used in extensions to add a symbol (function or object) to the extension's exported symbol table, so that it may be referenced by the base image.

Parameters

- x – Extension symbol to export to the base image

struct **llext_const_symbol**

#include <*symbol.h*> Constant symbols are unchangeable named memory addresses.

Symbols may be named function or global objects that have been exported for linking. These constant symbols are useful in the base image as they may be placed in ROM.

Note

When updating this structure, make sure to also update the 'scripts/build/llext_prepare_exptab.py' build script.

Public Members

const char *const **name**
: Name of symbol.

const uintptr_t **slid**
: Symbol Link Identifier.

union llext_const_symbol
: At build time, we always write to 'name'.

  At runtime, which field is used depends on CONFIG_LLEXT_EXPORT_BUILTINS_BY_SLID.

const void *const **addr**
: Address of symbol.

struct **llext_symbol**
: *#include* <*symbol.h*> Symbols are named memory addresses.

  Symbols may be named function or global objects that have been exported for linking. These are mutable and should come from extensions where the location may need updating depending on where memory is placed.

Public Members

const char ***name**
: Name of symbol.

void ***addr**
: Address of symbol.

struct **llext_symtable**
: *#include* <*symbol.h*> A symbol table.

  An array of symbols

Public Members

size_t **sym_cnt**
: Number of symbols in the table.

struct *llext_symbol* ***syms**
: Array of symbols.

group **llext_loader_apis**

Defines

LLEXT_BUF_LOADER(_buf, _buf_len)

Initializer for an *llext_buf_loader* structure.

Parameters

- **_buf** – Buffer containing the ELF binary
- **_buf_len** – Buffer length in bytes

struct **llext_buf_loader**

#include <buf_loader.h> Implementation of *llext_loader* that reads from a memory buffer.

Public Members

struct *llext_loader* **loader**

Extension loader.

struct **llext_loader**

#include <loader.h> Linkable loadable extension loader context.

This object is used to access the ELF file data and cache its contents while an extension is being loaded by the LLEXT subsystem. Once the extension is loaded, this object is no longer needed.

Public Members

int (***read**)(struct *llext_loader* *ldr, void *out, size_t len)

Function to read (copy) from the loader.

Copies len bytes into buf from the current position of the loader.

Param ldr
[in] Loader
Param out
[in] Output location
Param len
[in] Length to copy into the output location
Return
0 on success, or a negative error code.

int (***seek**)(struct *llext_loader* *ldr, size_t pos)

Function to seek to a new absolute location in the stream.

Changes the location of the loader position to a new absolute given position.

Param ldr
[in] Loader
Param pos
[in] Position in stream to move loader
Return
0 on success, or a negative error code.

void *(*peek)(struct *llext_loader* *ldr, size_t pos)

Optional function to peek at an absolute location in the ELF.

Return a pointer to the buffer at specified offset.

Param ldr
[in] Loader
Param pos
[in] Position to obtain a pointer to
Return
a pointer into the buffer or `NULL` if not supported

Note

The LLEXT subsystem requires architecture-specific support. It is currently available only on ARM and Xtensa cores.

4.12 Logging

- *Global Kconfig Options*
- *Usage*
  - *Logging in a module*
  - *Logging in a module instance*
  - *Controlling the logging*
- *Logging panic*
- *Printk*
- *Architecture*
  - *Default Frontend*
  - *Custom Frontend*
  - *Logging strings*
  - *Multi-domain support*
  - *Logging backends*
  - *Dictionary-based Logging*
- *Recommendations*
- *Benchmark*
- *Stack usage*
- *API Reference*
  - *Logger API*
  - *Logger control*
  - *Log message*
  - *Logger backend interface*
  - *Logger output formatting*

The logging API provides a common interface to process messages issued by developers. Messages are passed through a frontend and are then processed by active backends. Custom frontend and backends can be used if needed.

Summary of the logging features:

- Deferred logging reduces the time needed to log a message by shifting time consuming operations to a known context instead of processing and sending the log message when called.
- Multiple backends supported (up to 9 backends).
- Custom frontend support. It can work together with backends.
- Compile time filtering on module level.
- Run time filtering independent for each backend.
- Additional run time filtering on module instance level.
- Timestamping with user provided function. Timestamp can have 32 or 64 bits.
- Dedicated API for dumping data.
- Dedicated API for handling transient strings.
- Panic support - in panic mode logging switches to blocking, synchronous processing.

- Printk support - printk message can be redirected to the logging.
- Design ready for multi-domain/multi-processor system.
- Support for logging floating point variables and long long arguments.
- Built-in copying of transient strings used as arguments.
- Support for multi-domain logging.

Logging API is highly configurable at compile time as well as at run time. Using Kconfig options (see *Global Kconfig Options*) logs can be gradually removed from compilation to reduce image size and execution time when logs are not needed. During compilation logs can be filtered out on module basis and severity level.

Logs can also be compiled in but filtered on run time using dedicate API. Run time filtering is independent for each backend and each source of log messages. Source of log messages can be a module or specific instance of the module.

There are four severity levels available in the system: error, warning, info and debug. For each severity level the logging API (include/zephyr/logging/log.h) has set of dedicated macros. Logger API also has macros for logging data.

For each level the following set of macros are available:

- `LOG_X` for standard printf-like messages, e.g. *`LOG_ERR`*.
- `LOG_HEXDUMP_X` for dumping data, e.g. *`LOG_HEXDUMP_WRN`*.
- `LOG_INST_X` for standard printf-like message associated with the particular instance, e.g. *`LOG_INST_INF`*.
- `LOG_INST_HEXDUMP_X` for dumping data associated with the particular instance, e.g. *`LOG_INST_HEXDUMP_DBG`*

The warning level also exposes the following additional macro:

- *`LOG_WRN_ONCE`* for warnings where only the first occurrence is of interest.

There are two configuration categories: configurations per module and global configuration. When logging is enabled globally, it works for modules. However, modules can disable logging locally. Every module can specify its own logging level. The module must define the `LOG_LEVEL` macro before using the API. Unless a global override is set, the module logging level will be honored. The global override can only increase the logging level. It cannot be used to lower module logging levels that were previously set higher. It is also possible to globally limit logs by providing maximal severity level present in the system, where maximal means lowest severity (e.g. if maximal level in the system is set to info, it means that errors, warnings and info levels are present but debug messages are excluded).

Each module which is using the logging must specify its unique name and register itself to the logging. If module consists of more than one file, registration is performed in one file but each file must define a module name.

Logger's default frontend is designed to be thread safe and minimizes time needed to log the message. Time consuming operations like string formatting or access to the transport are not performed by default when logging API is called. When logging API is called a message is created and added to the list. Dedicated, configurable buffer for pool of log messages is used. There are 2 types of messages: standard and hexdump. Each message contain source ID (module or instance ID and domain ID which might be used for multiprocessor systems), timestamp and severity level. Standard message contains pointer to the string and arguments. Hexdump message contains copied data and string.

4.12.1 Global Kconfig Options

These options can be found in the following path subsys/logging/Kconfig.

CONFIG_LOG: Global switch, turns on/off the logging.

Mode of operations:

CONFIG_LOG_MODE_DEFERRED: Deferred mode.

CONFIG_LOG_MODE_IMMEDIATE: Immediate (synchronous) mode.

CONFIG_LOG_MODE_MINIMAL: Minimal footprint mode.

Filtering options:

CONFIG_LOG_RUNTIME_FILTERING: Enables runtime reconfiguration of the filtering.

CONFIG_LOG_DEFAULT_LEVEL: Default level, sets the logging level used by modules that are not setting their own logging level.

CONFIG_LOG_OVERRIDE_LEVEL: It overrides module logging level when it is not set or set lower than the override value.

CONFIG_LOG_MAX_LEVEL: Maximal (lowest severity) level which is compiled in.

Processing options:

CONFIG_LOG_MODE_OVERFLOW: When new message cannot be allocated, oldest one are discarded.

CONFIG_LOG_BLOCK_IN_THREAD: If enabled and new log message cannot be allocated thread context will block for up to CONFIG_LOG_BLOCK_IN_THREAD_TIMEOUT_MS or until log message is allocated.

CONFIG_LOG_PRINTK: Redirect printk calls to the logging.

CONFIG_LOG_PROCESS_TRIGGER_THRESHOLD: When the number of buffered log messages reaches the threshold, the dedicated thread (see *log_thread_set()*) is woken up. If CONFIG_LOG_PROCESS_THREAD is enabled then this threshold is used by the internal thread.

CONFIG_LOG_PROCESS_THREAD: When enabled, logging thread is created which handles log processing.

CONFIG_LOG_PROCESS_THREAD_STARTUP_DELAY_MS: Delay in milliseconds after which logging thread is started.

CONFIG_LOG_BUFFER_SIZE: Number of bytes dedicated for the circular packet buffer.

CONFIG_LOG_FRONTEND: Direct logs to a custom frontend.

CONFIG_LOG_FRONTEND_ONLY: No backends are used when messages goes to frontend.

CONFIG_LOG_FRONTEND_OPT_API: Optional API optimized for the most common simple messages.

CONFIG_LOG_CUSTOM_HEADER: Injects an application provided header into log.h

CONFIG_LOG_TIMESTAMP_64BIT: 64 bit timestamp.

CONFIG_LOG_SIMPLE_MSG_OPTIMIZE: Optimizes simple log messages for size and performance. Option available only for 32 bit architectures.

Formatting options:

CONFIG_LOG_FUNC_NAME_PREFIX_ERR: Prepend standard ERROR log messages with function name. Hexdump messages are not prepended.

CONFIG_LOG_FUNC_NAME_PREFIX_WRN: Prepend standard WARNING log messages with function name. Hexdump messages are not prepended.

CONFIG_LOG_FUNC_NAME_PREFIX_INF: Prepend standard INFO log messages with function name. Hexdump messages are not prepended.

CONFIG_LOG_FUNC_NAME_PREFIX_DBG: Prepend standard DEBUG log messages with function name. Hexdump messages are not prepended.

CONFIG_LOG_BACKEND_SHOW_COLOR: Enables coloring of errors (red) and warnings (yellow).

CONFIG_LOG_BACKEND_FORMAT_TIMESTAMP: If enabled timestamp is formatted to *hh:mm:ss:mmm,uuu*. Otherwise is printed in raw format.

Backend options:

CONFIG_LOG_BACKEND_UART: Enabled built-in UART backend.

4.12.2 Usage

Logging in a module

In order to use logging in the module, a unique name of a module must be specified and module must be registered using *LOG_MODULE_REGISTER*. Optionally, a compile time log level for the module can be specified as the second parameter. Default log level (CONFIG_LOG_DEFAULT_LEVEL) is used if custom log level is not provided.

```
#include <zephyr/logging/log.h>
LOG_MODULE_REGISTER(foo, CONFIG_FOO_LOG_LEVEL);
```

If the module consists of multiple files, then LOG_MODULE_REGISTER() should appear in exactly one of them. Each other file should use *LOG_MODULE_DECLARE* to declare its membership in the module. Optionally, a compile time log level for the module can be specified as the second parameter. Default log level (CONFIG_LOG_DEFAULT_LEVEL) is used if custom log level is not provided.

```
#include <zephyr/logging/log.h>
/* In all files comprising the module but one */
LOG_MODULE_DECLARE(foo, CONFIG_FOO_LOG_LEVEL);
```

In order to use logging API in a function implemented in a header file *LOG_MODULE_DECLARE* macro must be used in the function body before logging API is called. Optionally, a compile time log level for the module can be specified as the second parameter. Default log level (CONFIG_LOG_DEFAULT_LEVEL) is used if custom log level is not provided.

```
#include <zephyr/logging/log.h>

static inline void foo(void)
{
     LOG_MODULE_DECLARE(foo, CONFIG_FOO_LOG_LEVEL);

     LOG_INF("foo");
}
```

Dedicated Kconfig template (subsys/logging/Kconfig.template.log_config) can be used to create local log level configuration.

Example below presents usage of the template. As a result CONFIG_FOO_LOG_LEVEL will be generated:

```
module = FOO
module-str = foo
source "subsys/logging/Kconfig.template.log_config"
```

Logging in a module instance

In case of modules which are multi-instance and instances are widely used across the system enabling logs will lead to flooding. The logger provides the tools which can be used to provide filtering on instance level rather than module level. In that case logging can be enabled for particular instance.

In order to use instance level filtering following steps must be performed:

- a pointer to specific logging structure is declared in instance structure. LOG_INSTANCE_PTR_DECLARE is used for that.

```
#include <zephyr/logging/log_instance.h>

struct foo_object {
     LOG_INSTANCE_PTR_DECLARE(log);
     uint32_t id;
}
```

- module must provide macro for instantiation. In that macro, logging instance is registered and log instance pointer is initialized in the object structure.

```
#define FOO_OBJECT_DEFINE(_name)                                \
     LOG_INSTANCE_REGISTER(foo, _name, CONFIG_FOO_LOG_LEVEL) \
     struct foo_object _name = {                                \
             LOG_INSTANCE_PTR_INIT(log, foo, _name)          \
     }
```

Note that when logging is disabled logging instance and pointer to that instance are not created.

In order to use the instance logging API in a source file, a compile-time log level must be set using *LOG_LEVEL_SET*.

```
LOG_LEVEL_SET(CONFIG_FOO_LOG_LEVEL);

void foo_init(foo_object *f)
{
     LOG_INST_INF(f->log, "Initialized.");
}
```

In order to use the instance logging API in a header file, a compile-time log level must be set using *LOG_LEVEL_SET*.

```
static inline void foo_init(foo_object *f)
{
     LOG_LEVEL_SET(CONFIG_FOO_LOG_LEVEL);

     LOG_INST_INF(f->log, "Initialized.");
}
```

Controlling the logging

By default, logging processing in deferred mode is handled internally by the dedicated task which starts automatically. However, it might not be available if multithreading is disabled. It can also be disabled by unsetting CONFIG_LOG_PROCESS_TRIGGER_THRESHOLD. In that case, logging can be controlled using the API defined in include/zephyr/logging/log_ctrl.h. Logging must be initialized before it can be used. Optionally, the user can provide a function which returns the timestamp value. If not provided, k_cycle_get or *k_cycle_get_32* is used for timestamping. The *log_process()* function is used to trigger processing of one log message (if pending), and returns true if there are more messages pending. However, it is recommended to use macro wrappers (*LOG_INIT* and *LOG_PROCESS*) which handle the case where logging is disabled.

The following snippet shows how logging can be processed in simple forever loop.

```
#include <zephyr/logging/log_ctrl.h>

int main(void)
{
     LOG_INIT();
```

(continues on next page)

(continued from previous page)

```
	/* If multithreading is enabled provide thread id to the logging. */
	log_thread_set(k_current_get());

	while (1) {
		if (LOG_PROCESS() == false) {
			/* sleep */
		}
	}
}
```

If logs are processed from a thread (user or internal) then it is possible to enable a feature which will wake up processing thread when certain amount of log messages are buffered (see `CONFIG_LOG_PROCESS_TRIGGER_THRESHOLD`).

4.12.3 Logging panic

In case of error condition system usually can no longer rely on scheduler or interrupts. In that situation deferred log message processing is not an option. Logger controlling API provides a function for entering into panic mode (*log_panic()*) which should be called in such situation.

When *log_panic()* is called, _panic_ notification is sent to all active backends. Once all backends are notified, all buffered messages are flushed. Since that moment all logs are processed in a blocking way.

4.12.4 Printk

Typically, logging and `printk()` use the same output, which they compete for. This can lead to issues if the output does not support preemption but it may also result in corrupted output because logging data is interleaved with printk data. However, it is possible to redirect printk messages to the logging subsystem by enabling `CONFIG_LOG_PRINTK`. In that case, printk entries are treated as log messages with level 0 (they cannot be disabled). When enabled, logging manages the output so there is no interleaving. However, in deferred mode the printk behaviour is changed since the output is delayed until the logging thread processes the data. `CONFIG_LOG_PRINTK` is enabled by default.

4.12.5 Architecture

Logging consists of 3 main parts:

- Frontend
- Core
- Backends

Log message is generated by a source of logging which can be a module or instance of a module.

Default Frontend

Default frontend is engaged when the logging API is called in a source of logging (e.g. *LOG_INF*) and is responsible for filtering a message (compile and run time), allocating a buffer for the message, creating the message and committing that message. Since the logging API can be called in an interrupt, the frontend is optimized to log the message as fast as possible.

Log message A log message contains a message descriptor (source, domain and level), timestamp, formatted string details (see *Cbprintf Packaging*) and optional data. Log messages are stored in a continuous block of memory. Memory is allocated from a circular packet buffer (*Multi Producer Single Consumer Packet Buffer*), which has a few consequences:

- Each message is a self-contained, continuous block of memory thus it is suited for copying the message (e.g. for offline processing).
- Messages must be sequentially freed. Backend processing is synchronous. Backend can make a copy for deferred processing.

A log message has following format:

Message Header	2 bits: MPSC packet buffer header
	1 bit: Trace/Log message flag
	3 bits: Domain ID
	3 bits: Level
	10 bits: Cbprintf Package Length
	12 bits: Data length
	1 bit: Reserved
	pointer: Pointer to the source descriptor[1]
	32 or 64 bits: Timestamp[1]
	Optional padding[2]
Cbprintf package (optional)	Header
	Arguments
	Appended strings
Hexdump data (optional)	
Alignment padding (optional)	

Log message allocation It may happen that the frontend cannot allocate a message. This happens if the system is generating more log messages than it can process in certain time frame. There are two strategies to handle that case:

- No overflow - the new log is dropped if space for a message cannot be allocated.
- Overflow - the oldest pending messages are freed, until the new message can be allocated. Enabled by `CONFIG_LOG_MODE_OVERFLOW`. Note that it degrades performance thus it is recommended to adjust buffer size and amount of enabled logs to limit dropping.

Run-time filtering If run-time filtering is enabled, then for each source of logging a filter structure in RAM is declared. Such filter is using 32 bits divided into ten 3 bit slots. Except *slot 0*, each slot stores current filter for one backend in the system. *Slot 0* (bits 0-2) is used to aggregate maximal filter setting for given source of logging. Aggregate slot determines if log message is created for given entry since it indicates if there is at least one backend expecting that log entry. Backend slots are examined when message is processed by the core to determine if message is accepted by the given backend. Contrary to compile time filtering, binary footprint is increased because logs are compiled in.

In the example below backend 1 is set to receive errors (*slot 1*) and backend 2 up to info level (*slot 2*). Slots 3-9 are not used. Aggregated filter (*slot 0*) is set to info level and up to this level message from that particular source will be buffered.

[1] Depending on the platform and the timestamp size fields may be swapped.
[2] It may be required for cbprintf package alignment

slot 0	slot 1	slot 2	slot 3	...	slot 9
INF	ERR	INF	OFF	...	OFF

Custom Frontend

Custom frontend is enabled using `CONFIG_LOG_FRONTEND`. Logs are directed to functions declared in include/zephyr/logging/log_frontend.h. If option `CONFIG_LOG_FRONTEND_ONLY` is enabled then log message is not created and no backend is handled. Otherwise, custom frontend can coexist with backends.

In some cases, logs need to be redirected at the macro level. For these cases, `CONFIG_LOG_CUSTOM_HEADER` can be used to inject an application provided header named *zephyr_custom_log.h* at the end of include/zephyr/logging/log.h.

Logging strings

String arguments are handled by *Cbprintf Packaging*. See *Limitations and recommendations* for limitations and recommendations.

Multi-domain support

More complex systems can consist of multiple domains where each domain is an independent binary. Examples of domains are a core in a multicore SoC or one of the binaries (Secure or Nonsecure) on an ARM TrustZone core.

Tracing and debugging on a multi-domain system is more complex and requires an efficient logging system. Two approaches can be used to structure this logging system:

- Log inside each domain independently. This option is not always possible as it requires that each domain has an available backend (for example, UART). This approach can also be troublesome to use and not scalable, as logs are presented on independent outputs.
- Use a multi-domain logging system where log messages from each domain end up in one root domain, where they are processed exactly as in a single domain case. In this approach, log messages are passed between domains using a connection between domains created from the backend on one side and linked to the other.

  The Log link is an interface introduced in this multi-domain approach. The Log link is responsible for receiving any log message from another domain, creating a copy, and putting that local log message copy (including remote data) into the message queue. This specific log link implementation matches the complementary backend implementation to allow log messages exchange and logger control like configuring filtering, getting log source names, and so on.

There are three types of domains in a multi-domain system:

- The *end domain* has the logging core implementation and a cross-domain backend. It can also have other backends in parallel.
- The *relay domain* has one or more links to other domains but does not have backends that output logs to the user. It has a cross-domain backend either to another relay or to the root domain.
- The *root domain* has one or multiple links and a backend that outputs logs to the user.

See the following image for an example of a multi-domain setup:

In this architecture, a link can handle multiple domains. For example, let's consider an SoC with two ARM Cortex-M33 cores with TrustZone: cores A and B (see the example illustrated above).

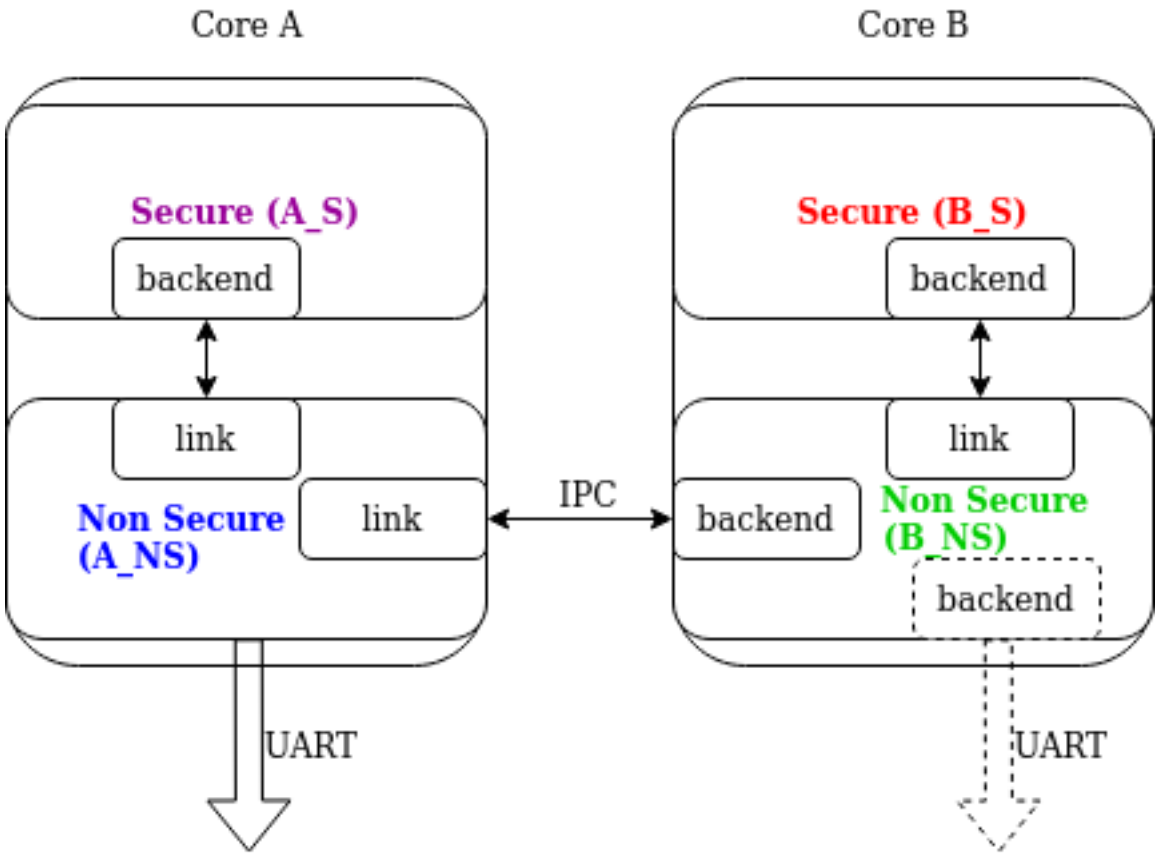


Fig. 5: Multi-domain example

There are four domains in the system, as each core has both a Secure and a Nonsecure domain. If *core A nonsecure* (A_NS) is the root domain, it has two links: one to *core A secure* (A_NS-A_S) and one to *core B nonsecure* (A_NS-B_NS). *B_NS* domain has one link, to *core B secure B_NS-B_S*), and a backend to *A_NS*.

Since in all instances there is a standard logging subsystem, it is always possible to have multiple backends and simultaneously output messages to them. An example of this is shown in the illustration above as a dotted UART backend on the *B_NS* domain.

Domain ID The source of each log message can be identified by the following fields in the header: `source_id` and `domain_id`.

The value assigned to the `domain_id` is relative. Whenever a domain creates a log message, it sets its `domain_id` to `0`. When a message crosses the domain, `domain_id` changes as it is increased by the link offset. The link offset is assigned during the initialization, where the logger core is iterating over all the registered links and assigned offsets.

The first link has the offset set to 1. The following offset equals the previous link offset plus the number of domains in the previous link.

The following example is shown below, where the assigned `domain_ids` are shown for each domain:

Let's consider a log message created on the *B_S* domain:

1. Initially, it has its `domain_id` set to `0`.
2. When the *B_NS-B_S* link receives the message, it increases the `domain_id` to `1` by adding the *B_NS-B_S* offset.
3. The message is passed to *A_NS*.

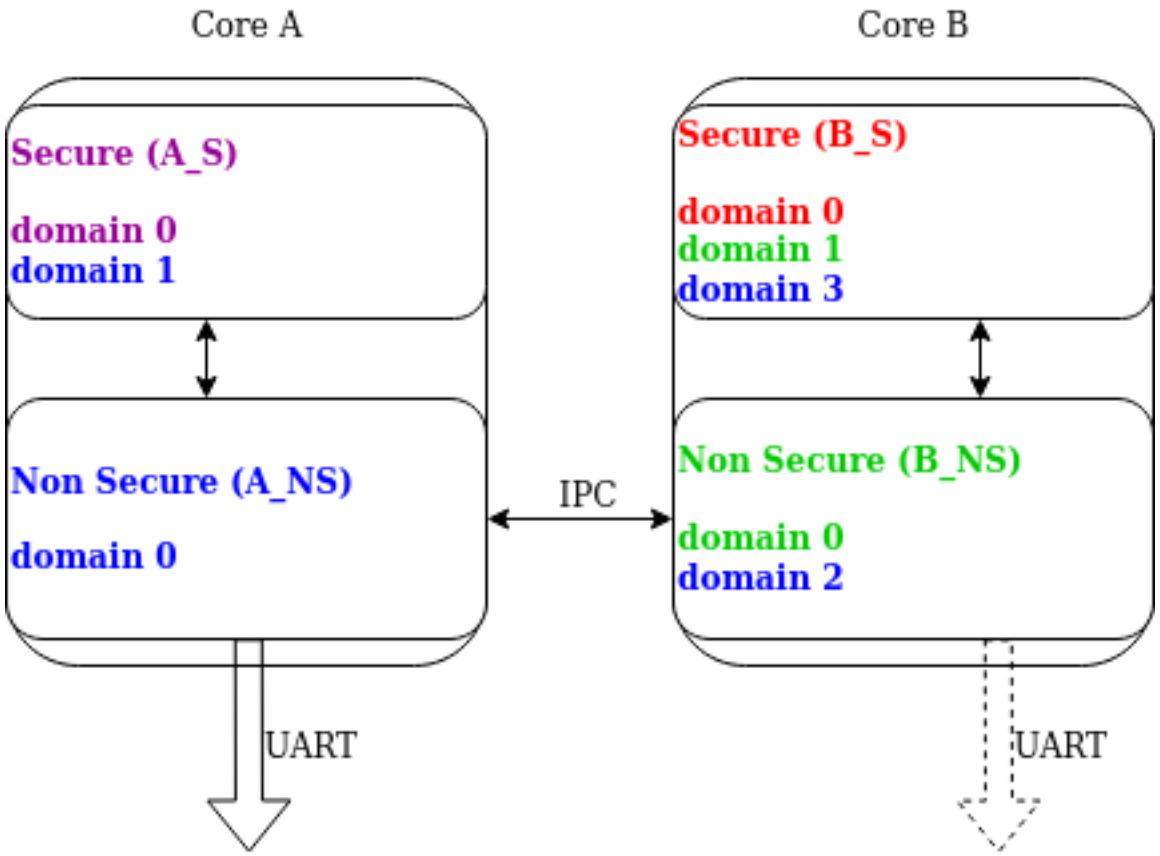


Fig. 6: Domain IDs assigning example

4. When the *A_NS-B_NS* link receives the message, it adds the offset (2) to the `domain_id`. The message ends up with the `domain_id` set to 3, which uniquely identifies the message originator.

Cross-domain log message In most cases, the address space of each domain is unique, and one domain cannot access directly the data in another domain. For this reason, the backend can partially process the message before it is passed to another domain. Partial processing can include converting a string package to a *fully self-contained* version (copying read-only strings to the package body).

Each domain can have a different timestamp source in terms of frequency and offset. Logging does not perform any timestamp conversion.

Runtime filtering In the single-domain case, each log source has a dedicated variable with runtime filtering for each backend in the system. In the multi-domain case, the originator of the log message is not aware of the number of backends in the root domain.

As such, to filter logs in multiple domains, each source requires a runtime filtering setting in each domain on the way to the root domain. As the number of sources in other domains is not known during the compilation, the runtime filtering of remote sources must use dynamically allocated memory (one word per source). When a backend in the root domain changes the filtering of the module from a remote domain, the local filter is updated. After the update, the aggregated filter (the maximum from all the local backends) is checked and, if changed, the remote domain is informed about this change. With this approach, the runtime filtering works identically in both multi-domain and single-domain scenarios.

Message ordering Logging does not provide any mechanism for synchronizing timestamps across multiple domains:

- If domains have different timestamp sources, messages will be processed in the order of arrival to the buffer in the root domain.
- If domains have the same timestamp source or if there is an out-of-bound mechanism that recalculates timestamps, there are 2 options:
  - Messages are processed as they arrive in the buffer in the root domain. Messages are unordered but they can be sorted by the host as the timestamp indicates the time of the message generation.
  - Links have dedicated buffers. During processing, the head of each buffer is checked and the oldest message is processed first.

    With this approach, it is possible to maintain the order of the messages at the cost of a suboptimal memory utilization (since the buffer is not shared) and increased processing latency (see `CONFIG_LOG_PROCESSING_LATENCY_US`).

Logging backends

Logging backends are registered using *`LOG_BACKEND_DEFINE`*. The macro creates an instance in the dedicated memory section. Backends can be dynamically enabled (*`log_backend_enable()`*) and disabled. When *Run-time filtering* is enabled, *`log_filter_set()`* can be used to dynamically change filtering of a module logs for given backend. Module is identified by source ID and domain ID. Source ID can be retrieved if source name is known by iterating through all registered sources.

Logging supports up to 9 concurrent backends. Log message is passed to the each backend in processing phase. Additionally, backend is notified when logging enter panic mode with *`log_backend_panic()`*. On that call backend should switch to synchronous, interrupt-less operation or shut down itself if that is not supported. Occasionally, logging may inform backend about number of dropped messages with *`log_backend_dropped()`*. Message processing API is version specific.

`log_backend_msg_process()` is used for processing message. It is common for standard and hexdump messages because log message hold string with arguments and data. It is also common for deferred and immediate logging.

Message formatting Logging provides set of function that can be used by the backend to format a message. Helper functions are available in include/zephyr/logging/log_output.h.

Example message formatted using *`log_output_msg_process()`*.

```
[00:00:00.000,274] <info> sample_instance.inst1: logging message
```

Dictionary-based Logging

Dictionary-based logging, instead of human readable texts, outputs the log messages in binary format. This binary format encodes arguments to formatted strings in their native storage formats which can be more compact than their text equivalents. For statically defined strings (including the format strings and any string arguments), references to the ELF file are encoded instead of the whole strings. A dictionary created at build time contains the mappings between these references and the actual strings. This allows the offline parser to obtain the strings from the dictionary to parse the log messages. This binary format allows a more compact representation of log messages in certain scenarios. However, this requires the use of an offline parser and is not as intuitive to use as text-based log messages.

Note that `long double` is not supported by Python's `struct` module. Therefore, log messages with `long double` will not display the correct values.

Configuration Here are kconfig options related to dictionary-based logging:

- `CONFIG_LOG_DICTIONARY_SUPPORT` enables dictionary-based logging support. This should be selected by the backends which require it.
- The UART backend can be used for dictionary-based logging. These are additional config for the UART backend:
  - `CONFIG_LOG_BACKEND_UART_OUTPUT_DICTIONARY_HEX` tells the UART backend to output hexadecimal characters for dictionary based logging. This is useful when the log data needs to be captured manually via terminals and consoles.
  - `CONFIG_LOG_BACKEND_UART_OUTPUT_DICTIONARY_BIN` tells the UART backend to output binary data.

Usage When dictionary-based logging is enabled via enabling related logging backends, a JSON database file, named `log_dictionary.json`, will be created in the build directory. This database file contains information for the parser to correctly parse the log data. Note that this database file only works with the same build, and cannot be used for any other builds.

To use the log parser:

```
./scripts/logging/dictionary/log_parser.py <build dir>/log_dictionary.json <log data file>
```

The parser takes two required arguments, where the first one is the full path to the JSON database file, and the second part is the file containing log data. Add an optional argument `--hex` to the end if the log data file contains hexadecimal characters (e.g. when `CONFIG_LOG_BACKEND_UART_OUTPUT_DICTIONARY_HEX=y`). This tells the parser to convert the hexadecimal characters to binary before parsing.

Please refer to the logging-dictionary sample to learn more on how to use the log parser.

4.12.6 Recommendations

The are following recommendations:

- Enable `CONFIG_LOG_SPEED` to slightly speed up deferred logging at the cost of slight increase in memory footprint.
- Compiler with C11 `_Generic` keyword support is recommended. Logging performance is significantly degraded without it. See *Cbprintf Packaging*.
- It is recommended to cast pointer to `const char *` when it is used with `%s` format specifier and it points to a constant string.
- It is recommended to cast pointer to `char *` when it is used with `%s` format specifier and it points to a transient string.
- It is recommended to cast character pointer to non character pointer (e.g., `void *`) when it is used with `%p` format specifier.

```
LOG_WRN("%s", str);
LOG_WRN("%p", (void *)str);
```

4.12.7 Benchmark

Benchmark numbers from tests/subsys/logging/log_benchmark performed on `qemu_x86`. It is a rough comparison to give a general overview.

Feature	
Kernel logging	7us[3]/11us
User logging	13us
kernel logging with overwrite	10us[3]/15us
Logging transient string	42us
Logging transient string from user	50us
Memory utilization[4]	518
Memory footprint (test)[5]	2k
Memory footprint (application)[6]	3.5k
Message footprint[7]	47[3]/32 bytes

Benchmark details

4.12.8 Stack usage

When logging is enabled it impacts stack usage of the context that uses logging API. If stack is optimized it may lead to stack overflow. Stack usage depends on mode and optimization. It also significantly varies between platforms. In general, when `CONFIG_LOG_MODE_DEFERRED` is used stack usage is smaller since logging is limited to creating and storing log message. When `CONFIG_LOG_MODE_IMMEDIATE` is used then log message is processed by the backend which includes string formatting. In case of that mode, stack usage will depend on which backends are used.

tests/subsys/logging/log_stack test is used to characterize stack usage depending on mode, optimization and platform used. Test is using only the default backend.

Some of the platforms characterization for log message with two `integer` arguments listed below:

Platform	Deferred	Deferred (no optimization)	Immediate	Immediate (no optimization)
ARM Cortex-M3	40	152	412	783
x86	12	224	388	796
riscv32	24	208	456	844
xtensa	72	336	504	944
x86_64	32	528	1088	1440

4.12.9 API Reference

[3] `CONFIG_LOG_SPEED` enabled.
[4] Number of log messages with various number of arguments that fits in 2048 bytes dedicated for logging.
[5] Logging subsystem memory footprint in tests/subsys/logging/log_benchmark where filtering and formatting features are not used.
[6] Logging subsystem memory footprint in samples/subsys/logging/logger.
[7] Average size of a log message (excluding string) with 2 arguments on `Cortex M3`

Logger API

Related code samples

BLE logging backend
Send log messages over BLE using the BLE logging backend.

Dictionary-based logging
Output binary log data using the dictionary-based logging API.

Logging
Output log messages to the console using the logging subsystem.

group `log_api`

Logger API.

Defines

`LOG_ERR`(...)

Writes an ERROR level message to the log.

It's meant to report severe errors, such as those from which it's not possible to recover.

Parameters

- ... – A string optionally containing printk valid conversion specifier, followed by as many values as specifiers.

`LOG_WRN`(...)

Writes a WARNING level message to the log.

It's meant to register messages related to unusual situations that are not necessarily errors.

Parameters

- ... – A string optionally containing printk valid conversion specifier, followed by as many values as specifiers.

`LOG_INF`(...)

Writes an INFO level message to the log.

It's meant to write generic user oriented messages.

Parameters

- ... – A string optionally containing printk valid conversion specifier, followed by as many values as specifiers.

`LOG_DBG`(...)

Writes a DEBUG level message to the log.

It's meant to write developer oriented information.

Parameters

- ... – A string optionally containing printk valid conversion specifier, followed by as many values as specifiers.

LOG_WRN_ONCE(...)

Writes a WARNING level message to the log on the first execution only.

It's meant for situations that warrant investigation but could clutter the logs if output on every execution.

Parameters

- ... – A string optionally containing printk valid conversion specifier, followed by as many values as specifiers.

LOG_PRINTK(...)

Unconditionally print raw log message.

The result is same as if printk was used but it goes through logging infrastructure thus utilizes logging mode, e.g. deferred mode.

Parameters

- ... – A string optionally containing printk valid conversion specifier, followed by as many values as specifiers.

LOG_RAW(...)

Unconditionally print raw log message.

Provided string is printed as is without appending any characters (e.g., color or newline).

Parameters

- ... – A string optionally containing printk valid conversion specifier, followed by as many values as specifiers.

LOG_INST_ERR(_log_inst, ...)

Writes an ERROR level message associated with the instance to the log.

Message is associated with specific instance of the module which has independent filtering settings (if runtime filtering is enabled) and message prefix (<module_name>.<instance_name>). It's meant to report severe errors, such as those from which it's not possible to recover.

Parameters

- **_log_inst** – Pointer to the log structure associated with the instance.
- ... – A string optionally containing printk valid conversion specifier, followed by as many values as specifiers.

LOG_INST_WRN(_log_inst, ...)

Writes a WARNING level message associated with the instance to the log.

Message is associated with specific instance of the module which has independent filtering settings (if runtime filtering is enabled) and message prefix (<module_name>.<instance_name>). It's meant to register messages related to unusual situations that are not necessarily errors.

Parameters

- **_log_inst** – Pointer to the log structure associated with the instance.
- ... – A string optionally containing printk valid conversion specifier, followed by as many values as specifiers.

LOG_INST_INF(_log_inst, ...)

Writes an INFO level message associated with the instance to the log.

Message is associated with specific instance of the module which has independent filtering settings (if runtime filtering is enabled) and message prefix (<module_name>.<instance_name>). It's meant to write generic user oriented messages.

Parameters

- **_log_inst** – Pointer to the log structure associated with the instance.
- **...** – A string optionally containing printk valid conversion specifier, followed by as many values as specifiers.

LOG_INST_DBG(_log_inst, ...)

Writes a DEBUG level message associated with the instance to the log.

Message is associated with specific instance of the module which has independent filtering settings (if runtime filtering is enabled) and message prefix (<module_name>.<instance_name>). It's meant to write developer oriented information.

Parameters

- **_log_inst** – Pointer to the log structure associated with the instance.
- **...** – A string optionally containing printk valid conversion specifier, followed by as many values as specifiers.

LOG_HEXDUMP_ERR(_data, _length, _str)

Writes an ERROR level hexdump message to the log.

It's meant to report severe errors, such as those from which it's not possible to recover.

Parameters

- **_data** – Pointer to the data to be logged.
- **_length** – Length of data (in bytes).
- **_str** – Persistent, raw string.

LOG_HEXDUMP_WRN(_data, _length, _str)

Writes a WARNING level message to the log.

It's meant to register messages related to unusual situations that are not necessarily errors.

Parameters

- **_data** – Pointer to the data to be logged.
- **_length** – Length of data (in bytes).
- **_str** – Persistent, raw string.

LOG_HEXDUMP_INF(_data, _length, _str)

Writes an INFO level message to the log.

It's meant to write generic user oriented messages.

Parameters

- **_data** – Pointer to the data to be logged.
- **_length** – Length of data (in bytes).
- **_str** – Persistent, raw string.

LOG_HEXDUMP_DBG(_data, _length, _str)

Writes a DEBUG level message to the log.

It's meant to write developer oriented information.

Parameters

- **_data** – Pointer to the data to be logged.
- **_length** – Length of data (in bytes).

- **_str** – Persistent, raw string.

LOG_INST_HEXDUMP_ERR(_log_inst, _data, _length, _str)

Writes an ERROR hexdump message associated with the instance to the log.

Message is associated with specific instance of the module which has independent filtering settings (if runtime filtering is enabled) and message prefix (<module_name>.<instance_name>). It's meant to report severe errors, such as those from which it's not possible to recover.

Parameters

- **_log_inst** – Pointer to the log structure associated with the instance.
- **_data** – Pointer to the data to be logged.
- **_length** – Length of data (in bytes).
- **_str** – Persistent, raw string.

LOG_INST_HEXDUMP_WRN(_log_inst, _data, _length, _str)

Writes a WARNING level hexdump message associated with the instance to the log.

It's meant to register messages related to unusual situations that are not necessarily errors.

Parameters

- **_log_inst** – Pointer to the log structure associated with the instance.
- **_data** – Pointer to the data to be logged.
- **_length** – Length of data (in bytes).
- **_str** – Persistent, raw string.

LOG_INST_HEXDUMP_INF(_log_inst, _data, _length, _str)

Writes an INFO level hexdump message associated with the instance to the log.

It's meant to write generic user oriented messages.

Parameters

- **_log_inst** – Pointer to the log structure associated with the instance.
- **_data** – Pointer to the data to be logged.
- **_length** – Length of data (in bytes).
- **_str** – Persistent, raw string.

LOG_INST_HEXDUMP_DBG(_log_inst, _data, _length, _str)

Writes a DEBUG level hexdump message associated with the instance to the log.

It's meant to write developer oriented information.

Parameters

- **_log_inst** – Pointer to the log structure associated with the instance.
- **_data** – Pointer to the data to be logged.
- **_length** – Length of data (in bytes).
- **_str** – Persistent, raw string.

LOG_MODULE_REGISTER(...)

Create module-specific state and register the module with Logger.

This macro normally must be used after including <zephyr/logging/log.h> to complete the initialization of the module.

Module registration can be skipped in two cases:

- The module consists of more than one file, and another file invokes this macro. (*LOG_MODULE_DECLARE()* should be used instead in all of the module's other files.)
- Instance logging is used and there is no need to create module entry. In that case *LOG_LEVEL_SET()* should be used to set log level used within the file.

Macro accepts one or two parameters:

- module name
- optional log level. If not provided then default log level is used in the file.

Example usage:

- *LOG_MODULE_REGISTER(foo, CONFIG_FOO_LOG_LEVEL)*
- *LOG_MODULE_REGISTER(foo)*

See also

LOG_MODULE_DECLARE

Note

The module's state is defined, and the module is registered, only if LOG_LEVEL for the current source file is non-zero or it is not defined and CONFIG_LOG_DEFAULT_LEVEL is non-zero. In other cases, this macro has no effect.

`LOG_MODULE_DECLARE(...)`

Macro for declaring a log module (not registering it).

Modules which are split up over multiple files must have exactly one file use *LOG_MODULE_REGISTER()* to create module-specific state and register the module with the logger core.

The other files in the module should use this macro instead to declare that same state. (Otherwise, *LOG_INF()* etc. will not be able to refer to module-specific state variables.)

Macro accepts one or two parameters:

- module name
- optional log level. If not provided then default log level is used in the file.

Example usage:

- *LOG_MODULE_DECLARE(foo, CONFIG_FOO_LOG_LEVEL)*
- *LOG_MODULE_DECLARE(foo)*

See also

LOG_MODULE_REGISTER

Note

The module's state is declared only if LOG_LEVEL for the current source file is non-zero or it is not defined and CONFIG_LOG_DEFAULT_LEVEL is non-zero. In other cases, this macro has no effect.

`LOG_LEVEL_SET`(level)

Macro for setting log level in the file or function where instance logging API is used.

Parameters

- `level` – Level used in file or in function.

Logger control

Related code samples

Logging
Output log messages to the console using the logging subsystem.

Remote syslog
Enable a remote syslog service that sends syslog messages to a remote server

group `log_ctrl`

Logger control API.

Since
1.13

Defines

`LOG_CORE_INIT`()

`LOG_INIT`()

`LOG_PANIC`()

`LOG_PROCESS`()

Typedefs

typedef log_timestamp_t (*`log_timestamp_get_t`)(void)

Functions

void `log_core_init`(void)

Function system initialization of the logger.

Function is called during start up to allow logging before user can explicitly initialize the logger.

void **log_init**(void)

Function for user initialization of the logger.

void **log_thread_trigger**(void)

Trigger the log processing thread to process logs immediately.

> **Note**
>
> Function has no effect when CONFIG_LOG_MODE_IMMEDIATE is set.

void **log_thread_set**(k_tid_t process_tid)

Function for providing thread which is processing logs.

See CONFIG_LOG_PROCESS_TRIGGER_THRESHOLD.

> **Note**
>
> Function has asserts and has no effect when CONFIG_LOG_PROCESS_THREAD is set.

Parameters

- **process_tid** – Process thread id. Used to wake up the thread.

int **log_set_timestamp_func**(*log_timestamp_get_t* timestamp_getter, uint32_t freq)

Function for providing timestamp function.

Parameters

- **timestamp_getter** – Timestamp function.
- **freq** – Timestamping frequency.

Returns

0 on success or error.

void **log_panic**(void)

Switch the logger subsystem to the panic mode.

Returns immediately if the logger is already in the panic mode.

On panic the logger subsystem informs all backends about panic mode. Backends must switch to blocking mode or halt. All pending logs are flushed after switching to panic mode. In panic mode, all log messages must be processed in the context of the call.

bool **log_process**(void)

Process one pending log message.

Return values

- **true** – There are more messages pending to be processed.
- **false** – No messages pending.

uint32_t **log_buffered_cnt**(void)

Return number of buffered log messages.

Returns

Number of currently buffered log messages.

uint32_t **log_src_cnt_get**(uint32_t domain_id)

Get number of independent logger sources (modules and instances)

Parameters

- **domain_id** – Domain ID.

Returns
Number of sources.

const char *log_source_name_get(uint32_t domain_id, uint32_t source_id)

Get name of the source (module or instance).

Parameters

- **domain_id** – Domain ID.
- **source_id** – Source ID.

Returns
Source name or NULL if invalid arguments.

static inline uint8_t log_domains_count(void)

Return number of domains present in the system.

There will be at least one local domain.

Returns
Number of domains.

const char *log_domain_name_get(uint32_t domain_id)

Get name of the domain.

Parameters

- **domain_id** – Domain ID.

Returns
Domain name.

int log_source_id_get(const char *name)

Function for finding source ID based on source name.

Parameters

- **name** – Source name

Returns
Source ID or negative number when source ID is not found.

uint32_t log_filter_get(struct *log_backend* const *const backend, uint32_t domain_id, int16_t source_id, bool runtime)

Get source filter for the provided backend.

Parameters

- **backend** – Backend instance.
- **domain_id** – ID of the domain.
- **source_id** – Source (module or instance) ID.
- **runtime** – True for runtime filter or false for compiled in.

Returns
Severity level.

uint32_t log_filter_set(struct *log_backend* const *const backend, uint32_t domain_id, int16_t source_id, uint32_t level)

Set filter on given source for the provided backend.

Parameters

- **backend** – Backend instance. NULL for all backends (and frontend).
- **domain_id** – ID of the domain.

- `source_id` – Source (module or instance) ID.
- `level` – Severity level.

Returns
Actual level set which may be limited by compiled level. If filter was set for all backends then maximal level that was set is returned.

uint32_t `log_frontend_filter_get`(int16_t source_id, bool runtime)

Get source filter for the frontend.

Parameters

- `source_id` – Source (module or instance) ID.
- `runtime` – True for runtime filter or false for compiled in.

Returns
Severity level.

uint32_t `log_frontend_filter_set`(int16_t source_id, uint32_t level)

Set filter on given source for the frontend.

Parameters

- `source_id` – Source (module or instance) ID.
- `level` – Severity level.

Returns
Actual level set which may be limited by compiled level.

void `log_backend_enable`(struct *log_backend* const *const backend, void *ctx, uint32_t level)

Enable backend with initial maximum filtering level.

Parameters

- `backend` – Backend instance.
- `ctx` – User context.
- `level` – Severity level.

void `log_backend_disable`(struct *log_backend* const *const backend)

Disable backend.

Parameters

- `backend` – Backend instance.

const struct *log_backend* *`log_backend_get_by_name`(const char *backend_name)

Get backend by name.

Parameters

- `backend_name` – **[in]** Name of the backend as defined by the LOG_BACKEND_DEFINE.

Return values
`Pointer` – to the backend instance if found, NULL if backend is not found.

const struct *log_backend* *`log_format_set_all_active_backends`(size_t log_type)

Sets logging format for all active backends.

Parameters

- `log_type` – Log format.

Return values
`Pointer` – to the last backend that failed, NULL for success.

static inline bool `log_data_pending`(void)

Check if there is pending data to be processed by the logging subsystem.

Function can be used to determine if all logs have been flushed. Function returns false when deferred mode is not enabled.

Return values

- `true` – There is pending data.
- `false` – No pending data to process.

int `log_set_tag`(const char *tag)

Configure tag used to prefix each message.

Parameters

- `tag` – Tag.

Return values

- `0` – on successful operation.
- `-ENOTSUP` – if feature is disabled.
- `-ENOMEM` – if string is longer than the buffer capacity. Tag will be trimmed.

int `log_mem_get_usage`(uint32_t *buf_size, uint32_t *usage)

Get current memory usage.

Parameters

- `buf_size` – **[out]** Capacity of the buffer used for storing log messages.
- `usage` – **[out]** Number of bytes currently containing pending log messages.

Return values

- `-EINVAL` – if logging mode does not use the buffer.
- `0` – successfully collected usage data.

int `log_mem_get_max_usage`(uint32_t *max)

Get maximum memory usage.

Requires CONFIG_LOG_MEM_UTILIZATION option.

Parameters

- `max` – **[out]** Maximum number of bytes used for pending log messages.

Return values

- `-EINVAL` – if logging mode does not use the buffer.
- `-ENOTSUP` – if instrumentation is not enabled. not been enabled.
- `0` – successfully collected usage data.

Log message

group `log_msg`

Log message API.

Defines

LOG_MSG_GENERIC_HDR

LOG_MSG_SIMPLE_ARG_CNT_CHECK(...)

LOG_MSG_SIMPLE_ARG_TYPE_CHECK_0(fmt)

LOG_MSG_SIMPLE_ARG_TYPE_CHECK_1(fmt, arg)

LOG_MSG_SIMPLE_ARG_TYPE_CHECK_2(fmt, arg0, arg1)

LOG_MSG_SIMPLE_ARG_TYPE_CHECK(...)

brief Determine if string arguments types allow to use simplified message creation mode.

Parameters

- ... – String with arguments.

LOG_MSG_SIMPLE_CHECK(...)

Check if message can be handled using simplified method.

Following conditions must be met:

- 32 bit platform
- Number of arguments from 0 to 2
- Type of an argument must be a numeric value that fits in 32 bit word.

Parameters

- ... – String with arguments.

Return values

- 1 – if message qualifies.
- 0 – if message does not qualify.

LOG_MSG_SIMPLE_FUNC(_source, _level, ...)

Call specific function to create a log message.

Macro picks matching function (based on number of arguments) and calls it. String arguments are casted to uint32_t.

Parameters

- **_source** – Source.
- **_level** – Severity level.
- ... – String with arguments.

Functions

static inline uint32_t **log_msg_get_total_wlen**(const struct *log_msg_desc* desc)

Get total length (in 32 bit words) of a log message.

Parameters

- **desc** – Log message descriptor.

Returns

Length.

static inline uint32_t **log_msg_generic_get_wlen**(const union mpsc_pbuf_generic *item)

Get length of the log item.

Parameters

- **item** – Item.

Returns

Length in 32 bit words.

static inline uint8_t **log_msg_get_domain**(struct *log_msg* *msg)

Get log message domain ID.

Parameters

- **msg** – Log message.

Returns

Domain ID

static inline uint8_t **log_msg_get_level**(struct *log_msg* *msg)

Get log message level.

Parameters

- **msg** – Log message.

Returns

Log level.

static inline const void ***log_msg_get_source**(struct *log_msg* *msg)

Get message source data.

Parameters

- **msg** – Log message.

Returns

Pointer to the source data.

int16_t **log_msg_get_source_id**(struct *log_msg* *msg)

Get log message source ID.

Parameters

- **msg** – Log message.

Returns

Source ID, or -1 if not available.

static inline log_timestamp_t **log_msg_get_timestamp**(struct *log_msg* *msg)

Get timestamp.

Parameters

- **msg** – Log message.

Returns

Timestamp.

static inline void ***log_msg_get_tid**(struct *log_msg* *msg)

Get Thread ID.

Parameters

- **msg** – Log message.

Returns

Thread ID.

static inline uint8_t *log_msg_get_data(struct *log_msg* *msg, size_t *len)

Get data buffer.

Parameters

- msg – log message.
- len – location where data length is written.

Returns

pointer to the data buffer.

static inline uint8_t *log_msg_get_package(struct *log_msg* *msg, size_t *len)

Get string package.

Parameters

- msg – log message.
- len – location where string package length is written.

Returns

pointer to the package.

struct log_msg_desc

#include <log_msg.h>

union log_msg_source

#include <log_msg.h>

Public Members

const struct log_source_const_data *fixed

struct log_source_dynamic_data *dynamic

void *raw

struct log_msg_hdr

#include <log_msg.h>

struct log_msg

#include <log_msg.h>

struct log_msg_generic_hdr

#include <log_msg.h>

union log_msg_generic

#include <log_msg.h>

Public Members

union mpsc_pbuf_generic buf

struct *log_msg_generic_hdr* `generic`

struct *log_msg* `log`

Logger backend interface

Related code samples

BLE logging backend
Send log messages over BLE using the BLE logging backend.

Remote syslog
Enable a remote syslog service that sends syslog messages to a remote server

group `log_backend`

Logger backend interface.

Defines

`LOG_BACKEND_DEFINE`(_name, _api, _autostart, ...)

Macro for creating a logger backend instance.

Parameters

- `_name` – Name of the backend instance.
- `_api` – Logger backend API.
- `_autostart` – If true backend is initialized and activated together with the logger subsystem.
- `...` – Optional context.

Enums

enum `log_backend_evt`

Backend events.

Values:

enumerator `LOG_BACKEND_EVT_PROCESS_THREAD_DONE`

Event when process thread finishes processing.

This event is emitted when the process thread finishes processing pending log messages.

Note

This is not emitted when there are no pending log messages being processed.

> **Note**
>
> Deferred mode only.

enumerator LOG_BACKEND_EVT_MAX

Maximum number of backend events.

Functions

static inline void **log_backend_init**(const struct *log_backend* *const backend)

Initialize or initiate the logging backend.

If backend initialization takes longer time it could block logging thread if backend is autostarted. That is because all backends are initialized in the context of the logging thread. In that case, backend shall provide function for polling for readiness (*log_backend_is_ready*).

Parameters

- backend – **[in]** Pointer to the backend instance.

static inline int **log_backend_is_ready**(const struct *log_backend* *const backend)

Poll for backend readiness.

If backend is ready immediately after initialization then backend may not provide this function.

Parameters

- backend – **[in]** Pointer to the backend instance.

Return values

- 0 – if backend is ready.
- -EBUSY – if backend is not yet ready.

static inline void **log_backend_msg_process**(const struct *log_backend* *const backend, union *log_msg_generic* *msg)

Process message.

Function is used in deferred and immediate mode. On return, message content is processed by the backend and memory can be freed.

Parameters

- backend – **[in]** Pointer to the backend instance.
- msg – **[in]** Pointer to message with log entry.

static inline void **log_backend_dropped**(const struct *log_backend* *const backend, uint32_t cnt)

Notify backend about dropped log messages.

Function is optional.

Parameters

- backend – **[in]** Pointer to the backend instance.
- cnt – **[in]** Number of dropped logs since last notification.

static inline void **log_backend_panic**(const struct *log_backend* *const backend)

Reconfigure backend to panic mode.

Parameters

- **backend** – **[in]** Pointer to the backend instance.

static inline void **log_backend_id_set**(const struct *log_backend* *const backend, uint8_t id)

Set backend id.

> **Note**
>
> It is used internally by the logger.

Parameters

- **backend** – Pointer to the backend instance.
- **id** – ID.

static inline uint8_t **log_backend_id_get**(const struct *log_backend* *const backend)

Get backend id.

> **Note**
>
> It is used internally by the logger.

Parameters

- **backend** – **[in]** Pointer to the backend instance.

Returns

Id.

static inline const struct *log_backend* ***log_backend_get**(uint32_t idx)

Get backend.

Parameters

- **idx** – **[in]** Pointer to the backend instance.

Returns

Pointer to the backend instance.

static inline int **log_backend_count_get**(void)

Get number of backends.

Returns

Number of backends.

static inline void **log_backend_activate**(const struct *log_backend* *const backend, void *ctx)

Activate backend.

Parameters

- **backend** – **[in]** Pointer to the backend instance.
- **ctx** – **[in]** User context.

static inline void **log_backend_deactivate**(const struct *log_backend* *const backend)

Deactivate backend.

Parameters

- **backend** – **[in]** Pointer to the backend instance.

static inline bool **log_backend_is_active**(const struct *log_backend* *const backend)

Check state of the backend.

Parameters

- **backend** – **[in]** Pointer to the backend instance.

Returns

True if backend is active, false otherwise.

static inline int **log_backend_format_set**(const struct *log_backend* *backend, uint32_t log_type)

Set logging format.

Parameters

- **backend** – Pointer to the backend instance.
- **log_type** – Log format.

Return values

- **-ENOTSUP** – If the backend does not support changing format types.
- **-EINVAL** – If the input is invalid.
- **0** – for success.

static inline void **log_backend_notify**(const struct *log_backend* *const backend, enum *log_backend_evt* event, union *log_backend_evt_arg* *arg)

Notify a backend of an event.

Parameters

- **backend** – Pointer to the backend instance.
- **event** – Event to be notified.
- **arg** – Pointer to the argument(s).

union **log_backend_evt_arg**

#include <*log_backend.h*> Argument(s) for backend events.

Public Members

void ***raw**

Unspecified argument(s).

struct **log_backend_api**

#include <*log_backend.h*> Logger backend API.

struct **log_backend_control_block**

#include <*log_backend.h*> Logger backend control block.

struct `log_backend`

#include <log_backend.h> Logger backend structure.

Logger output formatting

group `log_output`

Log output API.

Unnamed Group

void `log_custom_output_msg_process`(const struct *log_output* *log_output, struct *log_msg* *msg, uint32_t flags)

Custom logging output formatting.

Process log messages from an external output function set with log_custom_output_msg_set

Function is using provided context with the buffer and output function to process formatted string and output the data.

Parameters

- `log_output` – Pointer to the log output instance.
- `msg` – Log message.
- `flags` – Optional flags.

Defines

`LOG_OUTPUT_TEXT`

Supported backend logging format types for use with log_format_set() API to switch log format at runtime.

`LOG_OUTPUT_SYST`

`LOG_OUTPUT_DICT`

`LOG_OUTPUT_CUSTOM`

`LOG_OUTPUT_DEFINE`(_name, _func, _buf, _size)

Create *log_output* instance.

Parameters

- `_name` – Instance name.
- `_func` – Function for processing output data.
- `_buf` – Pointer to the output buffer.
- `_size` – Size of the output buffer.

Typedefs

typedef int (*`log_output_func_t`)(uint8_t *buf, size_t size, void *ctx)

Prototype of the function processing output data.

> **Note**
>
> If the log output function cannot process all of the data, it is its responsibility to mark them as dropped or discarded by returning the corresponding number of bytes dropped or discarded to the caller.

Param buf
The buffer data.

Param size
The buffer size.

Param ctx
User context.

Return
Number of bytes processed, dropped or discarded.

typedef void (*`log_format_func_t`)(const struct *log_output* *output, struct *log_msg* *msg, uint32_t flags)

Typedef of the function pointer table “format_table”.

Param output
Pointer to *log_output* struct.

Param msg
Pointer to *log_msg* struct.

Param flags
Flags used for text formatting options.

Return
Function pointer based on Kconfigs defined for backends.

Functions

log_format_func_t `log_format_func_t_get`(uint32_t log_type)

Declaration of the get routine for function pointer table format_table.

void `log_output_msg_process`(const struct *log_output* *log_output, struct *log_msg* *msg, uint32_t flags)

Process log messages v2 to readable strings.

Function is using provided context with the buffer and output function to process formatted string and output the data.

Parameters

- `log_output` – Pointer to the log output instance.
- `msg` – Log message.
- `flags` – Optional flags. See Log output formatting flags..

void **log_output_process**(const struct *log_output* *log_output, log_timestamp_t timestamp, const char *domain, const char *source, k_tid_t tid, uint8_t level, const uint8_t *package, const uint8_t *data, size_t data_len, uint32_t flags)

Process input data to a readable string.

Parameters

- **log_output** – Pointer to the log output instance.
- **timestamp** – Timestamp.
- **domain** – Domain name string. Can be NULL.
- **source** – Source name string. Can be NULL.
- **tid** – Thread ID.
- **level** – Criticality level.
- **package** – Cbprintf package with a logging message string.
- **data** – Data passed to hexdump API. Can be NULL.
- **data_len** – Data length.
- **flags** – Formatting flags. See Log output formatting flags..

void **log_output_msg_syst_process**(const struct *log_output* *log_output, struct *log_msg* *msg, uint32_t flags)

Process log messages v2 to SYS-T format.

Function is using provided context with the buffer and output function to process formatted string and output the data in sys-t log output format.

Parameters

- **log_output** – Pointer to the log output instance.
- **msg** – Log message.
- **flags** – Optional flags. See Log output formatting flags..

void **log_output_dropped_process**(const struct *log_output* *output, uint32_t cnt)

Process dropped messages indication.

Function prints error message indicating lost log messages.

Parameters

- **output** – Pointer to the log output instance.
- **cnt** – Number of dropped messages.

void **log_output_flush**(const struct *log_output* *output)

Flush output buffer.

Parameters

- **output** – Pointer to the log output instance.

static inline void **log_output_ctx_set**(const struct *log_output* *output, void *ctx)

Function for setting user context passed to the output function.

Parameters

- **output** – Pointer to the log output instance.
- **ctx** – User context.

static inline void **log_output_hostname_set**(const struct *log_output* *output, const char *hostname)

Function for setting hostname of this device.

Parameters

- **output** – Pointer to the log output instance.
- **hostname** – Hostname of this device

void **log_output_timestamp_freq_set**(uint32_t freq)

Set timestamp frequency.

Parameters

- **freq** – Frequency in Hz.

uint64_t **log_output_timestamp_to_us**(log_timestamp_t timestamp)

Convert timestamp of the message to us.

Parameters

- **timestamp** – Message timestamp

Returns

Timestamp value in us.

struct **log_output_control_block**

#include <*log_output.h*>

struct **log_output**

#include <*log_output.h*> Log_output instance structure.

4.13 Tracing

4.13.1 Overview

The tracing feature provides hooks that permits you to collect data from your application and allows tools running on a host to visualize the inner-working of the kernel and various subsystems.

Every system has application-specific events to trace out. Historically, that has implied:

1. Determining the application-specific payload,
2. Choosing suitable serialization-format,
3. Writing the on-target serialization code,
4. Deciding on and writing the I/O transport mechanics,
5. Writing the PC-side deserializer/parser,
6. Writing custom ad-hoc tools for filtering and presentation.

An application can use one of the existing formats or define a custom format by overriding the macros declared in include/zephyr/tracing/tracing.h.

Different formats, transports and host tools are available and supported in Zephyr.

In fact, I/O varies greatly from system to system. Therefore, it is instructive to create a taxonomy for I/O types when we must ensure the interface between payload/format (Top Layer) and the transport mechanics (bottom Layer) is generic and efficient enough to model these. See the *I/O taxonomy* section below.

4.13.2 Serialization Formats

Common Trace Format (CTF) Support

Common Trace Format, CTF, is an open format and language to describe trace formats. This enables tool reuse, of which line-textual (babeltrace) and graphical (TraceCompass) variants already exist.

CTF should look familiar to C programmers but adds stronger typing. See CTF - A Flexible, High-performance Binary Trace Format.

CTF allows us to formally describe application specific payload and the serialization format, which enables common infrastructure for host tools and parsers and tools for filtering and presentation.

A Generic Interface In CTF, an event is serialized to a packet containing one or more fields. As seen from *I/O taxonomy* section below, a bottom layer may:

- perform actions at transaction-start (e.g. mutex-lock),
- process each field in some way (e.g. sync-push emit, concat, enqueue to thread-bound FIFO),
- perform actions at transaction-stop (e.g. mutex-release, emit of concat buffer).

CTF Top-Layer Example The CTF_EVENT macro will serialize each argument to a field:

```
/* Example for illustration */
static inline void ctf_top_foo(uint32_t thread_id, ctf_bounded_string_t name)
{
  CTF_EVENT(
    CTF_LITERAL(uint8_t, 42),
    thread_id,
    name,
    "hello, I was emitted from function: ",
    __func__  /* __func__ is standard since C99 */
  );
}
```

How to serialize and emit fields as well as handling alignment, can be done internally and statically at compile-time in the bottom-layer.

The CTF top layer is enabled using the configuration option `CONFIG_TRACING_CTF` and can be used with the different transport backends both in synchronous and asynchronous modes.

SEGGER SystemView Support

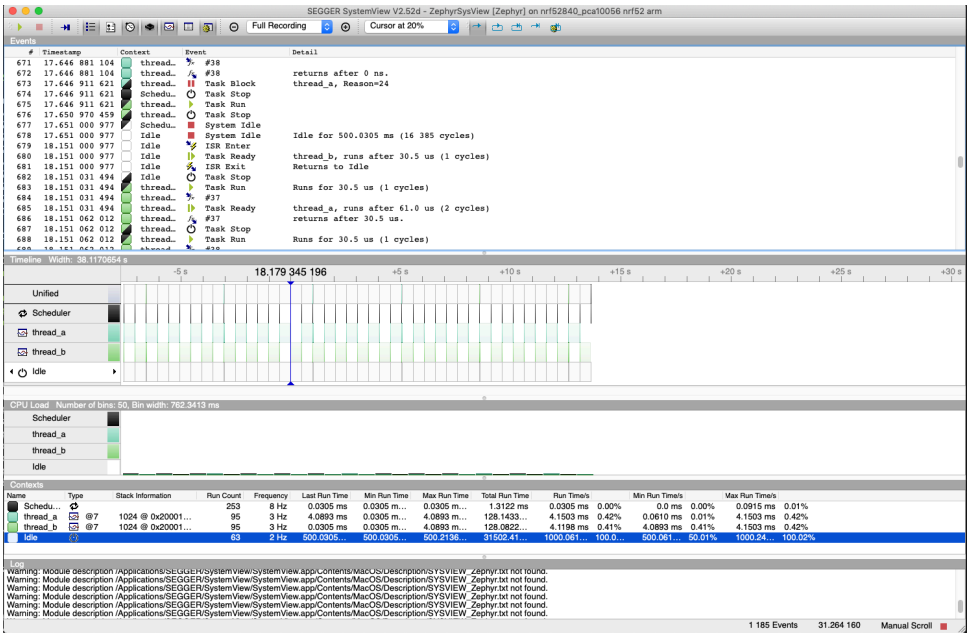
Zephyr provides built-in support for SEGGER SystemView that can be enabled in any application for platforms that have the required hardware support.

The payload and format used with SystemView is custom to the application and relies on RTT as a transport. Newer versions of SystemView support other transports, for example UART or using snapshot mode (both still not supported in Zephyr).

To enable tracing support with SEGGER SystemView add the configuration option `CONFIG_SEGGER_SYSTEMVIEW` to your project configuration file and set it to *y*. For example, this can be added to the synchronization sample to visualize fast switching between threads. SystemView can also be used for post-mortem tracing, which can be enabled with *CONFIG_SEGGER_SYSVIEW_POST_MORTEM_MODE*. In this mode, a debugger can be attached after

the system has crashed using `west attach` after which the latest data from the internal RAM buffer can be loaded into SystemView:

```
CONFIG_STDOUT_CONSOLE=y
# enable to use thread names
CONFIG_THREAD_NAME=y
CONFIG_SEGGER_SYSTEMVIEW=y
CONFIG_USE_SEGGER_RTT=y
CONFIG_TRACING=y
# enable for post-mortem tracing
CONFIG_SEGGER_SYSVIEW_POST_MORTEM_MODE=n
```



Recent versions of SEGGER SystemView come with an API translation table for Zephyr which is incomplete and does not match the current level of support available in Zephyr. To use the latest Zephyr API description table, copy the file available in the tree to your local configuration directory to override the builtin table:

```
# On Linux and MacOS
cp $ZEPHYR_BASE/subsys/tracing/sysview/SYSVIEW_Zephyr.txt ~/.config/SEGGER/
```

User-Defined Tracing

This tracing format allows the user to define functions to perform any work desired when a task is switched in or out, when an interrupt is entered or exited, and when the cpu is idle.

Examples include: - simple toggling of GPIO for external scope tracing while minimizing extra cpu load - generating/outputting trace data in a non-standard or proprietary format that can not be supported by the other tracing systems

The following functions can be defined by the user:

```
void sys_trace_thread_create_user(struct k_thread *thread);
void sys_trace_thread_abort_user(struct k_thread *thread);
void sys_trace_thread_suspend_user(struct k_thread *thread);
void sys_trace_thread_resume_user(struct k_thread *thread);
void sys_trace_thread_name_set_user(struct k_thread *thread);
void sys_trace_thread_switched_in_user(struct k_thread *thread);
void sys_trace_thread_switched_out_user(struct k_thread *thread);
```

(continues on next page)

(continued from previous page)

```
void sys_trace_thread_info_user(struct k_thread *thread);
void sys_trace_thread_sched_ready_user(struct k_thread *thread);
void sys_trace_thread_pend_user(struct k_thread *thread);
void sys_trace_thread_priority_set_user(struct k_thread *thread, int prio);
void sys_trace_isr_enter_user(int nested_interrupts);
void sys_trace_isr_exit_user(int nested_interrupts);
void sys_trace_idle_user();
```

Enable this format with the CONFIG_TRACING_USER option.

4.13.3 Transport Backends

The following backends are currently supported:

- UART
- USB
- File (Using the native port with POSIX architecture based targets)
- RTT (With SystemView)
- RAM (buffer to be retrieved by a debugger)

4.13.4 Using Tracing

The sample samples/subsys/tracing demonstrates tracing with different formats and backends.

To get started, the simplest way is to use the CTF format with the native_sim port, build the sample as follows:

Using west:

```
west build -b native_sim samples/subsys/tracing -- -DCONF_FILE=prj_native_ctf.conf
```

Using CMake and ninja:

```
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=native_sim -DCONF_FILE=prj_native_ctf.conf samples/subsys/
↪tracing

# Now run the build tool on the generated build system:
ninja -Cbuild
```

You can then run the resulting binary with the option `-trace-file` to generate the tracing data:

```
mkdir data
cp $ZEPHYR_BASE/subsys/tracing/ctf/tsdl/metadata data/
./build/zephyr/zephyr.exe -trace-file=data/channel0_0
```

The resulting CTF output can be visualized using babeltrace or TraceCompass by pointing the tool to the data directory with the metadata and trace files.

Using RAM backend

For devices that do not have available I/O for tracing such as USB or UART but have enough RAM to collect trace data, the ram backend can be enabled with configuration CONFIG_TRACING_BACKEND_RAM. Adjust CONFIG_RAM_TRACING_BUFFER_SIZE to be able to record enough

traces for your needs. Then thanks to a runtime debugger such as gdb this buffer can be fetched from the target to an host computer:

```
(gdb) dump binary memory data/channel0_0 <ram_tracing_start> <ram_tracing_end>
```

The resulting channel0_0 file have to be placed in a directory with the `metadata` file like the other backend.

4.13.5 Visualisation Tools

TraceCompass

TraceCompass is an open source tool that visualizes CTF events such as thread scheduling and interrupts, and is helpful to find unintended interactions and resource conflicts on complex systems.

See also the presentation by Ericsson, Advanced Trouble-shooting Of Real-time Systems.

4.13.6 Future LTTng Inspiration

Currently, the top-layer provided here is quite simple and bare-bones, and needlessly copied from Zephyr's Segger SystemView debug module.

For an OS like Zephyr, it would make sense to draw inspiration from Linux's LTTng and change the top-layer to serialize to the same format. Doing this would enable direct reuse of TraceCompass' canned analyses for Linux. Alternatively, LTTng-analyses in TraceCompass could be customized to Zephyr. It is ongoing work to enable TraceCompass visibility of Zephyr in a target-agnostic and open source way.

I/O Taxonomy

- Atomic Push/Produce/Write/Enqueue:
  - **synchronous:**
    means data-transmission has completed with the return of the call.
  - **asynchronous:**
    means data-transmission is pending or ongoing with the return of the call. Usually, interrupts/callbacks/signals or polling is used to determine completion.
  - **buffered:**
    means data-transmissions are copied and grouped together to form a larger ones. Usually for amortizing overhead (burst dequeue) or jitter-mitigation (steady dequeue).

  Examples:

  - **sync unbuffered**
    E.g. PIO via GPIOs having steady stream, no extra FIFO memory needed. Low jitter but may be less efficient (can't amortize the overhead of writing).
  - **sync buffered**
    E.g. `fwrite()` or enqueuing into FIFO. Blockingly burst the FIFO when its buffer-waterlevel exceeds threshold. Jitter due to bursts may lead to missed deadlines.
  - **async unbuffered**
    E.g. DMA, or zero-copying in shared memory. Be careful of data hazards, race conditions, etc!

  - **async buffered**
    E.g. enqueuing into FIFO.

- Atomic Pull/Consume/Read/Dequeue:
  - **synchronous:**
    means data-reception has completed with the return of the call.
  - **asynchronous:**
    means data-reception is pending or ongoing with the return of the call. Usually, interrupts/callbacks/signals or polling is used to determine completion.
  - **buffered:**
    means data is copied-in in larger chunks than request-size. Usually for amortizing wait-time.

  Examples:

  - **sync unbuffered**
    E.g. Blocking read-call, `fread()` or SPI-read, zero-copying in shared memory.
  - **sync buffered**
    E.g. Blocking read-call with caching applied. Makes sense if read pattern exhibits spatial locality.
  - **async unbuffered**
    E.g. zero-copying in shared memory. Be careful of data hazards, race conditions, etc!
  - **async buffered**
    E.g. `aio_read()` or DMA.

Unfortunately, I/O may not be atomic and may, therefore, require locking. Locking may not be needed if multiple independent channels are available.

- **The system has non-atomic write and one shared channel**
  E.g. UART. Locking required.

  ```
  lock(); emit(a); emit(b); emit(c); release();
  ```

- **The system has non-atomic write but many channels**
  E.g. Multi-UART. Lock-free if the bottom-layer maps each Zephyr thread+ISR to its own channel, thus alleviating races as each thread is sequentially consistent with itself.

  ```
  emit(a,thread_id); emit(b,thread_id); emit(c,thread_id);
  ```

- **The system has atomic write but one shared channel**
  E.g. `native_sim` or board with DMA. May or may not need locking.

  ```
  emit(a ## b ## c); /* Concat to buffer */

  lock(); emit(a); emit(b); emit(c); release(); /* No extra mem */
  ```

- **The system has atomic write and many channels**
  E.g. native_sim or board with multi-channel DMA. Lock-free.

  ```
  emit(a ## b ## c, thread_id);
  ```

4.13.7 Object tracking

The kernel can also maintain lists of objects that can be used to track their usage. Currently, the following lists can be enabled:

```
struct k_timer *_track_list_k_timer;
struct k_mem_slab *_track_list_k_mem_slab;
struct k_sem *_track_list_k_sem;
```

(continues on next page)

(continued from previous page)

```
struct k_mutex *_track_list_k_mutex;
struct k_stack *_track_list_k_stack;
struct k_msgq *_track_list_k_msgq;
struct k_mbox *_track_list_k_mbox;
struct k_pipe *_track_list_k_pipe;
struct k_queue *_track_list_k_queue;
struct k_event *_track_list_k_event;
```

Those global variables are the head of each list - they can be traversed with the help of macro `SYS_PORT_TRACK_NEXT`. For instance, to traverse all initialized mutexes, one can write:

```
struct k_mutex *cur = _track_list_k_mutex;
while (cur != NULL) {
  /* Do something */

  cur = SYS_PORT_TRACK_NEXT(cur);
}
```

To enable object tracking, enable `CONFIG_TRACING_OBJECT_TRACKING`. Note that each list can be enabled or disabled via their tracing configuration. For example, to disable tracking of semaphores, one can disable `CONFIG_TRACING_SEMAPHORE`.

Object tracking is behind tracing configuration as it currently leverages tracing infrastructure to perform the tracking.

4.13.8 API

Common

group **subsys_tracing_apis**

Tracing APIs.

Defines

sys_trace_sys_init_enter(entry, level)

Called when entering an init function.

sys_trace_sys_init_exit(entry, level, result)

Called when exiting an init function.

Functions

void **sys_trace_isr_enter**(void)

Called when entering an ISR.

void **sys_trace_isr_exit**(void)

Called when exiting an ISR.

void **sys_trace_isr_exit_to_scheduler**(void)

Called when exiting an ISR and switching to scheduler.

void **sys_trace_idle**(void)

Called when the cpu enters the idle state.

Threads

group `subsys_tracing_apis_thread`

Thread Tracing APIs.

Defines

`sys_port_trace_k_thread_foreach_enter`()

Called when entering a k_thread_foreach call.

`sys_port_trace_k_thread_foreach_exit`()

Called when exiting a k_thread_foreach call.

`sys_port_trace_k_thread_foreach_unlocked_enter`()

Called when entering a k_thread_foreach_unlocked.

`sys_port_trace_k_thread_foreach_unlocked_exit`()

Called when exiting a k_thread_foreach_unlocked.

`sys_port_trace_k_thread_create`(new_thread)

Trace creating a Thread.

Parameters

- `new_thread` – Thread object

`sys_port_trace_k_thread_user_mode_enter`()

Trace Thread entering user mode.

`sys_port_trace_k_thread_join_enter`(thread, timeout)

Called when entering a k_thread_join.

Parameters

- `thread` – Thread object
- `timeout` – Timeout period

`sys_port_trace_k_thread_join_blocking`(thread, timeout)

Called when k_thread_join blocks.

Parameters

- `thread` – Thread object
- `timeout` – Timeout period

`sys_port_trace_k_thread_join_exit`(thread, timeout, ret)

Called when exiting k_thread_join.

Parameters

- `thread` – Thread object
- `timeout` – Timeout period
- `ret` – Return value

`sys_port_trace_k_thread_sleep_enter`(timeout)

Called when entering k_thread_sleep.

Parameters

- `timeout` – Timeout period

`sys_port_trace_k_thread_sleep_exit`(timeout, ret)
Called when exiting k_thread_sleep.

Parameters

- `timeout` – Timeout period
- `ret` – Return value

`sys_port_trace_k_thread_msleep_enter`(ms)
Called when entering k_thread_msleep.

Parameters

- `ms` – Duration in milliseconds

`sys_port_trace_k_thread_msleep_exit`(ms, ret)
Called when exiting k_thread_msleep.

Parameters

- `ms` – Duration in milliseconds
- `ret` – Return value

`sys_port_trace_k_thread_usleep_enter`(us)
Called when entering k_thread_usleep.

Parameters

- `us` – Duration in microseconds

`sys_port_trace_k_thread_usleep_exit`(us, ret)
Called when exiting k_thread_usleep.

Parameters

- `us` – Duration in microseconds
- `ret` – Return value

`sys_port_trace_k_thread_busy_wait_enter`(usec_to_wait)
Called when entering k_thread_busy_wait.

Parameters

- `usec_to_wait` – Duration in microseconds

`sys_port_trace_k_thread_busy_wait_exit`(usec_to_wait)
Called when exiting k_thread_busy_wait.

Parameters

- `usec_to_wait` – Duration in microseconds

`sys_port_trace_k_thread_yield`()
Called when a thread yields.

`sys_port_trace_k_thread_wakeup`(thread)
Called when a thread wakes up.

Parameters

- `thread` – Thread object

`sys_port_trace_k_thread_start`(thread)
Called when a thread is started.

Parameters

- `thread` – Thread object

sys_port_trace_k_thread_abort(thread)
Called when a thread is being aborted.

Parameters

- thread – Thread object

sys_port_trace_k_thread_abort_enter(thread)
Called when a thread enters the k_thread_abort routine.

Parameters

- thread – Thread object

sys_port_trace_k_thread_abort_exit(thread)
Called when a thread exits the k_thread_abort routine.

Parameters

- thread – Thread object

sys_port_trace_k_thread_priority_set(thread)
Called when setting priority of a thread.

Parameters

- thread – Thread object

sys_port_trace_k_thread_suspend_enter(thread)
Called when a thread enters the k_thread_suspend function.

Parameters

- thread – Thread object

sys_port_trace_k_thread_suspend_exit(thread)
Called when a thread exits the k_thread_suspend function.

Parameters

- thread – Thread object

sys_port_trace_k_thread_resume_enter(thread)
Called when a thread enters the resume from suspension function.

Parameters

- thread – Thread object

sys_port_trace_k_thread_resume_exit(thread)
Called when a thread exits the resumed from suspension function.

Parameters

- thread – Thread object

sys_port_trace_k_thread_sched_lock()
Called when the thread scheduler is locked.

sys_port_trace_k_thread_sched_unlock()
Called when the thread scheduler is unlocked.

sys_port_trace_k_thread_name_set(thread, ret)
Called when a thread name is set.

Parameters

- thread – Thread object
- ret – Return value

sys_port_trace_k_thread_switched_out()
Called before a thread has been selected to run.

sys_port_trace_k_thread_switched_in()
Called after a thread has been selected to run.

sys_port_trace_k_thread_ready(thread)
Called when a thread is ready to run.

Parameters

- thread – Thread object

sys_port_trace_k_thread_pend(thread)
Called when a thread is pending.

Parameters

- thread – Thread object

sys_port_trace_k_thread_info(thread)
Provide information about specific thread.

Parameters

- thread – Thread object

sys_port_trace_k_thread_sched_wakeup(thread)
Trace implicit thread wakeup invocation by the scheduler.

Parameters

- thread – Thread object

sys_port_trace_k_thread_sched_abort(thread)
Trace implicit thread abort invocation by the scheduler.

Parameters

- thread – Thread object

sys_port_trace_k_thread_sched_priority_set(thread, prio)
Trace implicit thread set priority invocation by the scheduler.

Parameters

- thread – Thread object
- prio – Thread priority

sys_port_trace_k_thread_sched_ready(thread)
Trace implicit thread ready invocation by the scheduler.

Parameters

- thread – Thread object

sys_port_trace_k_thread_sched_pend(thread)
Trace implicit thread pend invocation by the scheduler.

Parameters

- thread – Thread object

sys_port_trace_k_thread_sched_resume(thread)
Trace implicit thread resume invocation by the scheduler.

Parameters

- thread – Thread object

`sys_port_trace_k_thread_sched_suspend`(thread)
Trace implicit thread suspend invocation by the scheduler.

Parameters

- `thread` – Thread object

Work Queues

group `subsys_tracing_apis_work`
Work Tracing APIs.

Defines

`sys_port_trace_k_work_init`(work)
Trace initialisation of a Work structure.

Parameters

- `work` – Work structure

`sys_port_trace_k_work_submit_to_queue_enter`(queue, work)
Trace submit work to work queue call entry.

Parameters

- `queue` – Work queue structure
- `work` – Work structure

`sys_port_trace_k_work_submit_to_queue_exit`(queue, work, ret)
Trace submit work to work queue call exit.

Parameters

- `queue` – Work queue structure
- `work` – Work structure
- `ret` – Return value

`sys_port_trace_k_work_submit_enter`(work)
Trace submit work to system work queue call entry.

Parameters

- `work` – Work structure

`sys_port_trace_k_work_submit_exit`(work, ret)
Trace submit work to system work queue call exit.

Parameters

- `work` – Work structure
- `ret` – Return value

`sys_port_trace_k_work_flush_enter`(work)
Trace flush work call entry.

Parameters

- `work` – Work structure

`sys_port_trace_k_work_flush_blocking`(work, timeout)
Trace flush work call blocking.

Parameters

- `work` – Work structure
- `timeout` – Timeout period

`sys_port_trace_k_work_flush_exit`(work, ret)
Trace flush work call exit.

Parameters

- `work` – Work structure
- `ret` – Return value

`sys_port_trace_k_work_cancel_enter`(work)
Trace cancel work call entry.

Parameters

- `work` – Work structure

`sys_port_trace_k_work_cancel_exit`(work, ret)
Trace cancel work call exit.

Parameters

- `work` – Work structure
- `ret` – Return value

`sys_port_trace_k_work_cancel_sync_enter`(work, sync)
Trace cancel sync work call entry.

Parameters

- `work` – Work structure
- `sync` – Sync object

`sys_port_trace_k_work_cancel_sync_blocking`(work, sync)
Trace cancel sync work call blocking.

Parameters

- `work` – Work structure
- `sync` – Sync object

`sys_port_trace_k_work_cancel_sync_exit`(work, sync, ret)
Trace cancel sync work call exit.

Parameters

- `work` – Work structure
- `sync` – Sync object
- `ret` – Return value

Poll

group `subsys_tracing_apis_poll`
Poll Tracing APIs.

Defines

`sys_port_trace_k_poll_api_event_init`(event)
Trace initialisation of a Poll Event.

Parameters

- `event` – Poll Event

`sys_port_trace_k_poll_api_poll_enter`(events)
Trace Polling call start.

Parameters

- `events` – Poll Events

`sys_port_trace_k_poll_api_poll_exit`(events, ret)
Trace Polling call outcome.

Parameters

- `events` – Poll Events
- `ret` – Return value

`sys_port_trace_k_poll_api_signal_init`(signal)
Trace initialisation of a Poll Signal.

Parameters

- `signal` – Poll Signal

`sys_port_trace_k_poll_api_signal_reset`(signal)
Trace resetting of Poll Signal.

Parameters

- `signal` – Poll Signal

`sys_port_trace_k_poll_api_signal_check`(signal)
Trace checking of Poll Signal.

Parameters

- `signal` – Poll Signal

`sys_port_trace_k_poll_api_signal_raise`(signal, ret)
Trace raising of Poll Signal.

Parameters

- `signal` – Poll Signal
- `ret` – Return value

Semaphore

group `subsys_tracing_apis_sem`
Semaphore Tracing APIs.

Defines

`sys_port_trace_k_sem_init`(sem, ret)
Trace initialisation of a Semaphore.

Parameters

- `sem` – Semaphore object
- `ret` – Return value

`sys_port_trace_k_sem_give_enter`(sem)
Trace giving a Semaphore entry.

Parameters

- `sem` – Semaphore object

`sys_port_trace_k_sem_give_exit`(sem)
Trace giving a Semaphore exit.

Parameters

- `sem` – Semaphore object

`sys_port_trace_k_sem_take_enter`(sem, timeout)
Trace taking a Semaphore attempt start.

Parameters

- `sem` – Semaphore object
- `timeout` – Timeout period

`sys_port_trace_k_sem_take_blocking`(sem, timeout)
Trace taking a Semaphore attempt blocking.

Parameters

- `sem` – Semaphore object
- `timeout` – Timeout period

`sys_port_trace_k_sem_take_exit`(sem, timeout, ret)
Trace taking a Semaphore attempt outcome.

Parameters

- `sem` – Semaphore object
- `timeout` – Timeout period
- `ret` – Return value

`sys_port_trace_k_sem_reset`(sem)
Trace resetting a Semaphore.

Parameters

- `sem` – Semaphore object

Mutex

group `subsys_tracing_apis_mutex`
Mutex Tracing APIs.

Defines

`sys_port_trace_k_mutex_init`(mutex, ret)
Trace initialization of Mutex.

Parameters

- `mutex` – Mutex object
- `ret` – Return value

`sys_port_trace_k_mutex_lock_enter`(mutex, timeout)
Trace Mutex lock attempt start.

Parameters

- `mutex` – Mutex object
- `timeout` – Timeout period

`sys_port_trace_k_mutex_lock_blocking`(mutex, timeout)
Trace Mutex lock attempt blocking.

Parameters

- `mutex` – Mutex object
- `timeout` – Timeout period

`sys_port_trace_k_mutex_lock_exit`(mutex, timeout, ret)
Trace Mutex lock attempt outcome.

Parameters

- `mutex` – Mutex object
- `timeout` – Timeout period
- `ret` – Return value

`sys_port_trace_k_mutex_unlock_enter`(mutex)
Trace Mutex unlock entry.

Parameters

- `mutex` – Mutex object

`sys_port_trace_k_mutex_unlock_exit`(mutex, ret)
Trace Mutex unlock exit.

Condition Variables

group `subsys_tracing_apis_condvar`
Conditional Variable Tracing APIs.

Defines

`sys_port_trace_k_condvar_init`(condvar, ret)
Trace initialization of Conditional Variable.

Parameters

- `condvar` – Conditional Variable object
- `ret` – Return value

`sys_port_trace_k_condvar_signal_enter`(condvar)
Trace Conditional Variable signaling start.

Parameters

- `condvar` – Conditional Variable object

`sys_port_trace_k_condvar_signal_blocking`(condvar, timeout)
Trace Conditional Variable signaling blocking.

Parameters

- `condvar` – Conditional Variable object
- `timeout` – Timeout period

`sys_port_trace_k_condvar_signal_exit`(condvar, ret)
Trace Conditional Variable signaling outcome.

Parameters

- `condvar` – Conditional Variable object
- `ret` – Return value

`sys_port_trace_k_condvar_broadcast_enter`(condvar)
Trace Conditional Variable broadcast enter.

Parameters

- `condvar` – Conditional Variable object

`sys_port_trace_k_condvar_broadcast_exit`(condvar, ret)
Trace Conditional Variable broadcast exit.

Parameters

- `condvar` – Conditional Variable object
- `ret` – Return value

`sys_port_trace_k_condvar_wait_enter`(condvar)
Trace Conditional Variable wait enter.

Parameters

- `condvar` – Conditional Variable object

`sys_port_trace_k_condvar_wait_exit`(condvar, ret)
Trace Conditional Variable wait exit.

Parameters

- `condvar` – Conditional Variable object
- `ret` – Return value

Queues

group `subsys_tracing_apis_queue`
Queue Tracing APIs.

Defines

sys_port_trace_k_queue_init(queue)
Trace initialization of Queue.

Parameters

- queue – Queue object

sys_port_trace_k_queue_cancel_wait(queue)
Trace Queue cancel wait.

Parameters

- queue – Queue object

sys_port_trace_k_queue_queue_insert_enter(queue, alloc)
Trace Queue insert attempt entry.

Parameters

- queue – Queue object
- alloc – Allocation flag

sys_port_trace_k_queue_queue_insert_blocking(queue, alloc, timeout)
Trace Queue insert attempt blocking.

Parameters

- queue – Queue object
- alloc – Allocation flag
- timeout – Timeout period

sys_port_trace_k_queue_queue_insert_exit(queue, alloc, ret)
Trace Queue insert attempt outcome.

Parameters

- queue – Queue object
- alloc – Allocation flag
- ret – Return value

sys_port_trace_k_queue_append_enter(queue)
Trace Queue append enter.

Parameters

- queue – Queue object

sys_port_trace_k_queue_append_exit(queue)
Trace Queue append exit.

Parameters

- queue – Queue object

sys_port_trace_k_queue_alloc_append_enter(queue)
Trace Queue alloc append enter.

Parameters

- queue – Queue object

sys_port_trace_k_queue_alloc_append_exit(queue, ret)
Trace Queue alloc append exit.

Parameters

- queue – Queue object
- ret – Return value

sys_port_trace_k_queue_prepend_enter(queue)
Trace Queue prepend enter.

Parameters

- queue – Queue object

sys_port_trace_k_queue_prepend_exit(queue)
Trace Queue prepend exit.

Parameters

- queue – Queue object

sys_port_trace_k_queue_alloc_prepend_enter(queue)
Trace Queue alloc prepend enter.

Parameters

- queue – Queue object

sys_port_trace_k_queue_alloc_prepend_exit(queue, ret)
Trace Queue alloc prepend exit.

Parameters

- queue – Queue object
- ret – Return value

sys_port_trace_k_queue_insert_enter(queue)
Trace Queue insert attempt entry.

Parameters

- queue – Queue object

sys_port_trace_k_queue_insert_blocking(queue, timeout)
Trace Queue insert attempt blocking.

Parameters

- queue – Queue object
- timeout – Timeout period

sys_port_trace_k_queue_insert_exit(queue)
Trace Queue insert attempt exit.

Parameters

- queue – Queue object

sys_port_trace_k_queue_append_list_enter(queue)
Trace Queue append list enter.

Parameters

- queue – Queue object

sys_port_trace_k_queue_append_list_exit(queue, ret)
Trace Queue append list exit.

Parameters

- queue – Queue object
- ret – Return value

sys_port_trace_k_queue_merge_slist_enter(queue)
Trace Queue merge slist enter.

Parameters

- queue – Queue object

sys_port_trace_k_queue_merge_slist_exit(queue, ret)
Trace Queue merge slist exit.

Parameters

- queue – Queue object
- ret – Return value

sys_port_trace_k_queue_get_enter(queue, timeout)
Trace Queue get attempt enter.

Parameters

- queue – Queue object
- timeout – Timeout period

sys_port_trace_k_queue_get_blocking(queue, timeout)
Trace Queue get attempt blockings.

Parameters

- queue – Queue object
- timeout – Timeout period

sys_port_trace_k_queue_get_exit(queue, timeout, ret)
Trace Queue get attempt outcome.

Parameters

- queue – Queue object
- timeout – Timeout period
- ret – Return value

sys_port_trace_k_queue_remove_enter(queue)
Trace Queue remove enter.

Parameters

- queue – Queue object

sys_port_trace_k_queue_remove_exit(queue, ret)
Trace Queue remove exit.

Parameters

- queue – Queue object
- ret – Return value

`sys_port_trace_k_queue_unique_append_enter`(queue)
Trace Queue unique append enter.

Parameters

- `queue` – Queue object

`sys_port_trace_k_queue_unique_append_exit`(queue, ret)
Trace Queue unique append exit.

Parameters

- `queue` – Queue object
- `ret` – Return value

`sys_port_trace_k_queue_peek_head`(queue, ret)
Trace Queue peek head.

Parameters

- `queue` – Queue object
- `ret` – Return value

`sys_port_trace_k_queue_peek_tail`(queue, ret)
Trace Queue peek tail.

Parameters

- `queue` – Queue object
- `ret` – Return value

FIFO

group `subsys_tracing_apis_fifo`
FIFO Tracing APIs.

Defines

`sys_port_trace_k_fifo_init_enter`(fifo)
Trace initialization of FIFO Queue entry.

Parameters

- `fifo` – FIFO object

`sys_port_trace_k_fifo_init_exit`(fifo)
Trace initialization of FIFO Queue exit.

Parameters

- `fifo` – FIFO object

`sys_port_trace_k_fifo_cancel_wait_enter`(fifo)
Trace FIFO Queue cancel wait entry.

Parameters

- `fifo` – FIFO object

sys_port_trace_k_fifo_cancel_wait_exit(fifo)
Trace FIFO Queue cancel wait exit.

Parameters

- fifo – FIFO object

sys_port_trace_k_fifo_put_enter(fifo, data)
Trace FIFO Queue put entry.

Parameters

- fifo – FIFO object
- data – Data item

sys_port_trace_k_fifo_put_exit(fifo, data)
Trace FIFO Queue put exit.

Parameters

- fifo – FIFO object
- data – Data item

sys_port_trace_k_fifo_alloc_put_enter(fifo, data)
Trace FIFO Queue alloc put entry.

Parameters

- fifo – FIFO object
- data – Data item

sys_port_trace_k_fifo_alloc_put_exit(fifo, data, ret)
Trace FIFO Queue alloc put exit.

Parameters

- fifo – FIFO object
- data – Data item
- ret – Return value

sys_port_trace_k_fifo_put_list_enter(fifo, head, tail)
Trace FIFO Queue put list entry.

Parameters

- fifo – FIFO object
- head – First ll-node
- tail – Last ll-node

sys_port_trace_k_fifo_put_list_exit(fifo, head, tail)
Trace FIFO Queue put list exit.

Parameters

- fifo – FIFO object
- head – First ll-node
- tail – Last ll-node

sys_port_trace_k_fifo_alloc_put_slist_enter(fifo, list)
Trace FIFO Queue put slist entry.

Parameters

- **fifo** – FIFO object
- **list** – Syslist object

sys_port_trace_k_fifo_alloc_put_slist_exit(fifo, list)
Trace FIFO Queue put slist exit.

Parameters

- **fifo** – FIFO object
- **list** – Syslist object

sys_port_trace_k_fifo_get_enter(fifo, timeout)
Trace FIFO Queue get entry.

Parameters

- **fifo** – FIFO object
- **timeout** – Timeout period

sys_port_trace_k_fifo_get_exit(fifo, timeout, ret)
Trace FIFO Queue get exit.

Parameters

- **fifo** – FIFO object
- **timeout** – Timeout period
- **ret** – Return value

sys_port_trace_k_fifo_peek_head_enter(fifo)
Trace FIFO Queue peek head entry.

Parameters

- **fifo** – FIFO object

sys_port_trace_k_fifo_peek_head_exit(fifo, ret)
Trace FIFO Queue peek head exit.

Parameters

- **fifo** – FIFO object
- **ret** – Return value

sys_port_trace_k_fifo_peek_tail_enter(fifo)
Trace FIFO Queue peek tail entry.

Parameters

- **fifo** – FIFO object

sys_port_trace_k_fifo_peek_tail_exit(fifo, ret)
Trace FIFO Queue peek tail exit.

Parameters

- **fifo** – FIFO object
- **ret** – Return value

LIFO

group subsys_tracing_apis_lifo
LIFO Tracing APIs.

Defines

sys_port_trace_k_lifo_init_enter(lifo)
Trace initialization of LIFO Queue entry.

Parameters

- lifo – LIFO object

sys_port_trace_k_lifo_init_exit(lifo)
Trace initialization of LIFO Queue exit.

Parameters

- lifo – LIFO object

sys_port_trace_k_lifo_put_enter(lifo, data)
Trace LIFO Queue put entry.

Parameters

- lifo – LIFO object
- data – Data item

sys_port_trace_k_lifo_put_exit(lifo, data)
Trace LIFO Queue put exit.

Parameters

- lifo – LIFO object
- data – Data item

sys_port_trace_k_lifo_alloc_put_enter(lifo, data)
Trace LIFO Queue alloc put entry.

Parameters

- lifo – LIFO object
- data – Data item

sys_port_trace_k_lifo_alloc_put_exit(lifo, data, ret)
Trace LIFO Queue alloc put exit.

Parameters

- lifo – LIFO object
- data – Data item
- ret – Return value

sys_port_trace_k_lifo_get_enter(lifo, timeout)
Trace LIFO Queue get entry.

Parameters

- lifo – LIFO object
- timeout – Timeout period

sys_port_trace_k_lifo_get_exit(lifo, timeout, ret)
Trace LIFO Queue get exit.

Parameters

- lifo – LIFO object
- timeout – Timeout period

- **ret** – Return value

Stacks

group subsys_tracing_apis_stack

Stack Tracing APIs.

Defines

sys_port_trace_k_stack_init(stack)

Trace initialization of Stack.

Parameters

- **stack** – Stack object

sys_port_trace_k_stack_alloc_init_enter(stack)

Trace Stack alloc init attempt entry.

Parameters

- **stack** – Stack object

sys_port_trace_k_stack_alloc_init_exit(stack, ret)

Trace Stack alloc init outcome.

Parameters

- **stack** – Stack object
- **ret** – Return value

sys_port_trace_k_stack_cleanup_enter(stack)

Trace Stack cleanup attempt entry.

Parameters

- **stack** – Stack object

sys_port_trace_k_stack_cleanup_exit(stack, ret)

Trace Stack cleanup outcome.

Parameters

- **stack** – Stack object
- **ret** – Return value

sys_port_trace_k_stack_push_enter(stack)

Trace Stack push attempt entry.

Parameters

- **stack** – Stack object

sys_port_trace_k_stack_push_exit(stack, ret)

Trace Stack push attempt outcome.

Parameters

- **stack** – Stack object
- **ret** – Return value

`sys_port_trace_k_stack_pop_enter`(stack, timeout)
Trace Stack pop attempt entry.

Parameters

- `stack` – Stack object
- `timeout` – Timeout period

`sys_port_trace_k_stack_pop_blocking`(stack, timeout)
Trace Stack pop attempt blocking.

Parameters

- `stack` – Stack object
- `timeout` – Timeout period

`sys_port_trace_k_stack_pop_exit`(stack, timeout, ret)
Trace Stack pop attempt outcome.

Parameters

- `stack` – Stack object
- `timeout` – Timeout period
- `ret` – Return value

Message Queues

group `subsys_tracing_apis_msgq`
Message Queue Tracing APIs.

Defines

`sys_port_trace_k_msgq_init`(msgq)
Trace initialization of Message Queue.

Parameters

- `msgq` – Message Queue object

`sys_port_trace_k_msgq_alloc_init_enter`(msgq)
Trace Message Queue alloc init attempt entry.

Parameters

- `msgq` – Message Queue object

`sys_port_trace_k_msgq_alloc_init_exit`(msgq, ret)
Trace Message Queue alloc init attempt outcome.

Parameters

- `msgq` – Message Queue object
- `ret` – Return value

`sys_port_trace_k_msgq_cleanup_enter`(msgq)
Trace Message Queue cleanup attempt entry.

Parameters

- `msgq` – Message Queue object

sys_port_trace_k_msgq_cleanup_exit(msgq, ret)
Trace Message Queue cleanup attempt outcome.

Parameters

- msgq – Message Queue object
- ret – Return value

sys_port_trace_k_msgq_put_enter(msgq, timeout)
Trace Message Queue put attempt entry.

Parameters

- msgq – Message Queue object
- timeout – Timeout period

sys_port_trace_k_msgq_put_blocking(msgq, timeout)
Trace Message Queue put attempt blocking.

Parameters

- msgq – Message Queue object
- timeout – Timeout period

sys_port_trace_k_msgq_put_exit(msgq, timeout, ret)
Trace Message Queue put attempt outcome.

Parameters

- msgq – Message Queue object
- timeout – Timeout period
- ret – Return value

sys_port_trace_k_msgq_get_enter(msgq, timeout)
Trace Message Queue get attempt entry.

Parameters

- msgq – Message Queue object
- timeout – Timeout period

sys_port_trace_k_msgq_get_blocking(msgq, timeout)
Trace Message Queue get attempt blockings.

Parameters

- msgq – Message Queue object
- timeout – Timeout period

sys_port_trace_k_msgq_get_exit(msgq, timeout, ret)
Trace Message Queue get attempt outcome.

Parameters

- msgq – Message Queue object
- timeout – Timeout period
- ret – Return value

sys_port_trace_k_msgq_peek(msgq, ret)
Trace Message Queue peek.

Parameters

- **msgq** – Message Queue object
- **ret** – Return value

sys_port_trace_k_msgq_purge(msgq)

Trace Message Queue purge.

Parameters

- **msgq** – Message Queue object

Mailbox

group subsys_tracing_apis_mbox

Mailbox Tracing APIs.

Defines

sys_port_trace_k_mbox_init(mbox)

Trace initialization of Mailbox.

Parameters

- **mbox** – Mailbox object

sys_port_trace_k_mbox_message_put_enter(mbox, timeout)

Trace Mailbox message put attempt entry.

Parameters

- **mbox** – Mailbox object
- **timeout** – Timeout period

sys_port_trace_k_mbox_message_put_blocking(mbox, timeout)

Trace Mailbox message put attempt blocking.

Parameters

- **mbox** – Mailbox object
- **timeout** – Timeout period

sys_port_trace_k_mbox_message_put_exit(mbox, timeout, ret)

Trace Mailbox message put attempt outcome.

Parameters

- **mbox** – Mailbox object
- **timeout** – Timeout period
- **ret** – Return value

sys_port_trace_k_mbox_put_enter(mbox, timeout)

Trace Mailbox put attempt entry.

Parameters

- **mbox** – Mailbox object
- **timeout** – Timeout period

`sys_port_trace_k_mbox_put_exit`(mbox, timeout, ret)
Trace Mailbox put attempt blocking.

Parameters

- `mbox` – Mailbox object
- `timeout` – Timeout period
- `ret` – Return value

`sys_port_trace_k_mbox_async_put_enter`(mbox, sem)
Trace Mailbox async put entry.

Parameters

- `mbox` – Mailbox object
- `sem` – Semaphore object

`sys_port_trace_k_mbox_async_put_exit`(mbox, sem)
Trace Mailbox async put exit.

Parameters

- `mbox` – Mailbox object
- `sem` – Semaphore object

`sys_port_trace_k_mbox_get_enter`(mbox, timeout)
Trace Mailbox get attempt entry.

Parameters

- `mbox` – Mailbox entry
- `timeout` – Timeout period

`sys_port_trace_k_mbox_get_blocking`(mbox, timeout)
Trace Mailbox get attempt blocking.

Parameters

- `mbox` – Mailbox entry
- `timeout` – Timeout period

`sys_port_trace_k_mbox_get_exit`(mbox, timeout, ret)
Trace Mailbox get attempt outcome.

Parameters

- `mbox` – Mailbox entry
- `timeout` – Timeout period
- `ret` – Return value

`sys_port_trace_k_mbox_data_get`(rx_msg)
Trace Mailbox data get.

rx_msg Receive Message object

Pipes

group `subsys_tracing_apis_pipe`
Pipe Tracing APIs.

Defines

sys_port_trace_k_pipe_init(pipe)
Trace initialization of Pipe.

Parameters

- pipe – Pipe object

sys_port_trace_k_pipe_cleanup_enter(pipe)
Trace Pipe cleanup entry.

Parameters

- pipe – Pipe object

sys_port_trace_k_pipe_cleanup_exit(pipe, ret)
Trace Pipe cleanup exit.

Parameters

- pipe – Pipe object
- ret – Return value

sys_port_trace_k_pipe_alloc_init_enter(pipe)
Trace Pipe alloc init entry.

Parameters

- pipe – Pipe object

sys_port_trace_k_pipe_alloc_init_exit(pipe, ret)
Trace Pipe alloc init exit.

Parameters

- pipe – Pipe object
- ret – Return value

sys_port_trace_k_pipe_flush_enter(pipe)
Trace Pipe flush entry.

Parameters

- pipe – Pipe object

sys_port_trace_k_pipe_flush_exit(pipe)
Trace Pipe flush exit.

Parameters

- pipe – Pipe object

sys_port_trace_k_pipe_buffer_flush_enter(pipe)
Trace Pipe buffer flush entry.

Parameters

- pipe – Pipe object

sys_port_trace_k_pipe_buffer_flush_exit(pipe)
Trace Pipe buffer flush exit.

Parameters

- pipe – Pipe object

`sys_port_trace_k_pipe_put_enter`(pipe, timeout)
Trace Pipe put attempt entry.

Parameters

- `pipe` – Pipe object
- `timeout` – Timeout period

`sys_port_trace_k_pipe_put_blocking`(pipe, timeout)
Trace Pipe put attempt blocking.

Parameters

- `pipe` – Pipe object
- `timeout` – Timeout period

`sys_port_trace_k_pipe_put_exit`(pipe, timeout, ret)
Trace Pipe put attempt outcome.

Parameters

- `pipe` – Pipe object
- `timeout` – Timeout period
- `ret` – Return value

`sys_port_trace_k_pipe_get_enter`(pipe, timeout)
Trace Pipe get attempt entry.

Parameters

- `pipe` – Pipe object
- `timeout` – Timeout period

`sys_port_trace_k_pipe_get_blocking`(pipe, timeout)
Trace Pipe get attempt blocking.

Parameters

- `pipe` – Pipe object
- `timeout` – Timeout period

`sys_port_trace_k_pipe_get_exit`(pipe, timeout, ret)
Trace Pipe get attempt outcome.

Parameters

- `pipe` – Pipe object
- `timeout` – Timeout period
- `ret` – Return value

Heaps

group `subsys_tracing_apis_heap`
Heap Tracing APIs.

Defines

sys_port_trace_k_heap_init(h)
Trace initialization of Heap.

Parameters

- h – Heap object

sys_port_trace_k_heap_aligned_alloc_enter(h, timeout)
Trace Heap aligned alloc attempt entry.

Parameters

- h – Heap object
- timeout – Timeout period

sys_port_trace_k_heap_aligned_alloc_blocking(h, timeout)
Trace Heap align alloc attempt blocking.

Parameters

- h – Heap object
- timeout – Timeout period

sys_port_trace_k_heap_aligned_alloc_exit(h, timeout, ret)
Trace Heap align alloc attempt outcome.

Parameters

- h – Heap object
- timeout – Timeout period
- ret – Return value

sys_port_trace_k_heap_alloc_enter(h, timeout)
Trace Heap alloc enter.

Parameters

- h – Heap object
- timeout – Timeout period

sys_port_trace_k_heap_alloc_exit(h, timeout, ret)
Trace Heap alloc exit.

Parameters

- h – Heap object
- timeout – Timeout period
- ret – Return value

sys_port_trace_k_heap_free(h)
Trace Heap free.

Parameters

- h – Heap object

sys_port_trace_k_heap_realloc_enter(h, ptr, bytes, timeout)
Trace Heap realloc enter.

Parameters

- h – Heap object

- **ptr** – Pointer to reallocate
- **bytes** – Bytes to reallocate
- **timeout** – Timeout period

sys_port_trace_k_heap_realloc_exit(h, ptr, bytes, timeout, ret)
Trace Heap realloc exit.

Parameters

- **h** – Heap object
- **ptr** – Pointer to reallocate
- **bytes** – Bytes to reallocate
- **timeout** – Timeout period
- **ret** – Return value

sys_port_trace_k_heap_sys_k_aligned_alloc_enter(heap)
Trace System Heap aligned alloc enter.

Parameters

- **heap** – Heap object

sys_port_trace_k_heap_sys_k_aligned_alloc_exit(heap, ret)
Trace System Heap aligned alloc exit.

Parameters

- **heap** – Heap object
- **ret** – Return value

sys_port_trace_k_heap_sys_k_malloc_enter(heap)
Trace System Heap aligned alloc enter.

Parameters

- **heap** – Heap object

sys_port_trace_k_heap_sys_k_malloc_exit(heap, ret)
Trace System Heap aligned alloc exit.

Parameters

- **heap** – Heap object
- **ret** – Return value

sys_port_trace_k_heap_sys_k_free_enter(heap, heap_ref)
Trace System Heap free entry.

Parameters

- **heap** – Heap object
- **heap_ref** – Heap reference

sys_port_trace_k_heap_sys_k_free_exit(heap, heap_ref)
Trace System Heap free exit.

Parameters

- **heap** – Heap object
- **heap_ref** – Heap reference

`sys_port_trace_k_heap_sys_k_calloc_enter`(heap)
Trace System heap calloc enter.

Parameters

- `heap`

`sys_port_trace_k_heap_sys_k_calloc_exit`(heap, ret)
Trace System heap calloc exit.

Parameters

- `heap` – Heap object
- `ret` – Return value

`sys_port_trace_k_heap_sys_k_realloc_enter`(heap, ptr)
Trace System heap realloc enter.

Parameters

- `heap`
- `ptr`

`sys_port_trace_k_heap_sys_k_realloc_exit`(heap, ptr, ret)
Trace System heap realloc exit.

Parameters

- `heap` – Heap object
- `ptr` – Memory pointer
- `ret` – Return value

Memory Slabs

group `subsys_tracing_apis_mslab`
Memory Slab Tracing APIs.

Defines

`sys_port_trace_k_mem_slab_init`(slab, rc)
Trace initialization of Memory Slab.

Parameters

- `slab` – Memory Slab object
- `rc` – Return value

`sys_port_trace_k_mem_slab_alloc_enter`(slab, timeout)
Trace Memory Slab alloc attempt entry.

Parameters

- `slab` – Memory Slab object
- `timeout` – Timeout period

`sys_port_trace_k_mem_slab_alloc_blocking`(slab, timeout)
Trace Memory Slab alloc attempt blocking.

Parameters

- `slab` – Memory Slab object
- `timeout` – Timeout period

`sys_port_trace_k_mem_slab_alloc_exit`(slab, timeout, ret)
Trace Memory Slab alloc attempt outcome.

Parameters

- `slab` – Memory Slab object
- `timeout` – Timeout period
- `ret` – Return value

`sys_port_trace_k_mem_slab_free_enter`(slab)
Trace Memory Slab free entry.

Parameters

- `slab` – Memory Slab object

`sys_port_trace_k_mem_slab_free_exit`(slab)
Trace Memory Slab free exit.

Parameters

- `slab` – Memory Slab object

Timers

group `subsys_tracing_apis_timer`
Timer Tracing APIs.

Defines

`sys_port_trace_k_timer_init`(timer)
Trace initialization of Timer.

Parameters

- `timer` – Timer object

`sys_port_trace_k_timer_start`(timer, duration, period)
Trace Timer start.

Parameters

- `timer` – Timer object
- `duration` – Timer duration
- `period` – Timer period

`sys_port_trace_k_timer_stop`(timer)
Trace Timer stop.

Parameters

- `timer` – Timer object

`sys_port_trace_k_timer_status_sync_enter`(timer)

Trace Timer status sync entry.

Parameters

- `timer` – Timer object

`sys_port_trace_k_timer_status_sync_blocking`(timer, timeout)

Trace Timer Status sync blocking.

Parameters

- `timer` – Timer object
- `timeout` – Timeout period

`sys_port_trace_k_timer_status_sync_exit`(timer, result)

Trace Time Status sync outcome.

Parameters

- `timer` – Timer object
- `result` – Return value

Object tracking

group `subsys_tracing_object_tracking`

Object tracking.

Object tracking provides lists to kernel objects, so their existence and current status can be tracked.

The following global variables are the heads of available lists:

- _track_list_k_timer
- _track_list_k_mem_slab
- _track_list_k_sem
- _track_list_k_mutex
- _track_list_k_stack
- _track_list_k_msgq
- _track_list_k_mbox
- _track_list_k_pipe
- _track_list_k_queue
- _track_list_k_event

Defines

`SYS_PORT_TRACK_NEXT`(list)

Gets node's next element in a object tracking list.

Parameters

- `list` – Node to get next element from.

Syscalls

group `subsys_tracing_apis_syscall`

Syscall Tracing APIs.

Defines

`sys_port_trace_syscall_enter`(id, name, ...)

Trace syscall entry.

Parameters

- `id` – Syscall ID (as defined in the generated syscall_list.h)
- `name` – Syscall name as a token (ex: k_thread_create)
- `...` – Other parameters passed to the syscall

`sys_port_trace_syscall_exit`(id, name, ...)

Trace syscall exit.

Parameters

- `id` – Syscall ID (as defined in the generated syscall_list.h)
- `name` – Syscall name as a token (ex: k_thread_create)
- `...` – Other parameters passed to the syscall, if the syscall has a return, the return value is the last parameter in the list

4.14 Resource Management

There are various situations where it's necessary to coordinate resource use at runtime among multiple clients. These include power rails, clocks, other peripherals, and binary device power management. The complexity of properly managing multiple consumers of a device in a multi-threaded system, especially when transitions may be asynchronous, suggests that a shared implementation is desirable.

Zephyr provides managers for several coordination policies. These managers are embedded into services that use them for specific functions.

- *On-Off Manager*

4.14.1 On-Off Manager

An on-off manager supports an arbitrary number of clients of a service which has a binary state. Example applications are power rails, clocks, and binary device power management.

The manager has the following properties:

- The stable states are off, on, and error. The service always begins in the off state. The service may also be in a transition to a given state.

- The core operations are request (add a dependency) and release (remove a dependency). Supporting operations are reset (to clear an error state) and cancel (to reclaim client data from an in-progress transition). The service manages the state based on calls to functions that initiate these operations.
- The service transitions from off to on when first client request is received.
- The service transitions from on to off when last client release is received.
- Each service configuration provides functions that implement the transition from off to on, from on to off, and optionally from an error state to off. Transitions must be invokable from both thread and interrupt context.
- The request and reset operations are asynchronous using *Asynchronous Notifications*. Both operations may be cancelled, but cancellation has no effect on the in-progress transition.
- Requests to turn on may be queued while a transition to off is in progress: when the service has turned off successfully it will be immediately turned on again (where context allows) and waiting clients notified when the start completes.

Requests are reference counted, but not tracked. That means clients are responsible for recording whether their requests were accepted, and for initiating a release only if they have previously successfully completed a request. Improper use of the API can cause an active client to be shut out, and the manager does not maintain a record of specific clients that have been granted a request.

Failures in executing a transition are recorded and inhibit further requests or releases until the manager is reset. Pending requests are notified (and cancelled) when errors are discovered.

Transition operation completion notifications are provided through *Asynchronous Notifications*.

Clients and other components interested in tracking all service state changes, including when a service begins turning off or enters an error state, can be informed of state transitions by registering a monitor with onoff_monitor_register(). Notification of changes are provided before issuing completion notifications associated with the new state.

Note

A generic API may be implemented by multiple drivers where the common case is asynchronous. The on-off client structure may be an appropriate solution for the generic API. Where drivers that can guarantee synchronous context-independent transitions a driver may use *onoff_sync_service* and its supporting API rather than *onoff_manager*, with only a small reduction in functionality (primarily no support for the monitor API).

group `resource_mgmt_onoff_apis`

Defines

`ONOFF_FLAG_ERROR`

Flag indicating an error state.

Error states are cleared using *onoff_reset()*.

`ONOFF_STATE_MASK`

Mask used to isolate bits defining the service state.

Mask a value with this then test for ONOFF_FLAG_ERROR to determine whether the machine has an unfixed error, or compare against ONOFF_STATE_ON,

ONOFF_STATE_OFF, ONOFF_STATE_TO_ON, ONOFF_STATE_TO_OFF, or ONOFF_STATE_RESETTING.

`ONOFF_STATE_OFF`

Value exposed by ONOFF_STATE_MASK when service is off.

`ONOFF_STATE_ON`

Value exposed by ONOFF_STATE_MASK when service is on.

`ONOFF_STATE_ERROR`

Value exposed by ONOFF_STATE_MASK when the service is in an error state (and not in the process of resetting its state).

`ONOFF_STATE_TO_ON`

Value exposed by ONOFF_STATE_MASK when service is transitioning to on.

`ONOFF_STATE_TO_OFF`

Value exposed by ONOFF_STATE_MASK when service is transitioning to off.

`ONOFF_STATE_RESETTING`

Value exposed by ONOFF_STATE_MASK when service is in the process of resetting.

`ONOFF_TRANSITIONS_INITIALIZER`(_start, _stop, _reset)

Initializer for a *onoff_transitions* object.

Parameters

- **`_start`** – a function used to transition from off to on state.
- **`_stop`** – a function used to transition from on to off state.
- **`_reset`** – a function used to clear errors and force the service to an off state. Can be null.

`ONOFF_CLIENT_EXTENSION_POS`

Identify region of *sys_notify* flags available for containing services.

Bits of the flags field of the *sys_notify* structure contained within the queued_operation structure at and above this position may be used by extensions to the *onoff_client* structure.

These bits are intended for use by containing service implementations to record client-specific information and are subject to other conditions of use specified on the *sys_notify* API.

Typedefs

typedef void (*`onoff_notify_fn`)(struct *onoff_manager* *mgr, int res)

Signature used to notify an on-off manager that a transition has completed.

Functions of this type are passed to service-specific transition functions to be used to report the completion of the operation. The functions may be invoked from any context.

Param mgr
the manager for which transition was requested.

Param res
the result of the transition. This shall be non-negative on success, or a negative error code. If an error is indicated the service shall enter an error state.

typedef void (*`onoff_transition_fn`)(struct *onoff_manager* *mgr, *onoff_notify_fn* notify)

Signature used by service implementations to effect a transition.

Service definitions use two required function pointers of this type to be notified that a transition is required, and a third optional one to reset the service when it is in an error state.

The start function will be called only from the off state.

The stop function will be called only from the on state.

The reset function (where supported) will be called only when *onoff_has_error()* returns true.

Note

All transitions functions must be isr-ok.

Param mgr
the manager for which transition was requested.

Param notify
the function to be invoked when the transition has completed. If the transition is synchronous, notify shall be invoked by the implementation before the transition function returns. Otherwise the implementation shall capture this parameter and invoke it when the transition completes.

typedef void (*`onoff_client_callback`)(struct *onoff_manager* *mgr, struct *onoff_client* *cli, uint32_t state, int res)

Signature used to notify an on-off service client of the completion of an operation.

These functions may be invoked from any context including pre-kernel, ISR, or cooperative or pre-emptible threads. Compatible functions must be isr-ok and not sleep.

Param mgr
the manager for which the operation was initiated. This may be null if the on-off service uses synchronous transitions.

Param cli
the client structure passed to the function that initiated the operation.

Param state
the state of the machine at the time of completion, restricted by ONOFF_STATE_MASK. ONOFF_FLAG_ERROR must be checked independently of whether res is negative as a machine error may indicate that all future operations except *onoff_reset()* will fail.

Param res
the result of the operation. Expected values are service-specific, but the value shall be non-negative if the operation succeeded, and negative if the operation failed. If res is negative ONOFF_FLAG_ERROR will be set in state, but if res is non-negative ONOFF_FLAG_ERROR may still be set in state.

typedef void (*`onoff_monitor_callback`)(struct *onoff_manager* *mgr, struct *onoff_monitor* *mon, uint32_t state, int res)

Signature used to notify a monitor of an onoff service of errors or completion of a state transition.

This is similar to onoff_client_callback but provides information about all transitions, not just ones associated with a specific client. Monitor callbacks are invoked before any completion notifications associated with the state change are made.

These functions may be invoked from any context including pre-kernel, ISR, or cooperative or pre-emptible threads. Compatible functions must be isr-ok and not sleep.

The callback is permitted to unregister itself from the manager, but must not register or unregister any other monitors.

Param mgr
the manager for which a transition has completed.

Param mon
the monitor instance through which this notification arrived.

Param state
the state of the machine at the time of completion, restricted by ONOFF_STATE_MASK. All valid states may be observed.

Param res
the result of the operation. Expected values are service- and state-specific, but the value shall be non-negative if the operation succeeded, and negative if the operation failed.

Functions

int `onoff_manager_init`(struct *onoff_manager* *mgr, const struct *onoff_transitions* *transitions)

Initialize an on-off service to off state.

This function must be invoked exactly once per service instance, by the infrastructure that provides the service, and before any other on-off service API is invoked on the service.

This function should never be invoked by clients of an on-off service.

Parameters

- `mgr` – the manager definition object to be initialized.
- `transitions` – pointer to a structure providing transition functions. The referenced object must persist as long as the manager can be referenced.

Return values

- `0` – on success
- `-EINVAL` – if start, stop, or flags are invalid

static inline bool `onoff_has_error`(const struct *onoff_manager* *mgr)

Test whether an on-off service has recorded an error.

This function can be used to determine whether the service has recorded an error. Errors may be cleared by invoking *onoff_reset()*.

This is an unlocked convenience function suitable for use only when it is known that no other process might invoke an operation that transitions the service between an error and non-error state.

Returns
true if and only if the service has an uncleared error.

int `onoff_request`(struct *onoff_manager* *mgr, struct *onoff_client* *cli)

Request a reservation to use an on-off service.

The return value indicates the success or failure of an attempt to initiate an operation to request the resource be made available. If initiation of the operation succeeds the result of the request operation is provided through the configured client notification method, possibly before this call returns.

Note that the call to this function may succeed in a case where the actual request fails. Always check the operation completion result.

Parameters

- `mgr` – the manager that will be used.
- `cli` – a non-null pointer to client state providing instructions on synchronous expectations and how to notify the client when the request completes. Behavior is undefined if client passes a pointer object associated with an incomplete service operation.

Return values

- `non-negative` – the observed state of the machine at the time the request was processed, if successful.
- `-EIO` – if service has recorded an error.
- `-EINVAL` – if the parameters are invalid.
- `-EAGAIN` – if the reference count would overflow.

int `onoff_release`(struct *onoff_manager* *mgr)

Release a reserved use of an on-off service.

This synchronously releases the caller's previous request. If the last request is released the manager will initiate a transition to off, which can be observed by registering an *onoff_monitor*.

> **Note**
>
> Behavior is undefined if this is not paired with a preceding *onoff_request()* call that completed successfully.

Parameters

- `mgr` – the manager for which a request was successful.

Return values

- `non-negative` – the observed state (ONOFF_STATE_ON) of the machine at the time of the release, if the release succeeds.
- `-EIO` – if service has recorded an error.
- `-ENOTSUP` – if the machine is not in a state that permits release.

int `onoff_cancel`(struct *onoff_manager* *mgr, struct *onoff_client* *cli)

Attempt to cancel an in-progress client operation.

It may be that a client has initiated an operation but needs to shut down before the operation has completed. For example, when a request was made and the need is no longer present.

Cancelling is supported only for *onoff_request()* and *onoff_reset()* operations, and is a synchronous operation. Be aware that any transition that was initiated on behalf of the client will continue to progress to completion: it is only notification of transition

completion that may be eliminated. If there are no active requests when a transition to on completes the manager will initiate a transition to off.

Client notification does not occur for cancelled operations.

Parameters

- `mgr` – the manager for which an operation is to be cancelled.
- `cli` – a pointer to the same client state that was provided when the operation to be cancelled was issued.

Return values

- `non-negative` – the observed state of the machine at the time of the cancellation, if the cancellation succeeds. On successful cancellation ownership of `*cli` reverts to the client.
- `-EINVAL` – if the parameters are invalid.
- `-EALREADY` – if cli was not a record of an uncompleted notification at the time the cancellation was processed. This likely indicates that the operation and client notification had already completed.

static inline int **onoff_cancel_or_release**(struct *onoff_manager* *mgr, struct *onoff_client* *cli)

Helper function to safely cancel a request.

Some applications may want to issue requests on an asynchronous event (such as connection to a USB bus) and to release on a paired event (such as loss of connection to a USB bus). Applications cannot precisely determine that an in-progress request is still pending without using *onoff_monitor* and carefully avoiding race conditions.

This function is a helper that attempts to cancel the operation and issues a release if cancellation fails because the request was completed. This synchronously ensures that ownership of the client data reverts to the client so is available for a future request.

Parameters

- `mgr` – the manager for which an operation is to be cancelled.
- `cli` – a pointer to the same client state that was provided when *onoff_request()* was invoked. Behavior is undefined if this is a pointer to client data associated with an *onoff_reset()* request.

Return values

- `ONOFF_STATE_TO_ON` – if the cancellation occurred before the transition completed.
- `ONOFF_STATE_ON` – if the cancellation occurred after the transition completed.
- `-EINVAL` – if the parameters are invalid.
- `negative` – other errors produced by *onoff_release()*.

int **onoff_reset**(struct *onoff_manager* *mgr, struct *onoff_client* *cli)

Clear errors on an on-off service and reset it to its off state.

A service can only be reset when it is in an error state as indicated by *onoff_has_error()*.

The return value indicates the success or failure of an attempt to initiate an operation to reset the resource. If initiation of the operation succeeds the result of the reset operation itself is provided through the configured client notification method, possibly before this call returns. Multiple clients may request a reset; all are notified when it is complete.

Note that the call to this function may succeed in a case where the actual reset fails. Always check the operation completion result.

> **Note**
>
> Due to the conditions on state transition all incomplete asynchronous operations will have been informed of the error when it occurred. There need be no concern about dangling requests left after a reset completes.

Parameters

- `mgr` – the manager to be reset.
- `cli` – pointer to client state, including instructions on how to notify the client when reset completes. Behavior is undefined if cli references an object associated with an incomplete service operation.

Return values

- `non-negative` – the observed state of the machine at the time of the reset, if the reset succeeds.
- `-ENOTSUP` – if reset is not supported by the service.
- `-EINVAL` – if the parameters are invalid.
- `-EALREADY` – if the service does not have a recorded error.

int `onoff_monitor_register`(struct *onoff_manager* *mgr, struct *onoff_monitor* *mon)

Add a monitor of state changes for a manager.

Parameters

- `mgr` – the manager for which a state changes are to be monitored.
- `mon` – a linkable node providing a non-null callback to be invoked on state changes.

Returns

non-negative on successful addition, or a negative error code.

int `onoff_monitor_unregister`(struct *onoff_manager* *mgr, struct *onoff_monitor* *mon)

Remove a monitor of state changes from a manager.

Parameters

- `mgr` – the manager for which a state changes are to be monitored.
- `mon` – a linkable node providing the callback to be invoked on state changes.

Returns

non-negative on successful removal, or a negative error code.

int `onoff_sync_lock`(struct *onoff_sync_service* *srv, *k_spinlock_key_t* *keyp)

Lock a synchronous onoff service and provide its state.

> **Note**
>
> If an error state is returned it is the caller's responsibility to decide whether to preserve it (finalize with the same error state) or clear the error (finalize with a non-error result).

Parameters

- `srv` – pointer to the synchronous service state.
- `keyp` – pointer to where the lock key should be stored

Returns
negative if the service is in an error state, otherwise the number of active requests at the time the lock was taken. The lock is held on return regardless of whether a negative state is returned.

int `onoff_sync_finalize`(struct *onoff_sync_service* *srv, *k_spinlock_key_t* key, struct *onoff_client* *cli, int res, bool on)

Process the completion of a transition in a synchronous service and release lock.

This function updates the service state on the `res` and `on` parameters then releases the lock. If `cli` is not null it finalizes the client notification using `res`.

If the service was in an error state when locked, and `res` is non-negative when finalized, the count is reset to zero before completing finalization.

Parameters

- `srv` – pointer to the synchronous service state
- `key` – the key returned by the preceding invocation of *onoff_sync_lock()*.
- `cli` – pointer to the onoff client through which completion information is returned. If a null pointer is passed only the state of the service is updated. For compatibility with the behavior of callbacks used with the manager API `cli` must be null when `on` is false (the manager does not support callbacks when turning off devices).
- `res` – the result of the transition. A negative value places the service into an error state. A non-negative value increments or decrements the reference count as specified by `on`.
- `on` – Only when `res` is non-negative, the service reference count will be incremented if`on` is `true`, and decremented if `on` is `false`.

Returns
negative if the service is left or put into an error state, otherwise the number of active requests at the time the lock was released.

struct `onoff_transitions`

#include <*onoff.h*> On-off service transition functions.

Public Members

onoff_transition_fn `start`

Function to invoke to transition the service to on.

onoff_transition_fn `stop`

Function to invoke to transition the service to off.

onoff_transition_fn `reset`

Function to force the service state to reset, where supported.

struct `onoff_manager`

#include <*onoff.h*> State associated with an on-off manager.

No fields in this structure are intended for use by service providers or clients. The state is to be initialized once, using *onoff_manager_init()*, when the service provider is initialized. In case of error it may be reset through the *onoff_reset()* API.

Public Members

sys_slist_t **clients**

List of clients waiting for request or reset completion notifications.

sys_slist_t **monitors**

List of monitors to be notified of state changes including errors and transition completion.

const struct *onoff_transitions* ***transitions**

Transition functions.

struct *k_spinlock* **lock**

Mutex protection for other fields.

int **last_res**

The result of the last transition.

uint16_t **flags**

Flags identifying the service state.

uint16_t **refs**

Number of active clients for the service.

struct **onoff_client**

#include <*onoff.h*> State associated with a client of an on-off service.

Objects of this type are allocated by a client, which is responsible for zero-initializing the node field and invoking the appropriate *sys_notify* init function to configure notification.

Control of the object content transfers to the service provider when a pointer to the object is passed to any on-off manager function. While the service provider controls the object the client must not change any object fields. Control reverts to the client concurrent with release of the owned *sys_notify* structure, or when indicated by an *onoff_cancel()* return value.

After control has reverted to the client the notify field must be reinitialized for the next operation.

Public Members

struct *sys_notify* **notify**

Notification configuration.

struct **onoff_monitor**

#include <onoff.h> Registration state for notifications of onoff service transitions.

Any given *onoff_monitor* structure can be associated with at most one *onoff_manager* instance.

Public Members

sys_snode_t **node**

Links the client into the set of waiting service users.

This must be zero-initialized.

onoff_monitor_callback **callback**

Callback to be invoked on state change.

This must not be null.

struct **onoff_sync_service**

#include <onoff.h> State used when a driver uses the on-off service API for synchronous operations.

This is useful when a subsystem API uses the on-off API to support asynchronous operations but the transitions required by a particular driver are isr-ok and not sleep. It serves as a substitute for *onoff_manager*, with locking and persisted state updates supported by *onoff_sync_lock()* and *onoff_sync_finalize()*.

Public Members

struct *k_spinlock* **lock**

Mutex protection for other fields.

int32_t **count**

Negative is error, non-negative is reference count.

4.15 Memory Attributes

It is possible in the devicetree to mark the memory regions with attributes by using the `zephyr,memory-attr` property. This property and the related memory region can then be retrieved at run-time by leveraging a provided helper library.

The set of general attributes that can be specified in the property are defined and explained in include/zephyr/dt-bindings/memory-attr/memory-attr.h.

For example, to mark a memory region in the devicetree as non-volatile, cacheable, out-of-order:

```
mem: memory@10000000 {
    compatible = "mmio-sram";
    reg = <0x10000000 0x1000>;
    zephyr,memory-attr = <( DT_MEM_NON_VOLATILE | DT_MEM_CACHEABLE | DT_MEM_OOO )>;
};
```

Note

The `zephyr,memory-attr` usage does not result in any memory region actually created. When it is needed to create an actual section out of the devicetree defined memory region, it is possible to use the compatible `zephyr,memory-region` that will result (only when supported by the architecture) in a new linker section and region.

The `zephyr,memory-attr` property can also be used to set architecture-specific and software-specific custom attributes that can be interpreted at run time. This is leveraged, among other things, to create MPU regions out of devicetree defined memory regions, for example:

```
mem: memory@10000000 {
    compatible = "mmio-sram";
    reg = <0x10000000 0x1000>;
    zephyr,memory-region = "NOCACHE_REGION";
    zephyr,memory-attr = <( DT_MEM_ARM(ATTR_MPU_RAM_NOCACHE) )>;
};
```

See include/zephyr/dt-bindings/memory-attr/memory-attr-arm.h and *Arm Cortex-M Developer Guide* for more details about MPU usage.

The conventional and recommended way to deal and manage with memory regions marked with attributes is by using the provided `mem-attr` helper library by enabling `CONFIG_MEM_ATTR`. When this option is enabled the list of memory regions and their attributes are compiled in a user-accessible array and a set of functions is made available that can be used to query, probe and act on regions and attributes (see next section for more details).

Note

The `zephyr,memory-attr` property is only a descriptive property of the capabilities of the associated memory region, but it does not result in any actual setting for the memory to be set. The user, code or subsystem willing to use this information to do some work (for example creating an MPU region out of the property) must use either the provided `mem-attr` library or the usual devicetree helpers to perform the required work / setting.

A test for the `mem-attr` library and its usage is provided in `tests/subsys/mem_mgmt/mem_attr/`.

4.15.1 Migration guide from *zephyr,memory-region-mpu*

When the `zephyr,memory-attr` property was introduced, the `zephyr,memory-region-mpu` property was removed and deprecated.

The developers that are still using the deprecated property can move to the new one by renaming the property and changing its value according to the following list:

```
"RAM"         -> <( DT_ARM_MPU(ATTR_MPU_RAM) )>
"RAM_NOCACHE" -> <( DT_ARM_MPU(ATTR_MPU_RAM_NOCACHE) )>
"FLASH"       -> <( DT_ARM_MPU(ATTR_MPU_FLASH) )>
"PPB"         -> <( DT_ARM_MPU(ATTR_MPU_PPB) )>
"IO"          -> <( DT_ARM_MPU(ATTR_MPU_IO) )>
"EXTMEM"      -> <( DT_ARM_MPU(ATTR_MPU_EXTMEM) )>
```

4.15.2 Memory Attributes Heap Allocator

It is possible to leverage the memory attribute property `zephyr,memory-attr` to define and create a set of memory heaps from which the user can allocate memory from with certain attributes /

capabilities.

When the CONFIG_MEM_ATTR_HEAP is set, every region marked with one of the memory attributes listed in include/zephyr/dt-bindings/memory-attr/memory-attr-sw.h is added to a pool of memory heaps used for dynamic allocation of memory buffers with certain attributes.

Here a non exhaustive list of possible attributes:

```
DT_MEM_SW_ALLOC_CACHE
DT_MEM_SW_ALLOC_NON_CACHE
DT_MEM_SW_ALLOC_DMA
```

For example we can define several memory regions with different attributes and use the appropriate attribute to indicate that it is possible to dynamically allocate memory from those regions:

```
mem_cacheable: memory@10000000 {
    compatible = "mmio-sram";
    reg = <0x10000000 0x1000>;
    zephyr,memory-attr = <( DT_MEM_CACHEABLE | DT_MEM_SW_ALLOC_CACHE )>;
};

mem_non_cacheable: memory@20000000 {
    compatible = "mmio-sram";
    reg = <0x20000000 0x1000>;
    zephyr,memory-attr = <( DT_MEM_NON_CACHEABLE | ATTR_SW_ALLOC_NON_CACHE )>;
};

mem_cacheable_big: memory@30000000 {
    compatible = "mmio-sram";
    reg = <0x30000000 0x10000>;
    zephyr,memory-attr = <( DT_MEM_CACHEABLE | DT_MEM_OOO | DT_MEM_SW_ALLOC_CACHE )>;
};

mem_cacheable_dma: memory@40000000 {
    compatible = "mmio-sram";
    reg = <0x40000000 0x10000>;
    zephyr,memory-attr = <( DT_MEM_CACHEABLE      | DT_MEM_DMA |
                            DT_MEM_SW_ALLOC_CACHE | DT_MEM_SW_ALLOC_DMA )>;
};
```

The user can then dynamically carve memory out of those regions using the provided functions, the library will take care of allocating memory from the correct heap depending on the provided attribute and size:

```
// Init the pool
mem_attr_heap_pool_init();

// Allocate 0x100 bytes of cacheable memory from `mem_cacheable`
block = mem_attr_heap_alloc(DT_MEM_SW_ALLOC_CACHE, 0x100);

// Allocate 0x200 bytes of non-cacheable memory aligned to 32 bytes
// from `mem_non_cacheable`
block = mem_attr_heap_aligned_alloc(ATTR_SW_ALLOC_NON_CACHE, 0x100, 32);

// Allocate 0x100 bytes of cacheable and dma-able memory from `mem_cacheable_dma`
block = mem_attr_heap_alloc(DT_MEM_SW_ALLOC_CACHE | DT_MEM_SW_ALLOC_DMA, 0x100);
```

When several regions are marked with the same attributes, the memory is allocated:

1. From the regions where the zephyr,memory-attr property has the requested property (or properties).
2. Among the regions as at point 1, from the smallest region if there is any unallocated space left for the requested size

3. If there is not enough space, from the next bigger region able to accommodate the requested size

The following example shows the point 3:

```
// This memory is allocated from `mem_non_cacheable`
block = mem_attr_heap_alloc(DT_MEM_SW_ALLOC_NON_CACHE, 0x100);

// This memory is allocated from `mem_cacheable_big`
block = mem_attr_heap_alloc(DT_MEM_SW_ALLOC_CACHE, 0x5000);
```

Note

The framework is assuming that the memory regions used to create the heaps are usable by the code and available at init time. The user must take of initializing and setting the memory area before calling *mem_attr_heap_pool_init()*.

That means that the region must be correctly configured in terms of MPU / MMU (if needed) and that an actual heap can be created out of it, for example by leveraging the `zephyr,memory-region` property to create a proper linker section to accommodate the heap.

4.15.3 API Reference

group **memory_attr_interface**

Memory-Attr Interface.

Defines

DT_MEMORY_ATTR_FOREACH_STATUS_OKAY_NODE(fn)

Invokes `fn` for every status `okay` node in the tree with property `zephyr,memory-attr`

The macro `fn` must take one parameter, which will be a node identifier with the `zephyr,memory-attr` property. The macro is expanded once for each node in the tree with status `okay`. The order that nodes are visited in is not specified.

Parameters

- **fn** – macro to invoke

Functions

size_t **mem_attr_get_regions**(const struct *mem_attr_region_t* **region)

Get the list of memory regions.

Get the list of enabled memory regions with their memory-attribute as gathered by DT.

Parameters

- **region** – Pointer to pointer to the list of memory regions.

Return values

Number – of memory regions returned in the parameter.

int **mem_attr_check_buf**(void *addr, size_t size, uint32_t attr)

Check if a buffer has correct size and attributes.

This function is used to check if a given buffer with a given set of attributes fully match a memory region in terms of size and attributes.

This is usually used to verify that a buffer has the expected attributes (for example the buffer is cacheable / non-cacheable or belongs to RAM / FLASH, etc...) and it has been correctly allocated.

The expected set of attributes for the buffer is and-matched against the full set of attributes for the memory region it belongs to (bitmask). So the buffer is considered matching when at least that set of attributes are valid for the memory region (but the region can be marked also with other attributes besides the one passed as parameter).

Parameters

- **addr** – Virtual address of the user buffer.
- **size** – Size of the user buffer.
- **attr** – Expected / desired attribute for the buffer.

Return values

- **0** – if the buffer has the correct size and attribute.
- **-ENOSYS** – if the operation is not supported (for example if the MMU is enabled).
- **-ENOTSUP** – if the wrong parameters were passed.
- **-EINVAL** – if the buffer has the wrong set of attributes.
- **-ENOSPC** – if the buffer is too big for the region it belongs to.
- **-ENOBUFS** – if the buffer is entirely allocated outside a memory region.

struct **mem_attr_region_t**

#include <*mem_attr.h*> memory-attr region structure.

This structure represents the data gathered from DT about a memory-region marked with memory attributes.

Public Members

const char ***dt_name**

Memory node full name.

uintptr_t **dt_addr**

Memory region physical address.

size_t **dt_size**

Memory region size.

uint32_t **dt_attr**

Memory region attributes.

group **memory_attr_heap**

Memory heaps based on memory attributes.

Functions

int **mem_attr_heap_pool_init**(void)

Init the memory pool.

This must be the first function to be called to initialize the memory pools from all the memory regions with the a software attribute.

Return values

- `0` – on success.
- `-EALREADY` – if the pool was already initialized.
- `-ENOMEM` – too many regions already allocated.

void ***mem_attr_heap_alloc**(uint32_t attr, size_t bytes)

Allocate memory with a specified attribute and size.

Allocates a block of memory of the specified size in bytes and with a specified capability / attribute. The attribute is used to select the correct memory heap to allocate memory from.

Parameters

- `attr` – capability / attribute requested for the memory block.
- `bytes` – requested size of the allocation in bytes.

Return values

- `ptr` – a valid pointer to the allocated memory.
- `NULL` – if no memory is available with that attribute and size.

void ***mem_attr_heap_aligned_alloc**(uint32_t attr, size_t align, size_t bytes)

Allocate aligned memory with a specified attribute, size and alignment.

Allocates a block of memory of the specified size in bytes and with a specified capability / attribute. Takes an additional parameter specifying a power of two alignment in bytes.

Parameters

- `attr` – capability / attribute requested for the memory block.
- `align` – power of two alignment for the returned pointer in bytes.
- `bytes` – requested size of the allocation in bytes.

Return values

- `ptr` – a valid pointer to the allocated memory.
- `NULL` – if no memory is available with that attribute and size.

void **mem_attr_heap_free**(void *block)

Free the allocated memory.

Used to free the passed block of memory that must be the return value of a previously call to *mem_attr_heap_alloc* or *mem_attr_heap_aligned_alloc*.

Parameters

- `block` – block to free, must be a pointer to a block allocated by *mem_attr_heap_alloc* or *mem_attr_heap_aligned_alloc*.

const struct *mem_attr_region_t* ***mem_attr_heap_get_region**(void *addr)

Get a specific memory region descriptor for a provided address.

Finds the memory region descriptor struct controlling the provided pointer.

Parameters

- `addr` – address to be found, must be a pointer to a block allocated by *mem_attr_heap_alloc* or *mem_attr_heap_aligned_alloc*.

Return values

`str` – pointer to a memory region structure the address belongs to.

4.16 Modbus

Modbus is an industrial messaging protocol. The protocol is specified for different types of networks or buses. Zephyr OS implementation supports communication over serial line and may be used with different physical interfaces, like RS485 or RS232. TCP support is not implemented directly, but there are helper functions to realize TCP support according to the application's needs.

Modbus communication is based on client/server model. Only one client may be present on the bus. Client can communicate with several server devices. Server devices themselves are passive and must not send requests or unsolicited responses. Services requested by the client are specified by function codes (FCxx), and can be found in the specification or documentation of the API below.

Zephyr RTOS implementation supports both client and server roles.

More information about Modbus and Modbus RTU can be found on the website MODBUS Protocol Specifications.

4.16.1 Samples

- modbus-rtu-server and modbus-rtu-client samples give the possibility to try out RTU server and RTU client implementation with an evaluation board.
- modbus-tcp-server sample is a simple Modbus TCP server.
- modbus-gateway sample shows how to build a TCP to serial line gateway with Zephyr OS.

4.16.2 API Reference

Related code samples

Modbus RTU client
Communicate with a Modbus RTU server.

Modbus RTU server
Implement a Modbus RTU server exposing Modbus commands to control LEDs.

Modbus TCP server
Implement a Modbus TCP server exposing Modbus commands to control LEDs.

Modbus TCP-to-serial gateway
Implement a gateway between an Ethernet TCP-IP network and a Modbus serial line.

group modbus

MODBUS transport protocol API.

Modbus exception codes

MODBUS_EXC_NONE
: No exception.

MODBUS_EXC_ILLEGAL_FC
: Illegal function code.

MODBUS_EXC_ILLEGAL_DATA_ADDR
: Illegal data address.

MODBUS_EXC_ILLEGAL_DATA_VAL
: Illegal data value.

MODBUS_EXC_SERVER_DEVICE_FAILURE
: Server device failure.

MODBUS_EXC_ACK
: Acknowledge.

MODBUS_EXC_SERVER_DEVICE_BUSY
: Server device busy.

MODBUS_EXC_MEM_PARITY_ERROR
: Memory parity error.

MODBUS_EXC_GW_PATH_UNAVAILABLE
: Gateway path unavailable.

MODBUS_EXC_GW_TARGET_FAILED_TO_RESP
: Gateway target device failed to respond.

Defines

MODBUS_MBAP_LENGTH
: Length of MBAP Header.

MODBUS_MBAP_AND_FC_LENGTH
: Length of MBAP Header plus function code.

MODBUS_CUSTOM_FC_DEFINE(name, user_cb, user_fc, userdata)
: INTERNAL_HIDDEN.

  Helper macro for initializing custom function code structs

Typedefs

typedef int (*`modbus_raw_cb_t`)(const int iface, const struct *modbus_adu* *adu, void *user_data)

ADU raw callback function signature.

Param iface
Modbus RTU interface index

Param adu
Pointer to the RAW ADU struct to send

Param user_data
Pointer to the user data

Retval 0
If transfer was successful

typedef bool (*`modbus_custom_cb_t`)(const int iface, const struct *modbus_adu* *const rx_adu, struct *modbus_adu* *const tx_adu, uint8_t *const excep_code, void *const user_data)

Custom function code handler function signature.

Modbus allows user defined function codes which can be used to extend the base protocol. These callbacks can also be used to implement function codes currently not supported by Zephyr's Modbus subsystem.

If an error occurs during the handling of the request, the handler should signal this by setting excep_code to a modbus exception code.

User data pointer can be used to pass state between subsequent calls to the handler.

Param iface
Modbus interface index

Param rx_adu
Pointer to the received ADU struct

Param tx_adu
Pointer to the outgoing ADU struct

Param excep_code
Pointer to possible exception code

Param user_data
Pointer to user data

Retval true
If response should be sent, false otherwise

Enums

enum `modbus_mode`

Modbus interface mode.

Values:

enumerator `MODBUS_MODE_RTU`

Modbus over serial line RTU mode.

enumerator MODBUS_MODE_ASCII

Modbus over serial line ASCII mode.

enumerator MODBUS_MODE_RAW

Modbus raw ADU mode.

Functions

int modbus_read_coils(const int iface, const uint8_t unit_id, const uint16_t start_addr, uint8_t *const coil_tbl, const uint16_t num_coils)

Coil read (FC01)

Sends a Modbus message to read the status of coils from a server.

Parameters

- iface – Modbus interface index
- unit_id – Modbus unit ID of the server
- start_addr – Coil starting address
- coil_tbl – Pointer to an array of bytes containing the value of the coils read. The format is:

```
                MSB                                 LSB
                B7   B6   B5   B4   B3   B2   B1   B0
                -------------------------------------
coil_tbl[0]     #8   #7                            #1
coil_tbl[1]     #16  #15                           #9
     :
     :
```

  Note that the array that will be receiving the coil values must be greater than or equal to: (num_coils - 1) / 8 + 1

- num_coils – Quantity of coils to read

Return values

0 – If the function was successful

int modbus_read_dinputs(const int iface, const uint8_t unit_id, const uint16_t start_addr, uint8_t *const di_tbl, const uint16_t num_di)

Read discrete inputs (FC02)

Sends a Modbus message to read the status of discrete inputs from a server.

Parameters

- iface – Modbus interface index
- unit_id – Modbus unit ID of the server
- start_addr – Discrete input starting address
- di_tbl – Pointer to an array that will receive the state of the discrete inputs. The format of the array is as follows:

```
              MSB                                 LSB
              B7   B6   B5   B4   B3   B2   B1   B0
              -------------------------------------
di_tbl[0]     #8   #7                            #1
di_tbl[1]     #16  #15                           #9
```

(continues on next page)

(continued from previous page)

```
    :
    :
```

Note that the array that will be receiving the discrete input values must be greater than or equal to: (num_di - 1) / 8 + 1

- `num_di` – Quantity of discrete inputs to read

Return values
`0` – If the function was successful

int `modbus_read_holding_regs`(const int iface, const uint8_t unit_id, const uint16_t start_addr, uint16_t *const reg_buf, const uint16_t num_regs)

Read holding registers (FC03)

Sends a Modbus message to read the value of holding registers from a server.

Parameters

- `iface` – Modbus interface index
- `unit_id` – Modbus unit ID of the server
- `start_addr` – Register starting address
- `reg_buf` – Is a pointer to an array that will receive the current values of the holding registers from the server. The array pointed to by 'reg_buf' needs to be able to hold at least 'num_regs' entries.
- `num_regs` – Quantity of registers to read

Return values
`0` – If the function was successful

int `modbus_read_input_regs`(const int iface, const uint8_t unit_id, const uint16_t start_addr, uint16_t *const reg_buf, const uint16_t num_regs)

Read input registers (FC04)

Sends a Modbus message to read the value of input registers from a server.

Parameters

- `iface` – Modbus interface index
- `unit_id` – Modbus unit ID of the server
- `start_addr` – Register starting address
- `reg_buf` – Is a pointer to an array that will receive the current value of the holding registers from the server. The array pointed to by 'reg_buf' needs to be able to hold at least 'num_regs' entries.
- `num_regs` – Quantity of registers to read

Return values
`0` – If the function was successful

int `modbus_write_coil`(const int iface, const uint8_t unit_id, const uint16_t coil_addr, const bool coil_state)

Write single coil (FC05)

Sends a Modbus message to write the value of single coil to a server.

Parameters

- `iface` – Modbus interface index

- **unit_id** – Modbus unit ID of the server
- **coil_addr** – Coils starting address
- **coil_state** – Is the desired state of the coil

Return values
0 – If the function was successful

int **modbus_write_holding_reg**(const int iface, const uint8_t unit_id, const uint16_t start_addr, const uint16_t reg_val)

Write single holding register (FC06)

Sends a Modbus message to write the value of single holding register to a server unit.

Parameters

- **iface** – Modbus interface index
- **unit_id** – Modbus unit ID of the server
- **start_addr** – Coils starting address
- **reg_val** – Desired value of the holding register

Return values
0 – If the function was successful

int **modbus_request_diagnostic**(const int iface, const uint8_t unit_id, const uint16_t sfunc, const uint16_t data, uint16_t *const data_out)

Read diagnostic (FC08)

Sends a Modbus message to perform a diagnostic function of a server unit.

Parameters

- **iface** – Modbus interface index
- **unit_id** – Modbus unit ID of the server
- **sfunc** – Diagnostic sub-function code
- **data** – Sub-function data
- **data_out** – Pointer to the data value

Return values
0 – If the function was successful

int **modbus_write_coils**(const int iface, const uint8_t unit_id, const uint16_t start_addr, uint8_t *const coil_tbl, const uint16_t num_coils)

Write coils (FC15)

Sends a Modbus message to write to coils on a server unit.

Parameters

- **iface** – Modbus interface index
- **unit_id** – Modbus unit ID of the server
- **start_addr** – Coils starting address
- **coil_tbl** – Pointer to an array of bytes containing the value of the coils to write. The format is:

```
                MSB                                 LSB
                B7   B6   B5   B4   B3   B2   B1   B0
                -------------------------------------
coil_tbl[0]     #8   #7                             #1
```

(continues on next page)

(continued from previous page)

```
coil_tbl[1]     #16  #15                          #9
     :
     :
```

Note that the array that will be receiving the coil values must be greater than or equal to: (num_coils - 1) / 8 + 1

- `num_coils` – Quantity of coils to write

Return values

`0` – If the function was successful

int `modbus_write_holding_regs`(const int iface, const uint8_t unit_id, const uint16_t start_addr, uint16_t *const reg_buf, const uint16_t num_regs)

Write holding registers (FC16)

Sends a Modbus message to write to integer holding registers to a server unit.

Parameters

- `iface` – Modbus interface index
- `unit_id` – Modbus unit ID of the server
- `start_addr` – Register starting address
- `reg_buf` – Is a pointer to an array containing the value of the holding registers to write. Note that the array containing the register values must be greater than or equal to 'num_regs'
- `num_regs` – Quantity of registers to write

Return values

`0` – If the function was successful

int `modbus_read_holding_regs_fp`(const int iface, const uint8_t unit_id, const uint16_t start_addr, float *const reg_buf, const uint16_t num_regs)

Read floating-point holding registers (FC03)

Sends a Modbus message to read the value of floating-point holding registers from a server unit.

Parameters

- `iface` – Modbus interface index
- `unit_id` – Modbus unit ID of the server
- `start_addr` – Register starting address
- `reg_buf` – Is a pointer to an array that will receive the current values of the holding registers from the server. The array pointed to by 'reg_buf' needs to be able to hold at least 'num_regs' entries.
- `num_regs` – Quantity of registers to read

Return values

`0` – If the function was successful

int `modbus_write_holding_regs_fp`(const int iface, const uint8_t unit_id, const uint16_t start_addr, float *const reg_buf, const uint16_t num_regs)

Write floating-point holding registers (FC16)

Sends a Modbus message to write to floating-point holding registers to a server unit.

Parameters

- `iface` – Modbus interface index
- `unit_id` – Modbus unit ID of the server
- `start_addr` – Register starting address
- `reg_buf` – Is a pointer to an array containing the value of the holding registers to write. Note that the array containing the register values must be greater than or equal to 'num_regs'
- `num_regs` – Quantity of registers to write

Return values

`0` – If the function was successful

int `modbus_iface_get_by_name`(const char *iface_name)

Get Modbus interface index according to interface name.

If there is more than one interface, it can be used to clearly identify interfaces in the application.

Parameters

- `iface_name` – Modbus interface name

Return values

`Modbus` – interface index or negative error value.

int `modbus_init_server`(const int iface, struct *modbus_iface_param* param)

Configure Modbus Interface as raw ADU server.

Parameters

- `iface` – Modbus RTU interface index
- `param` – Configuration parameter of the server interface

Return values

`0` – If the function was successful

int `modbus_init_client`(const int iface, struct *modbus_iface_param* param)

Configure Modbus Interface as raw ADU client.

Parameters

- `iface` – Modbus RTU interface index
- `param` – Configuration parameter of the client interface

Return values

`0` – If the function was successful

int `modbus_disable`(const uint8_t iface)

Disable Modbus Interface.

This function is called to disable Modbus interface.

Parameters

- `iface` – Modbus interface index

Return values

`0` – If the function was successful

int `modbus_raw_submit_rx`(const int iface, const struct *modbus_adu* *adu)

Submit raw ADU.

Parameters

- **iface** – Modbus RTU interface index
- **adu** – Pointer to the RAW ADU struct that is received

Return values
0 – If transfer was successful

void **modbus_raw_put_header**(const struct *modbus_adu* *adu, uint8_t *header)

Put MBAP header into a buffer.

Parameters

- **adu** – Pointer to the RAW ADU struct
- **header** – Pointer to the buffer in which MBAP header will be placed.

void **modbus_raw_get_header**(struct *modbus_adu* *adu, const uint8_t *header)

Get MBAP header from a buffer.

Parameters

- **adu** – Pointer to the RAW ADU struct
- **header** – Pointer to the buffer containing MBAP header

void **modbus_raw_set_server_failure**(struct *modbus_adu* *adu)

Set Server Device Failure exception.

This function modifies ADU passed by the pointer.

Parameters

- **adu** – Pointer to the RAW ADU struct

int **modbus_raw_backend_txn**(const int iface, struct *modbus_adu* *adu)

Use interface as backend to send and receive ADU.

This function overwrites ADU passed by the pointer and generates exception responses if backend interface is misconfigured or target device is unreachable.

Parameters

- **iface** – Modbus client interface index
- **adu** – Pointer to the RAW ADU struct

Return values
0 – If transfer was successful

int **modbus_register_user_fc**(const int iface, struct modbus_custom_fc *custom_fc)

Register a user-defined function code handler.

The Modbus specification allows users to define standard function codes missing from Zephyr's Modbus implementation as well as add non-standard function codes in the ranges 65 to 72 and 100 to 110 (decimal), as per specification.

This function registers a new handler at runtime for the given function code.

Parameters

- **iface** – Modbus client interface index
- **custom_fc** – User defined function code and callback pair

Return values
0 – on success

struct **modbus_adu**

#include <*modbus.h*> Frame struct used internally and for raw ADU support.

Public Members

uint16_t **trans_id**
Transaction Identifier.

uint16_t **proto_id**
Protocol Identifier.

uint16_t **length**
Length of the data only (not the length of unit ID + PDU)

uint8_t **unit_id**
Unit Identifier.

uint8_t **fc**
Function Code.

uint8_t **data**[CONFIG_MODBUS_BUFFER_SIZE - 4]
Transaction Data.

uint16_t **crc**
RTU CRC.

struct **modbus_user_callbacks**
#include <*modbus.h*> Modbus Server User Callback structure.

Public Members

int (***coil_rd**)(uint16_t addr, bool *state)
Coil read callback.

int (***coil_wr**)(uint16_t addr, bool state)
Coil write callback.

int (***discrete_input_rd**)(uint16_t addr, bool *state)
Discrete Input read callback.

int (***input_reg_rd**)(uint16_t addr, uint16_t *reg)
Input Register read callback.

int (***input_reg_rd_fp**)(uint16_t addr, float *reg)
Floating Point Input Register read callback.

int (***holding_reg_rd**)(uint16_t addr, uint16_t *reg)
Holding Register read callback.

int (***holding_reg_wr**)(uint16_t addr, uint16_t reg)
Holding Register write callback.

int (*holding_reg_rd_fp)(uint16_t addr, float *reg)

Floating Point Holding Register read callback.

int (*holding_reg_wr_fp)(uint16_t addr, float reg)

Floating Point Holding Register write callback.

struct modbus_serial_param

#include <*modbus.h*> Modbus serial line parameter.

Public Members

uint32_t baud

Baudrate of the serial line.

enum *uart_config_parity* parity

parity UART's parity setting: UART_CFG_PARITY_NONE, UART_CFG_PARITY_EVEN, UART_CFG_PARITY_ODD

enum *uart_config_stop_bits* stop_bits_client

stop_bits_client UART's stop bits setting if in client mode: UART_CFG_STOP_BITS_0_5, UART_CFG_STOP_BITS_1, UART_CFG_STOP_BITS_1_5, UART_CFG_STOP_BITS_2,

struct modbus_server_param

#include <*modbus.h*> Modbus server parameter.

Public Members

struct *modbus_user_callbacks* *user_cb

Pointer to the User Callback structure.

uint8_t unit_id

Modbus unit ID of the server.

struct modbus_raw_cb

#include <*modbus.h*>

struct modbus_iface_param

#include <*modbus.h*> User parameter structure to configure Modbus interface as client or server.

Public Members

enum *modbus_mode* mode

Mode of the interface.

uint32_t **rx_timeout**

Amount of time client will wait for a response from the server.

struct *modbus_serial_param* **serial**

Serial support parameter of the interface.

struct *modbus_raw_cb* **rawcb**

Pointer to raw ADU callback function.

4.17 Modem modules

This service provides modules necessary to communicate with modems.

Modems are self-contained devices that implement the hardware and software necessary to perform RF (Radio-Frequency) communication, including GNSS, Cellular, WiFi etc.

The modem modules are inter-connected dynamically using data-in/data-out pipes making them independently testable and highly flexible, ensuring stability and scalability.

4.17.1 Modem pipe

This module is used to abstract data-in/data-out communication over a variety of mechanisms, like UART and CMUX DLCI channels, in a thread-safe manner.

A modem backend will internally contain an instance of a modem_pipe structure, alongside any buffers and additional structures required to abstract away its underlying mechanism.

The modem backend will return a pointer to its internal modem_pipe structure when initialized, which will be used to interact with the backend through the modem pipe API.

group **modem_pipe**

Modem Pipe.

Typedefs

typedef void (***modem_pipe_api_callback**)(struct modem_pipe *pipe, enum *modem_pipe_event* event, void *user_data)

Enums

enum **modem_pipe_event**

Modem pipe event.

Values:

enumerator **MODEM_PIPE_EVENT_OPENED** = 0

enumerator **MODEM_PIPE_EVENT_RECEIVE_READY**

enumerator MODEM_PIPE_EVENT_TRANSMIT_IDLE

enumerator MODEM_PIPE_EVENT_CLOSED

Functions

int modem_pipe_open(struct modem_pipe *pipe)

Open pipe.

Warning

Be cautious when using this synchronous version of the call. It may block the calling thread, which in the case of the system workqueue can result in a deadlock until this call times out waiting for the pipe to be open.

Parameters

- pipe – Pipe instance

Return values

- 0 – if pipe was successfully opened or was already open
- -errno – code otherwise

int modem_pipe_open_async(struct modem_pipe *pipe)

Open pipe asynchronously.

Note

The MODEM_PIPE_EVENT_OPENED event is invoked immediately if pipe is already opened.

Parameters

- pipe – Pipe instance

Return values

- 0 – if pipe open was called successfully or pipe was already open
- -errno – code otherwise

void modem_pipe_attach(struct modem_pipe *pipe, *modem_pipe_api_callback* callback, void *user_data)

Attach pipe to callback.

Note

The MODEM_PIPE_EVENT_RECEIVE_READY event is invoked immediately if pipe has pending data ready to receive.

Parameters

- pipe – Pipe instance

- `callback` – Callback called when pipe event occurs
- `user_data` – Free to use user data passed with callback

int `modem_pipe_transmit`(struct modem_pipe *pipe, const uint8_t *buf, size_t size)

Transmit data through pipe.

Warning

This call must be non-blocking

Parameters

- `pipe` – Pipe to transmit through
- `buf` – Data to transmit
- `size` – Number of bytes to transmit

Return values

- `Number` – of bytes placed in pipe
- `-EPERM` – if pipe is closed
- `-errno` – code on error

int `modem_pipe_receive`(struct modem_pipe *pipe, uint8_t *buf, size_t size)

Receive data through pipe.

Warning

This call must be non-blocking

Parameters

- `pipe` – Pipe to receive from
- `buf` – Destination for received data; must not be already in use in a modem module.
- `size` – Capacity of destination for received data

Return values

- `Number` – of bytes received from pipe
- `-EPERM` – if pipe is closed
- `-errno` – code on error

void `modem_pipe_release`(struct modem_pipe *pipe)

Clear callback.

Parameters

- `pipe` – Pipe instance

int `modem_pipe_close`(struct modem_pipe *pipe)

Close pipe.

Warning

Be cautious when using this synchronous version of the call. It may block the calling thread, which in the case of the system workqueue can result in a deadlock until this call times out waiting for the pipe to be closed.

Parameters

- `pipe` – Pipe instance

Return values

- `0` – if pipe open was called closed or pipe was already closed
- `-errno` – code otherwise

int `modem_pipe_close_async`(struct modem_pipe *pipe)

Close pipe asynchronously.

Note

The MODEM_PIPE_EVENT_CLOSED event is invoked immediately if pipe is already closed.

Parameters

- `pipe` – Pipe instance

Return values

- `0` – if pipe close was called successfully or pipe was already closed
- `-errno` – code otherwise

4.17.2 Modem PPP

This module defines and binds a L2 PPP network interface, described in *L2 Layer Management*, to a modem backend. The L2 PPP interface sends and receives network packets. These network packets have to be wrapped in PPP frames before being transported via a modem backend. This module performs said wrapping.

group `modem_ppp`

Modem PPP.

Defines

`MODEM_PPP_DEFINE`(_name, _init_iface, _prio, _mtu, _buf_size)

Define a modem PPP module and bind it to a network interface.

This macro defines the modem_ppp instance, initializes a PPP L2 network device instance, and binds the modem_ppp instance to the PPP L2 instance.

Parameters

- `_name` – Name of the statically defined modem_ppp instance
- `_init_iface` – Hook for the PPP L2 network interface init function
- `_prio` – Initialization priority of the PPP L2 net iface

- **_mtu** – Max size of *net_pkt* data sent and received on PPP L2 net iface
- **_buf_size** – Size of partial PPP frame transmit and receive buffers

Typedefs

typedef void (***modem_ppp_init_iface**)(struct *net_if* *iface)

L2 network interface init callback.

Functions

int **modem_ppp_attach**(struct modem_ppp *ppp, struct modem_pipe *pipe)

Attach pipe to instance and connect.

Parameters

- **ppp** – Modem PPP instance
- **pipe** – Pipe to attach to modem PPP instance

struct *net_if* ***modem_ppp_get_iface**(struct modem_ppp *ppp)

Get network interface modem PPP instance is bound to.

Parameters

- **ppp** – Modem PPP instance

Returns

Pointer to network interface modem PPP instance is bound to

void **modem_ppp_release**(struct modem_ppp *ppp)

Release pipe from instance.

Parameters

- **ppp** – Modem PPP instance

4.17.3 Modem CMUX

This module is an implementation of CMUX following the 3GPP 27.010 specification. CMUX is a multiplexing protocol, allowing for multiple bi-directional streams of data, called DLCI channels. The module attaches to a single modem backend, exposing multiple modem backends, each representing a DLCI channel.

group **modem_cmux**

Modem CMUX.

Typedefs

typedef void (***modem_cmux_callback**)(struct modem_cmux *cmux, enum *modem_cmux_event* event, void *user_data)

Enums

enum **modem_cmux_event**

Values:

enumerator **MODEM_CMUX_EVENT_CONNECTED** = 0

enumerator **MODEM_CMUX_EVENT_DISCONNECTED**

Functions

void **modem_cmux_init**(struct modem_cmux *cmux, const struct *modem_cmux_config* *config)

Initialize CMUX instance.

Parameters

- **cmux** – CMUX instance
- **config** – Configuration to apply to CMUX instance

struct modem_pipe ***modem_cmux_dlci_init**(struct modem_cmux *cmux, struct modem_cmux_dlci *dlci, const struct *modem_cmux_dlci_config* *config)

Initialize DLCI instance and register it with CMUX instance.

Parameters

- **cmux** – CMUX instance which the DLCI will be registered to
- **dlci** – DLCI instance which will be registered and configured
- **config** – Configuration to apply to DLCI instance

int **modem_cmux_attach**(struct modem_cmux *cmux, struct modem_pipe *pipe)

Attach CMUX instance to pipe.

Parameters

- **cmux** – CMUX instance
- **pipe** – Pipe instance to attach CMUX instance to

int **modem_cmux_connect**(struct modem_cmux *cmux)

Connect CMUX instance.

This will send a CMUX connect request to target on the serial bus. If successful, DLCI channels can be now be opened using *modem_pipe_open()*

> **Note**
>
> When connected, the bus pipe must not be used directly

Parameters

- **cmux** – CMUX instance

int **modem_cmux_connect_async**(struct modem_cmux *cmux)

Connect CMUX instance asynchronously.

This will send a CMUX connect request to target on the serial bus. If successful, DLCI channels can be now be opened using *modem_pipe_open()*.

> **Note**
>
> When connected, the bus pipe must not be used directly

Parameters

- **cmux** – CMUX instance

int **modem_cmux_disconnect**(struct modem_cmux *cmux)

Close down and disconnect CMUX instance.

This will close all open DLCI channels, and close down the CMUX connection.

> **Note**
>
> The bus pipe must be released using *modem_cmux_release()* after disconnecting before being reused.

Parameters

- **cmux** – CMUX instance

int **modem_cmux_disconnect_async**(struct modem_cmux *cmux)

Close down and disconnect CMUX instance asynchronously.

This will close all open DLCI channels, and close down the CMUX connection.

> **Note**
>
> The bus pipe must be released using *modem_cmux_release()* after disconnecting before being reused.

Parameters

- **cmux** – CMUX instance

void **modem_cmux_release**(struct modem_cmux *cmux)

Release CMUX instance from pipe.

Releases the pipe and hard resets the CMUX instance internally. CMUX should be disconnected using *modem_cmux_disconnect()*.

> **Note**
>
> The bus pipe can be used directly again after CMUX instance is released.

Parameters

- **cmux** – CMUX instance

struct **modem_cmux_config**

#include <*cmux.h*> Contains CMUX instance configuration data.

Public Members

modem_cmux_callback **callback**

Invoked when event occurs.

void ***user_data**

Free to use pointer passed to event handler when invoked.

uint8_t ***receive_buf**

Receive buffer.

uint16_t **receive_buf_size**

Size of receive buffer in bytes [127, ...].

uint8_t ***transmit_buf**

Transmit buffer.

uint16_t **transmit_buf_size**

Size of transmit buffer in bytes [149, ...].

struct **modem_cmux_dlci_config**

#include <*cmux.h*> CMUX DLCI configuration.

Public Members

uint8_t **dlci_address**

DLCI channel address.

uint8_t ***receive_buf**

Receive buffer used by pipe.

uint16_t **receive_buf_size**

Size of receive buffer used by pipe [127, ...].

4.17.4 Modem pipelink

This module is used to share modem pipes globally. This module aims to decouple the creation and setup of modem pipes in device drivers from the users of said pipes. See drivers/modem/modem_at_shell.c and drivers/modem/modem_cellular.c for examples of how to use the modem pipelink between device driver and application.

group **modem_pipelink**

Modem pipelink.

MODEM_PIPELINK_DT_INST macros

Device driver instance variants of MODEM_PIPELINK_DT macros

`MODEM_PIPELINK_DT_INST_DECLARE`(inst, name)

`MODEM_PIPELINK_DT_INST_DEFINE`(inst, name)

`MODEM_PIPELINK_DT_INST_GET`(inst, name)

Defines

`MODEM_PIPELINK_DT_DECLARE`(node_id, name)

Declare pipelink from devicetree node identifier and name.

Parameters

- `node_id` – Devicetree node identifier
- `name` – Pipelink name

`MODEM_PIPELINK_DT_DEFINE`(node_id, name)

Define pipelink from devicetree node identifier and name.

Parameters

- `node_id` – Devicetree node identifier
- `name` – Pipelink name

`MODEM_PIPELINK_DT_GET`(node_id, name)

Get pointer to pipelink from devicetree node identifier and name.

Parameters

- `node_id` – Devicetree node identifier
- `name` – Pipelink name

Typedefs

typedef void (*`modem_pipelink_callback`)(struct modem_pipelink *link, enum *modem_pipelink_event* event, void *user_data)

Pipelink callback definition.

Param link
Modem pipelink instance

Param event
Modem pipelink event

Param user_data
User data passed to *modem_pipelink_attach()*

Enums

enum `modem_pipelink_event`

Pipelink event.

Values:

enumerator `MODEM_PIPELINK_EVENT_CONNECTED` = 0

Modem pipe has been connected and can be opened.

enumerator `MODEM_PIPELINK_EVENT_DISCONNECTED`

Modem pipe has been disconnected and can't be opened.

Functions

void `modem_pipelink_attach`(struct modem_pipelink *link, *modem_pipelink_callback* callback, void *user_data)

Attach callback to pipelink.

Parameters

- `link` – Pipelink instance
- `callback` – Pipelink callback
- `user_data` – User data passed to pipelink callback

bool `modem_pipelink_is_connected`(struct modem_pipelink *link)

Check whether pipelink pipe is connected.

Parameters

- `link` – Pipelink instance

Return values

- `true` – if pipe is connected
- `false` – if pipe is not connected

struct modem_pipe *`modem_pipelink_get_pipe`(struct modem_pipelink *link)

Get pipe from pipelink.

Parameters

- `link` – Pipelink instance

Return values

- `Pointer` – to pipe if pipelink has been initialized
- `NULL` – if pipelink has not been initialized

void `modem_pipelink_release`(struct modem_pipelink *link)

Clear callback.

Parameters

- `link` – Pipelink instance

4.18 Asynchronous Notifications

Zephyr APIs often include *async* functions where an operation is initiated and the application needs to be informed when it completes, and whether it succeeded. Using *k_poll()* is often a good method, but some application architectures may be more suited to a callback notification, and operations like enabling clocks and power rails may need to be invoked before kernel functions are available so a busy-wait for completion may be needed.

This API is intended to be embedded within specific subsystems such as *On-Off Manager* and other APIs that support async transactions. The subsystem wrappers are responsible for extracting operation-specific data from requests that include a notification element, and for invoking callbacks with the parameters required by the API.

A limitation is that this API is not suitable for *System Calls* because:

- *sys_notify* is not a kernel object;
- copying the notification content from userspace will break use of *CONTAINER_OF* in the implementing function;
- neither the spin-wait nor callback notification methods can be accepted from userspace callers.

Where a notification is required for an asynchronous operation invoked from a user mode thread the subsystem or driver should provide a syscall API that uses *k_poll_signal* for notification.

4.18.1 API Reference

group `sys_notify_apis`

Typedefs

typedef void (*`sys_notify_generic_callback`)()

Generic signature used to notify of result completion by callback.

Functions with this role may be invoked from any context including pre-kernel, ISR, or cooperative or pre-emptible threads. Compatible functions must be isr-ok and not sleep.

Parameters that should generally be passed to such functions include:

- a pointer to a specific client request structure, i.e. the one that contains the *sys_notify* structure.
- the result of the operation, either as passed to *sys_notify_finalize()* or extracted afterwards using *sys_notify_fetch_result()*. Expected values are service-specific, but the value shall be non-negative if the operation succeeded, and negative if the operation failed.

Functions

static inline uint32_t `sys_notify_get_method`(const struct *sys_notify* *notify)

int **sys_notify_validate**(struct *sys_notify* *notify)

Validate and initialize the notify structure.

This should be invoked at the start of any service-specific configuration validation. It ensures that the basic asynchronous notification configuration is consistent, and clears the result.

Note that this function does not validate extension bits (zeroed by async notify API init functions like *sys_notify_init_callback()*). It may fail to recognize that an uninitialized structure has been passed because only method bits of flags are tested against method settings. To reduce the chance of accepting an uninitialized operation service validation of structures that contain an *sys_notify* instance should confirm that the extension bits are set or cleared as expected.

Return values

- **0** – on successful validation and reinitialization
- **-EINVAL** – if the configuration is not valid.

sys_notify_generic_callback **sys_notify_finalize**(struct *sys_notify* *notify, int res)

Record and signal the operation completion.

Parameters

- **notify** – pointer to the notification state structure.
- **res** – the result of the operation. Expected values are service-specific, but the value shall be non-negative if the operation succeeded, and negative if the operation failed.

Returns

If the notification is to be done by callback this returns the generic version of the function to be invoked. The caller must immediately invoke that function with whatever arguments are expected by the callback. If notification is by spin-wait or signal, the notification has been completed by the point this function returns, and a null pointer is returned.

static inline int **sys_notify_fetch_result**(const struct *sys_notify* *notify, int *result)

Check for and read the result of an asynchronous operation.

Parameters

- **notify** – pointer to the object used to specify asynchronous function behavior and store completion information.
- **result** – pointer to storage for the result of the operation. The result is stored only if the operation has completed.

Return values

- **0** – if the operation has completed.
- **-EAGAIN** – if the operation has not completed.

static inline void **sys_notify_init_spinwait**(struct *sys_notify* *notify)

Initialize a notify object for spin-wait notification.

Clients that use this initialization receive no asynchronous notification, and instead must periodically check for completion using *sys_notify_fetch_result()*.

On completion of the operation the client object must be reinitialized before it can be re-used.

Parameters

- **notify** – pointer to the notification configuration object.

static inline void **sys_notify_init_signal**(struct *sys_notify* *notify, struct *k_poll_signal* *sigp)

Initialize a notify object for (k_poll) signal notification.

Clients that use this initialization will be notified of the completion of operations through the provided signal.

On completion of the operation the client object must be reinitialized before it can be re-used.

Note

This capability is available only when `CONFIG_POLL` is selected.

Parameters

- **notify** – pointer to the notification configuration object.
- **sigp** – pointer to the signal to use for notification. The value must not be null. The signal must be reset before the client object is passed to the on-off service API.

static inline void **sys_notify_init_callback**(struct *sys_notify* *notify, *sys_notify_generic_callback* handler)

Initialize a notify object for callback notification.

Clients that use this initialization will be notified of the completion of operations through the provided callback. Note that callbacks may be invoked from various contexts depending on the specific service; see *sys_notify_generic_callback*.

On completion of the operation the client object must be reinitialized before it can be re-used.

Parameters

- **notify** – pointer to the notification configuration object.
- **handler** – a function pointer to use for notification.

static inline bool **sys_notify_uses_callback**(const struct *sys_notify* *notify)

Detect whether a particular notification uses a callback.

The generic handler does not capture the signature expected by the callback, and the translation to a service-specific callback must be provided by the service. This check allows abstracted services to reject callback notification requests when the service doesn't provide a translation function.

Returns

true if and only if a callback is to be used for notification.

struct **sys_notify**

#include <notify.h> State associated with notification for an asynchronous operation.

Objects of this type are allocated by a client, which must use an initialization function (e.g. *sys_notify_init_signal()*) to configure them. Generally the structure is a member of a service-specific client structure, such as *onoff_client*.

Control of the containing object transfers to the service provider when a pointer to the object is passed to a service function that is documented to take control of the object, such as onoff_service_request(). While the service provider controls the object the client must not change any object fields. Control reverts to the client:

- if the call to the service API returns an error;

- when operation completion is posted. This may occur before the call to the service API returns.

Operation completion is technically posted when the flags field is updated so that *sys_notify_fetch_result()* returns success. This will happen before the signal is posted or callback is invoked. Note that although the manager will no longer reference the *sys_notify* object past this point, the containing object may have state that will be referenced within the callback. Where callbacks are used control of the containing object does not revert to the client until the callback has been invoked. (Re-use within the callback is explicitly permitted.)

After control has reverted to the client the notify object must be reinitialized for the next operation.

The content of this structure is not public API to clients: all configuration and inspection should be done with functions like *sys_notify_init_callback()* and *sys_notify_fetch_result()*. However, services that use this structure may access certain fields directly.

union `method`

#include <notify.h>

Public Members

struct *k_poll_signal* *`signal`

sys_notify_generic_callback `callback`

4.19 Power Management

Zephyr RTOS power management subsystem provides several means for a system integrator to implement power management support that can take full advantage of the power saving features of SOCs.

4.19.1 Overview

The interfaces and APIs provided by the power management subsystem are designed to be architecture and SOC independent. This enables power management implementations to be easily adapted to different SOCs and architectures.

The architecture and SOC independence is achieved by separating the core PM infrastructure from implementations of the SOC specific components. Thus a coherent abstraction is presented to the rest of the OS and the application layer.

The power management features are classified into the following categories.

- System Power Management
- Device Power Management

4.19.2 System Power Management

Introduction

The kernel enters the idle state when it has nothing to schedule. Enabling `CONFIG_PM` allows the kernel to call upon the power management subsystem to put an idle system into one of the supported power states. The kernel requests an amount of time it would like to suspend, then the PM subsystem decides the appropriate power state to transition to based on the configured power management policy.

It is the application's responsibility to set up a wake-up event. A wake-up event will typically be an interrupt triggered by an SoC peripheral module. Examples include a SysTick, RTC, counter, or GPIO. Keep in mind that depending on the SoC and the power mode in question, not all peripherals may be active, and therefore some wake-up sources may not be usable in all power modes.

The following diagram describes system power management:

Power States The power management subsystem defines a set of states described by the power consumption and context retention associated with each of them.

The set of power states is defined by *pm_state*. In general, lower power states (higher index in the enum) will offer greater power savings and have higher wake latencies.

Power Management Policies The power management subsystem supports the following power management policies:

- Residency based
- Application defined

The policy manager is the component of the power management subsystem responsible for deciding which power state the system should transition to. The policy manager can only choose between states that have been defined for the platform. Other constraints placed upon the decision may include locks disallowing certain power states, or various kinds of minimum and maximum latency values, depending on the policy.

More details on the states definition can be found in the `zephyr,power-state` binding documentation.

Residency Under the residency policy, the system will enter the power state which offers the highest power savings, with the constraint that the sum of the minimum residency value (see `zephyr,power-state`) and the latency to exit the mode must be less than or equal to the system idle time duration scheduled by the kernel.

Thus the core logic can be summarized with the following expression:

```
if (time_to_next_scheduled_event >= (state.min_residency_us + state.exit_latency)) {
   return state
}
```

Application The application defines the power management policy by implementing the `pm_policy_next_state()` function. In this policy, the application is free to decide which power state the system should transition to based on the remaining time until the next scheduled timeout.

An example of an application that defines its own policy can be found in tests/subsys/pm/power_mgmt/.

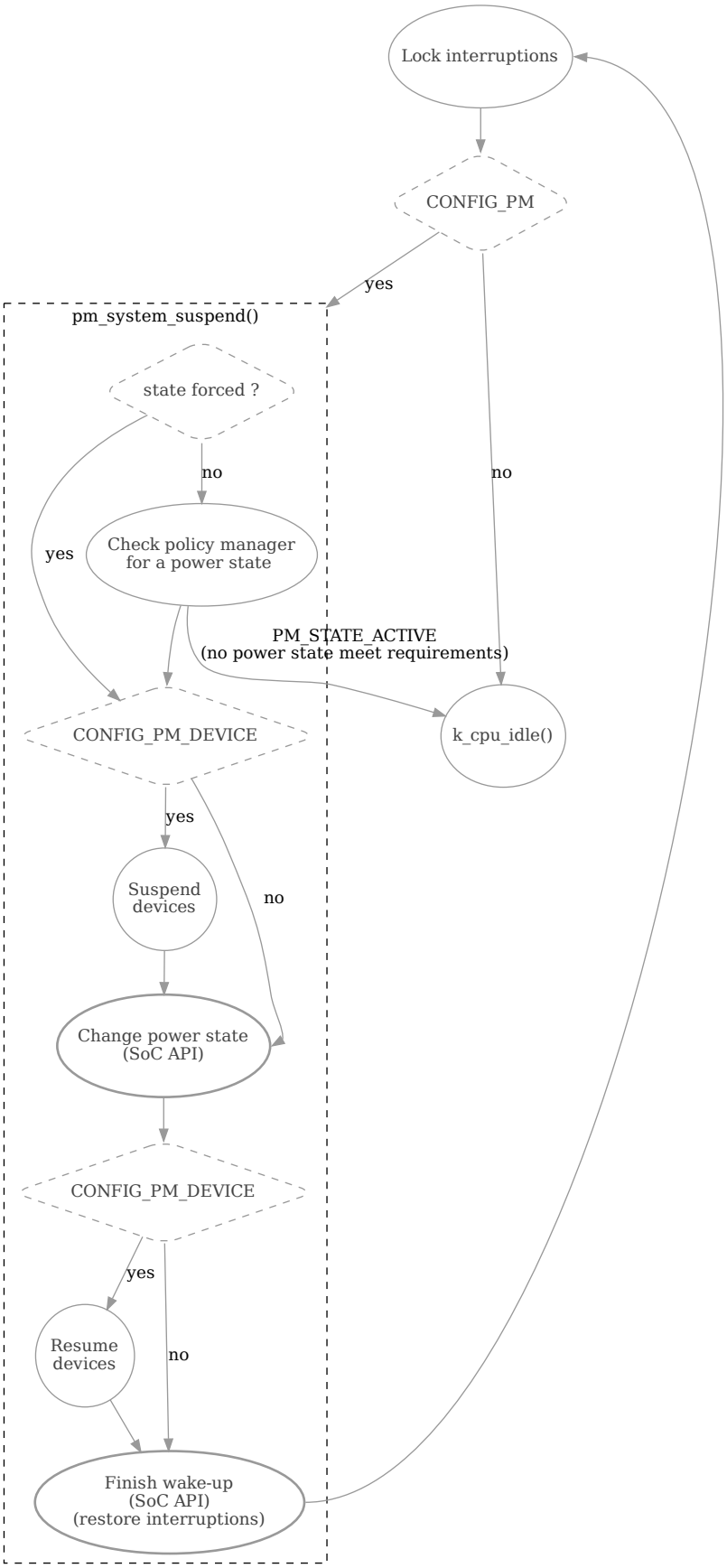


Fig. 7: System power management

Policy and Power States The power management subsystem allows different Zephyr components and applications to configure the policy manager to block the system from transitioning into certain power states. This can be used by devices when executing tasks in background to prevent the system from going to a specific state where it would lose context. See *pm_policy_state_lock_get()*.

Examples Some helpful examples showing different power management features:

- samples/boards/stm32/power_mgmt/blinky/
- samples/boards/esp32/deep_sleep/
- samples/subsys/pm/device_pm/
- tests/subsys/pm/power_mgmt/
- tests/subsys/pm/power_mgmt_soc/
- tests/subsys/pm/power_states_api/

4.19.3 Device Power Management

Introduction

Device power management (PM) on Zephyr is a feature that enables devices to save energy when they are not being used. This feature can be enabled by setting `CONFIG_PM_DEVICE` to `y`. When this option is selected, device drivers implementing power management will be able to take advantage of the device power management subsystem.

Zephyr supports two methods of device power management:

- *Device Runtime Power Management*
- *System-Managed Device Power Management*

Device Runtime Power Management Device runtime power management involves coordinated interaction between device drivers, subsystems, and applications. While device drivers play a crucial role in directly controlling the power state of devices, the decision to suspend or resume a device can also be influenced by higher layers of the software stack.

Each layer—device drivers, subsystems, and applications—can operate independently without needing to know about the specifics of the other layers because the subsystem uses reference count to check when it needs to suspend or resume a device.

- **Device drivers** are responsible for managing the power state of devices. They interact directly with the hardware to put devices into low-power states (suspend) when they are not in use, and bring them back (resume) when needed. Drivers should use the *device runtime power management APIs* provided by Zephyr to control the power state of devices.
- **Subsystems**, such as sensors, file systems, and network, can also influence device power management. Subsystems may have better knowledge about the overall system state and workload, allowing them to make informed decisions about when to suspend or resume devices. For example, a networking subsystem may decide to keep a network interface powered on if it expects network activity in the near future.
- **Applications** running on Zephyr can impact device power management as well. An application may have specific requirements regarding device usage and power consumption. For example, an application that streams data over a network may need to keep the network interface powered on continuously.

Coordination between device drivers, subsystems, and applications is key to efficient device power management. For example, a device driver may not know that a subsystem will perform a series of sequential operations that require a device to remain powered on. In such cases, the subsystem can use device runtime power management to ensure that the device remains in an active state until the operations are complete.

When using this Device Runtime Power Management, the System Power Management subsystem is able to change power states quickly because it does not need to spend time suspending and resuming devices that are runtime enabled.

For more information, see *Device Runtime Power Management*.

System-Managed Device Power Management The system managed device power management (PM) framework is a method where devices are suspended along with the system entering a CPU (or SoC) power state. It can be enabled by setting `CONFIG_PM_DEVICE_SYSTEM_MANAGED`. When using this method, device power management is mostly done inside `pm_system_suspend()`.

If a decision to enter a CPU lower power state is made, the power management subsystem will check if the selected low power state triggers device power management and then suspend devices before changing state. The subsystem takes care of suspending devices following their initialization order, ensuring that possible dependencies between them are satisfied. As soon as the CPU wakes up from a sleep state, devices are resumed in the opposite order that they were suspended.

The decision about suspending devices when entering a low power state is done based on the state and if it has set the property `zephyr,pm-device-disabled`. Here is an example of a target with two low power states with only one triggering device power management:

```
/* Node in a DTS file */
cpus {
     power-states {
             state0: state0 {
                     compatible = "zephyr,power-state";
                     power-state-name = "standby";
                     min-residency-us = <5000>;
                     exit-latency-us = <240>;
                     zephyr,pm-device-disabled;
             };
             state1: state1 {
                     compatible = "zephyr,power-state";
                     power-state-name = "suspend-to-ram";
                     min-residency-us = <8000>;
                     exit-latency-us = <360>;
             };
     };
};
```

Note

When using *System Power Management*, device transitions can be run from the idle thread. As functions in this context cannot block, transitions that intend to use blocking APIs **must** check whether they can do so with *k_can_yield()*.

This method of device power management can be useful in the following scenarios:

- Systems with no device requiring any blocking operations when suspending and resuming. This implementation is reasonably simpler than device runtime power management.
- For devices that can not make any power management decision and have to be always active. For example a firmware using Zephyr that is controlled by an external entity (e.g

Host CPU). In this scenario, some devices have to be always active and should be suspended together with the SoC when requested by this external entity.

It is important to emphasize that this method has drawbacks (see above) and *Device Runtime Power Management* is the **preferred** method for implementing device power management.

Note

When using this method of device power management, the CPU will not enter a low-power state if a device cannot be suspended. For example, if a device returns an error such as `-EBUSY` in response to the `PM_DEVICE_ACTION_SUSPEND` action, indicating it is in the middle of a transaction that cannot be interrupted. Another condition that prevents the CPU from entering a low-power state is if the option `CONFIG_PM_NEED_ALL_DEVICES_IDLE` is set and a device is marked as busy.

Note

Devices are suspended only when the last active core is entering a low power state and devices are resumed by the first core that becomes active.

Device Power Management States

The power management subsystem defines device states in *pm_device_state*. This method is used to track power states of a particular device. It is important to emphasize that, although the state is tracked by the subsystem, it is the responsibility of each device driver to handle device actions(*pm_device_action*) which change device state.

Each *pm_device_action* have a direct an unambiguous relationship with a *pm_device_state*.

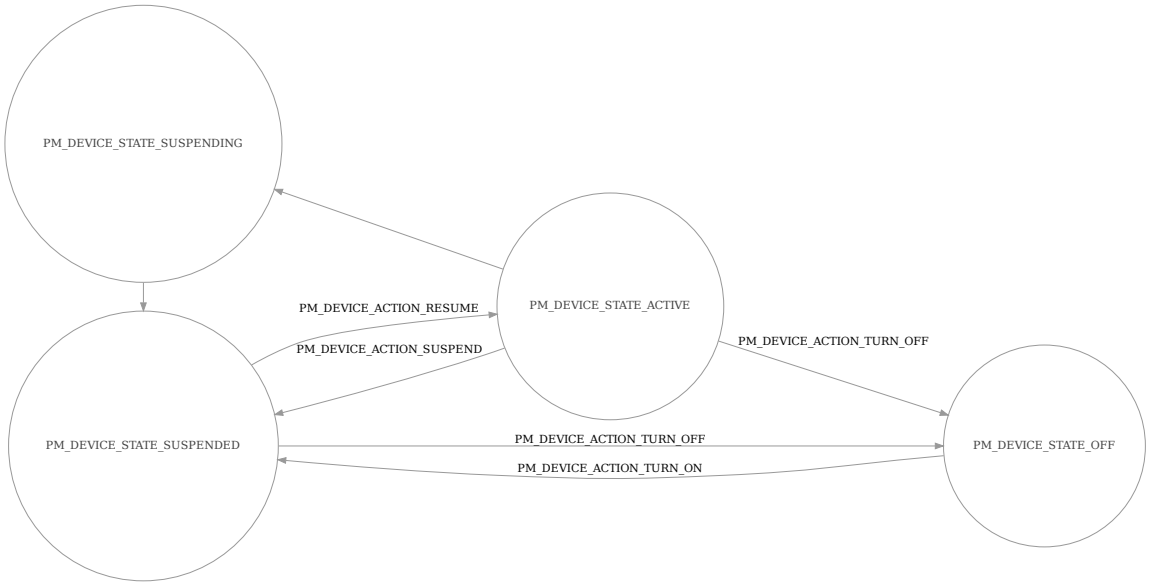


Fig. 8: Device actions x states

As mentioned above, device drivers do not directly change between these states. This is entirely done by the power management subsystem. Instead, drivers are responsible for implementing any hardware-specific tasks needed to handle state changes.

Device Model with Power Management Support

Drivers initialize devices using macros. See *Device Driver Model* for details on how these macros are used. A driver which implements device power management support must provide these macros with arguments that describe its power management implementation.

Use *PM_DEVICE_DEFINE* or *PM_DEVICE_DT_DEFINE* to define the power management resources required by a driver. These macros allocate the driver-specific state which is required by the power management subsystem.

Drivers can use *PM_DEVICE_GET* or *PM_DEVICE_DT_GET* to get a pointer to this state. These pointers should be passed to DEVICE_DEFINE or DEVICE_DT_DEFINE to initialize the power management field in each *device*.

Here is some example code showing how to implement device power management support in a device driver.

```
#define DT_DRV_COMPAT dummy_device

static int dummy_driver_pm_action(const struct device *dev,
                                  enum pm_device_action action)
{
    switch (action) {
    case PM_DEVICE_ACTION_SUSPEND:
        /* suspend the device */
        ...
        break;
    case PM_DEVICE_ACTION_RESUME:
        /* resume the device */
        ...
        break;
    case PM_DEVICE_ACTION_TURN_ON:
        /*
         * powered on the device, used when the power
         * domain this device belongs is resumed.
         */
        ...
        break;
    case PM_DEVICE_ACTION_TURN_OFF:
        /*
         * power off the device, used when the power
         * domain this device belongs is suspended.
         */
        ...
        break;
    default:
        return -ENOTSUP;
    }

    return 0;
}

PM_DEVICE_DT_INST_DEFINE(0, dummy_driver_pm_action);

DEVICE_DT_INST_DEFINE(0, &dummy_init,
    PM_DEVICE_DT_INST_GET(0), NULL, NULL, POST_KERNEL,
    CONFIG_KERNEL_INIT_PRIORITY_DEFAULT, NULL);
```

Shell Commands

Power management actions can be triggered from shell commands for testing purposes. To do that, enable the CONFIG_PM_DEVICE_SHELL option and issue a pm command on a device from the

shell, for example:

```
uart:~$ device list
- buttons (active)
uart:~$ pm suspend buttons
uart:~$ device list
devices:
- buttons (suspended)
```

To print the power management state of a device, enable `CONFIG_DEVICE_SHELL` and use the `device list` command, for example:

```
uart:~$ device list
devices:
- i2c@40003000 (active)
- buttons (active, usage=1)
- leds (READY)
```

In this case, `leds` does not support PM, `i2c` supports PM with manual suspend and resume actions and it's currently active, `buttons` supports runtime PM and it's currently active with one user.

Busy Status Indication

When the system is idle and the SoC is going to sleep, the power management subsystem can suspend devices, as described in *System-Managed Device Power Management*. This can cause device hardware to lose some states. Suspending a device which is in the middle of a hardware transaction, such as writing to a flash memory, may lead to undefined behavior or inconsistent states. This API guards such transactions by indicating to the kernel that the device is in the middle of an operation and should not be suspended.

When *pm_device_busy_set()* is called, the device is marked as busy and the system will not do power management on it. After the device is no longer doing an operation and can be suspended, it should call *pm_device_busy_clear()*.

Device Power Management X System Power Management

When managing power in embedded systems, it's crucial to understand the interplay between device power state and the overall system power state. Some devices may have dependencies on the system power state. For example, certain low-power states of the SoC might not supply power to peripheral devices, leading to problems if the device is in the middle of an operation. Proper coordination is essential to maintain system stability and data integrity.

To avoid this sort of problem, devices must *get and release lock* power states that cause power loss during an operation.

Zephyr provides a mechanism for devices to declare which power states cause power loss and an API that automatically get and put lock on them. This feature is enabled setting `CONFIG_PM_POLICY_DEVICE_CONSTRAINTS` to `y`.

Once this feature is enabled, devices must declare in devicetree which states cause power loss. In the following example, device `test_dev` says that power states `state1` and `state2` cause power loss.

```
power-states {
        state0: state0 {
                compatible = "zephyr,power-state";
                power-state-name = "suspend-to-idle";
                min-residency-us = <10000>;
                exit-latency-us = <100>;
```

(continues on next page)

(continued from previous page)

```
        };

        state1: state1 {
                compatible = "zephyr,power-state";
                power-state-name = "standby";
                min-residency-us = <20000>;
                exit-latency-us = <200>;
        };

        state2: state2 {
                compatible = "zephyr,power-state";
                power-state-name = "suspend-to-ram";
                min-residency-us = <50000>;
                exit-latency-us = <500>;
        };

        state3: state3 {
                compatible = "zephyr,power-state";
                power-state-name = "suspend-to-ram";
                status = "disabled";
        };
};

test_dev: test_dev {
        compatible = "test-device-pm";
        status = "okay";
        zephyr,disabling-power-states = <&state1 &state2>;
};
```

After that devices can lock these state calling *pm_policy_device_power_lock_get()* and release with *pm_policy_device_power_lock_put()*. For example:

```
static void timer_expire_cb(struct k_timer *timer)
{
       struct test_driver_data *data = k_timer_user_data_get(timer);

       data->ongoing = false;
       k_timer_stop(timer);
       pm_policy_device_power_lock_put(data->self);
}

void test_driver_async_operation(const struct device *dev)
{
       struct test_driver_data *data = dev->data;

       data->ongoing = true;
       pm_policy_device_power_lock_get(dev);

       /** Lets set a timer big enough to ensure that any deep
        *  sleep state would be suitable but constraints will
        *  make only state0 (suspend-to-idle) will be used.
        */
       k_timer_start(&data->timer, K_MSEC(500), K_NO_WAIT);
}
```

Wakeup capability

Some devices are capable of waking the system up from a sleep state. When a device has such capability, applications can enable or disable this feature on a device dynamically using *pm_device_wakeup_enable()*.

This property can be set on device declaring the property `wakeup-source` in the device node in devicetree. For example, this devicetree fragment sets the `gpio0` device as a "wakeup" source:

```
gpio0: gpio@40022000 {
        compatible = "ti,cc13xx-cc26xx-gpio";
        reg = <0x40022000 0x400>;
        interrupts = <0 0>;
        status = "disabled";
        label = "GPIO_0";
        gpio-controller;
        wakeup-source;
        #gpio-cells = <2>;
};
```

By default, "wakeup" capable devices do not have this functionality enabled during the device initialization. Applications can enable this functionality later calling *pm_device_wakeup_enable()*.

Note

This property is **only** used by the system power management to identify devices that should not be suspended. It is responsibility of driver or the application to do any additional configuration required by the device to support it.

Examples

Some helpful examples showing device power management features:

- samples/subsys/pm/device_pm/
- tests/subsys/pm/power_mgmt/
- tests/subsys/pm/device_wakeup_api/
- tests/subsys/pm/device_driver_init/

4.19.4 Device Runtime Power Management

Introduction

The device runtime power management (PM) framework is an active power management mechanism which reduces the overall system power consumption by suspending the devices which are idle or not used independently of the system state. It can be enabled by setting `CONFIG_PM_DEVICE_RUNTIME`. In this model the device driver is responsible to indicate when it needs the device and when it does not. This information is used to determine when to suspend or resume a device based on usage count.

When device runtime power management is enabled on a device, its state will be initially set to a *PM_DEVICE_STATE_SUSPENDED* indicating it is not used. On the first device request, it will be resumed and so put into the *PM_DEVICE_STATE_ACTIVE* state. The device will remain in this state until it is no longer used. At this point, the device will be suspended until the next device request. If the suspension is performed synchronously the device will be immediately put into the *PM_DEVICE_STATE_SUSPENDED* state, whereas if it is performed asynchronously, it will be put into the *PM_DEVICE_STATE_SUSPENDING* state first and then into the *PM_DEVICE_STATE_SUSPENDED* state when the action is run.

For devices on a power domain (via the devicetree 'power-domain' property), device runtime power management automatically attempts to request and release the dependent domain in response to *pm_device_runtime_get()* and *pm_device_runtime_put()* calls on the child device.

For the previous to automatically control the power domain state, device runtime PM must be enabled on the power domain device (either through the *zephyr,pm-device-runtime-auto* device-tree property or `pm_device_runtime_enable()`).

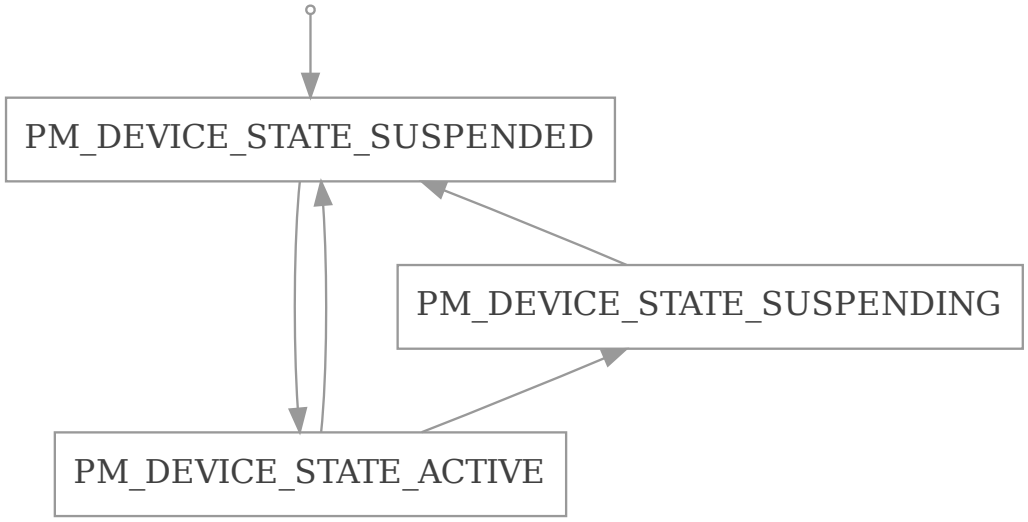


Fig. 9: Device states and transitions

The device runtime power management framework has been designed to minimize devices power consumption with minimal application work. Device drivers are responsible for indicating when they need the device to be operational and when they do not. Therefore, applications can not manually suspend or resume a device. An application can, however, decide when to disable or enable runtime power management for a device. This can be useful, for example, if an application wants a particular device to be always active.

Design principles

When runtime PM is enabled on a device it will no longer be resumed or suspended during system power transitions. Instead, the device is fully responsible to indicate when it needs a device and when it does not. The device runtime PM API uses reference counting to keep track of device's usage. This allows the API to determine when a device needs to be resumed or suspended. The API uses the *get* and *put* terminology to indicate when a device is needed or not, respectively. This mechanism plays a key role when we account for device dependencies. For example, if a bus device is used by multiple sensors, we can keep the bus active until the last sensor has finished using it.

Note

As of today, the device runtime power management API does not manage device dependencies. This effectively means that, if a device depends on other devices to operate (e.g. a sensor may depend on a bus device), the bus will be resumed and suspended on every transaction. In general, it is more efficient to keep parent devices active when their children are used, since the children may perform multiple transactions in a short period of time. Until this feature is added, devices can manually *get* or *put* their dependencies.

The *pm_device_runtime_get()* function can be used by a device driver to indicate it *needs* the device to be active or operational. This function will increase device usage count and resume the device if necessary. Similarly, the *pm_device_runtime_put()* function can be used to indicate that the device is no longer needed. This function will decrease the device usage count and suspend the device if necessary. It is worth to note that in both cases, the operation is carried out synchronously. The sequence diagram shown below illustrates how a device can use this API and the expected sequence of events.

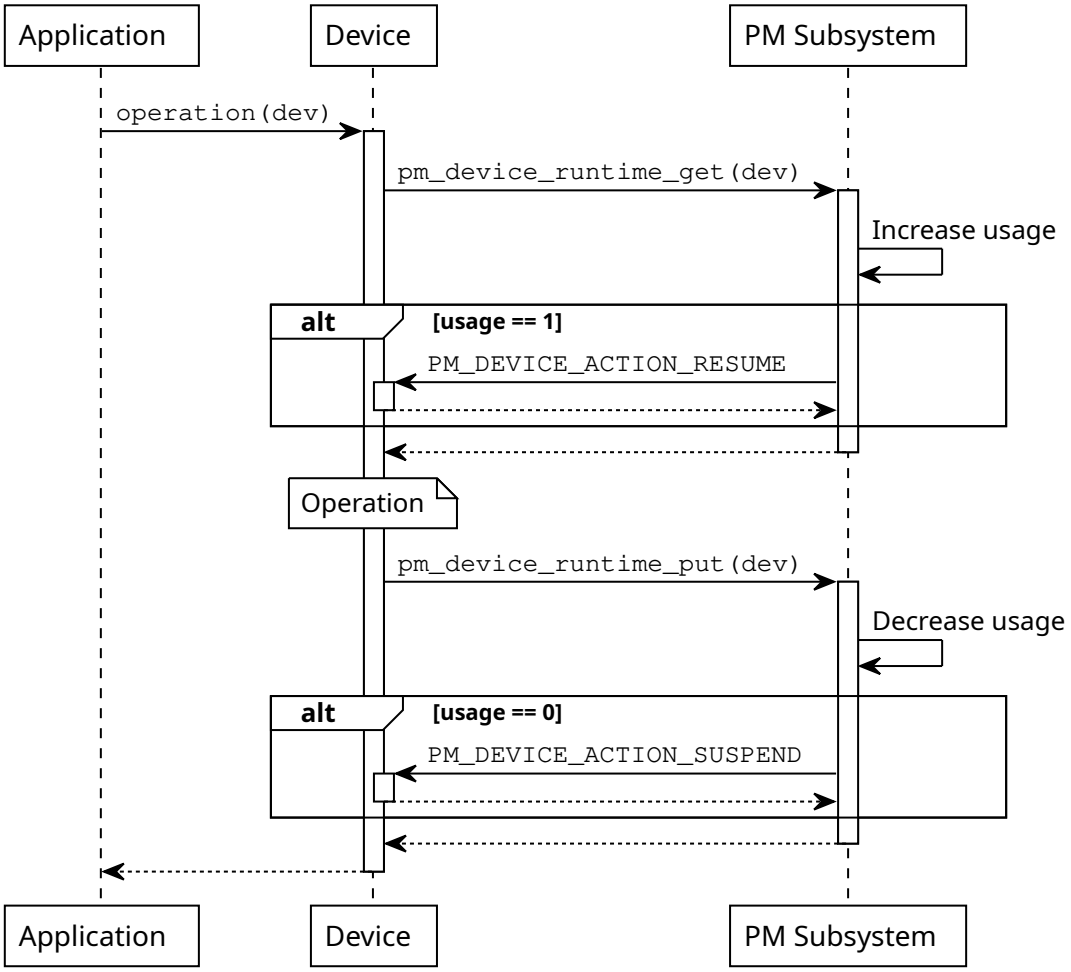


Fig. 10: Synchronous operation on a single device

The synchronous model is as simple as it gets. However, it may introduce unnecessary delays since the application will not get the operation result until the device is suspended (in case device is no longer used). It will likely not be a problem if the operation is fast, e.g. a register toggle. However, the situation will not be the same if suspension involves sending packets through a slow bus. For this reason the device drivers can also make use of the *pm_device_runtime_put_async()* function. This function will schedule the suspend operation, again, if device is no longer used. The suspension will then be carried out when the system work queue gets the chance to run. The sequence diagram shown below illustrates this scenario.

Implementation guidelines

In a first place, a device driver needs to implement the PM action callback used by the PM subsystem to suspend or resume devices.

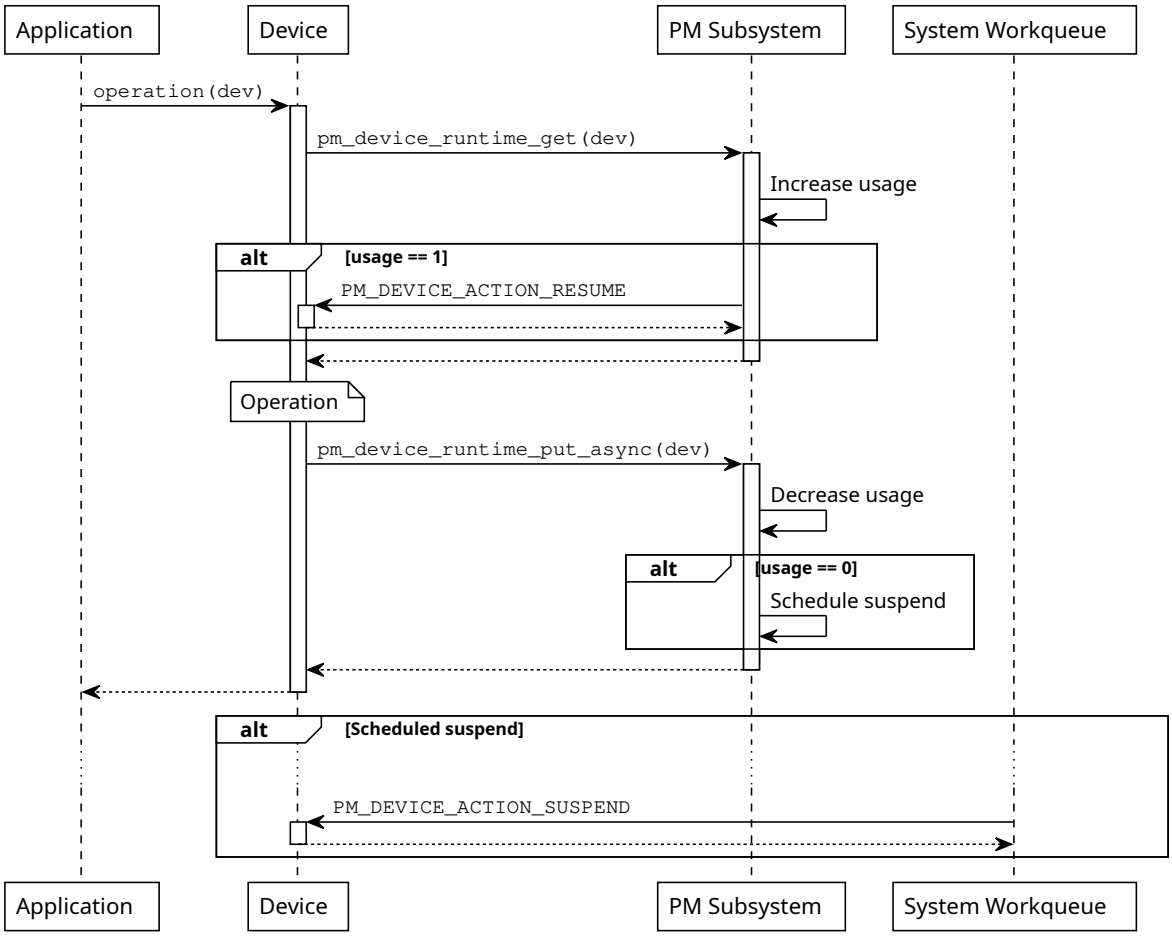


Fig. 11: Asynchronous operation on a single device

```
static int mydev_pm_action(const struct device *dev,
                           enum pm_device_action *action)
{
    switch (action) {
    case PM_DEVICE_ACTION_SUSPEND:
        /* suspend the device */
        ...
        break;
    case PM_DEVICE_ACTION_RESUME:
        /* resume the device */
        ...
        break;
    default:
        return -ENOTSUP;
    }

    return 0;
}
```

The PM action callback calls are serialized by the PM subsystem, therefore, no special synchronization is required.

To enable device runtime power management on a device, the driver needs to call *pm_device_runtime_enable()* at initialization time. Note that this function will suspend the device if its state is *PM_DEVICE_STATE_ACTIVE*. In case the device is physically suspended, the init function should call *pm_device_init_suspended()* before calling *pm_device_runtime_enable()*.

```
/* device driver initialization function */
static int mydev_init(const struct device *dev)
{
    int ret;
    ...

    /* OPTIONAL: mark device as suspended if it is physically suspended */
    pm_device_init_suspended(dev);

    /* enable device runtime power management */
    ret = pm_device_runtime_enable(dev);
    if ((ret < 0) && (ret != -ENOSYS)) {
        return ret;
    }
}
```

Device runtime power management can also be automatically enabled on a device instance by adding the `zephyr,pm-device-runtime-auto` flag onto the corresponding devicetree node. If enabled, *pm_device_runtime_enable()* is called immediately after the `init` function of the device runs and returns successfully.

```
foo {
    /* ... */
    zephyr,pm-device-runtime-auto;
};
```

Assuming an example device driver that implements an `operation` API call, the *get* and *put* operations could be carried out as follows:

```
static int mydev_operation(const struct device *dev)
{
    int ret;

    /* "get" device (increases usage count, resumes device if suspended) */
    ret = pm_device_runtime_get(dev);
```

(continues on next page)

(continued from previous page)

```
    if (ret < 0) {
        return ret;
    }

    /* do something with the device */
    ...

    /* "put" device (decreases usage count, suspends device if no more users) */
    return pm_device_runtime_put(dev);
}
```

In case the suspend operation is *slow*, the device driver can use the asynchronous API:

```
static int mydev_operation(const struct device *dev)
{
    int ret;

    /* "get" device (increases usage count, resumes device if suspended) */
    ret = pm_device_runtime_get(dev);
    if (ret < 0) {
        return ret;
    }

    /* do something with the device */
    ...

    /* "put" device (decreases usage count, schedule suspend if no more users) */
    return pm_device_runtime_put_async(dev, K_NO_WAIT);
}
```

Examples

Some helpful examples showing device runtime power management features:

- tests/subsys/pm/device_runtime_api/
- tests/subsys/pm/device_power_domains/
- tests/subsys/pm/power_domain/

4.19.5 Power Domain

Introduction

The Zephyr power domain abstraction is designed to support groupings of devices powered by a common source to be notified of power source state changes in a generic fashion. Application code that is using device A does not need to know that device B is on the same power domain and should also be configured into a low power state.

Power domains are optional on Zephyr, to enable this feature the option `CONFIG_PM_DEVICE_POWER_DOMAIN` has to be set.

When a power domain turns itself on or off, it is the responsibility of the power domain to notify all devices using it through their power management callback called with *PM_DEVICE_ACTION_TURN_ON* or *PM_DEVICE_ACTION_TURN_OFF* respectively. This work flow is illustrated in the diagram below.

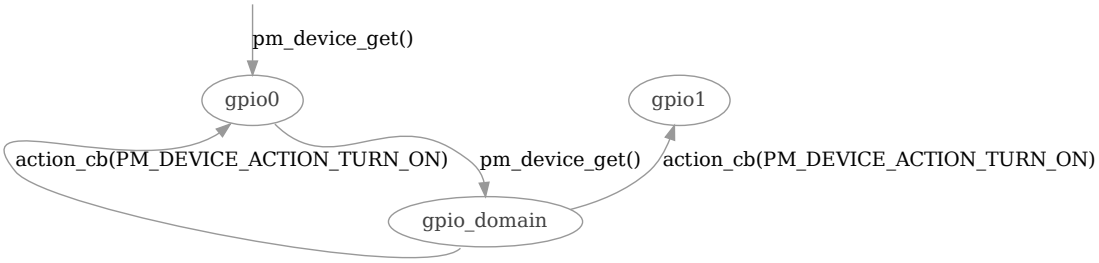


Fig. 12: Power domain work flow

Internal Power Domains Most of the devices in an SoC have independent power control that can be turned on or off to reduce power consumption. But there is a significant amount of static current leakage that can't be controlled only using device power management. To solve this problem, SoCs normally are divided into several regions grouping devices that are generally used together, so that an unused region can be completely powered off to eliminate this leakage. These regions are called "power domains", can be present in a hierarchy and can be nested.

External Power Domains Devices external to a SoC can be powered from sources other than the main power source of the SoC. These external sources are typically a switch, a regulator, or a dedicated power IC. Multiple devices can be powered from the same source, and this grouping of devices is typically called a "power domain".

Placing devices on power domains can be done for a variety of reasons, including to enable devices with high power consumption in low power mode to be completely turned off when not in use.

Implementation guidelines

In a first place, a device that acts as a power domain needs to declare compatible with `power-domain`. Taking *Power domain work flow* as example, the following code defines a domain called `gpio_domain`.

```
gpio_domain: gpio_domain@4 {
        compatible = "power-domain";
        ...
};
```

A power domain needs to implement the PM action callback used by the PM subsystem to turn devices on and off.

```
static int mydomain_pm_action(const struct device *dev,
                              enum pm_device_action *action)
{
    switch (action) {
    case PM_DEVICE_ACTION_RESUME:
        /* resume the domain */
        ...
        /* notify children domain is now powered */
        pm_device_children_action_run(dev, PM_DEVICE_ACTION_TURN_ON, NULL);
        break;
    case PM_DEVICE_ACTION_SUSPEND:
```

(continues on next page)

(continued from previous page)

```
        /* notify children domain is going down */
        pm_device_children_action_run(dev, PM_DEVICE_ACTION_TURN_OFF, NULL);
        /* suspend the domain */
        ...
        break;
    case PM_DEVICE_ACTION_TURN_ON:
        /* turn on the domain (e.g. setup control pins to disabled) */
        ...
        break;
    case PM_DEVICE_ACTION_TURN_OFF:
        /* turn off the domain (e.g. reset control pins to default state) */
        ...
        break;
    default:
        return -ENOTSUP;
    }

    return 0;
}
```

Devices belonging to this device can be declared referring it in the `power-domain` node's property. The example below declares devices `gpio0` and `gpio1` belonging to domain `gpio_domain`.

```
&gpio0 {
        compatible = "zephyr,gpio-emul";
        gpio-controller;
        power-domain = <&gpio_domain>;
};

&gpio1 {
        compatible = "zephyr,gpio-emul";
        gpio-controller;
        power-domain = <&gpio_domain>;
};
```

All devices under a domain will be notified when the domain changes state. These notifications are sent as actions in the device PM action callback and can be used by them to do any additional work required. They can safely be ignored though.

```
static int mydev_pm_action(const struct device *dev,
                           enum pm_device_action *action)
{
    switch (action) {
    case PM_DEVICE_ACTION_SUSPEND:
        /* suspend the device */
        ...
        break;
    case PM_DEVICE_ACTION_RESUME:
        /* resume the device */
        ...
        break;
    case PM_DEVICE_ACTION_TURN_ON:
        /* configure the device into low power mode */
        ...
        break;
    case PM_DEVICE_ACTION_TURN_OFF:
        /* prepare the device for power down */
        ...
        break;
    default:
        return -ENOTSUP;
```

(continues on next page)

(continued from previous page)

```
    }

    return 0;
}
```

Note

It is responsibility of driver or the application to set the domain as “wakeup” source if a device depending on it is used as “wakeup” source.

Examples

Some helpful examples showing power domain features:

- tests/subsys/pm/device_power_domains/
- tests/subsys/pm/power_domain/

4.19.6 Power Management APIs

System PM APIs

group `subsys_pm_sys`

System Power Management API.

Since

1.2

Functions

bool `pm_state_force`(uint8_t cpu, const struct *pm_state_info* *info)

Force usage of given power state.

This function overrides decision made by PM policy forcing usage of given power state upon next entry of the idle thread.

Note

This function can only run in thread context

Parameters

- `cpu` – CPU index.
- `info` – Power state which should be used in the ongoing suspend operation.

void `pm_notifier_register`(struct *pm_notifier* *notifier)

Register a power management notifier.

Register the given notifier from the power management notification list.

Parameters

- `notifier` – *pm_notifier* object to be registered.

int `pm_notifier_unregister`(struct *pm_notifier* *notifier)

Unregister a power management notifier.

Remove the given notifier from the power management notification list. After that this object callbacks will not be called.

Parameters

- `notifier` – *pm_notifier* object to be unregistered.

Returns

0 if the notifier was successfully removed, a negative value otherwise.

const struct *pm_state_info* *`pm_state_next_get`(uint8_t cpu)

Gets the next power state that will be used.

This function returns the next power state that will be used by the SoC.

Parameters

- `cpu` – CPU index.

Returns

next *pm_state_info* that will be used

void `pm_system_resume`(void)

Notify exit from kernel sleep.

This function would notify exit from kernel idling if a corresponding pm_system_suspend() notification was handled and did not return PM_STATE_ACTIVE.

This function should be called from the ISR context of the event that caused the exit from kernel idling.

This is required for cpu power states that would require interrupts to be enabled while entering low power states. e.g. C1 in x86. In those cases, the ISR would be invoked immediately after the event wakes up the CPU, before code following the CPU wait, gets a chance to execute. This can be ignored if no operation needs to be done at the wake event notification.

struct `pm_notifier`

#include <pm.h> Power management notifier struct.

This struct contains callbacks that are called when the target enters and exits power states.

As currently implemented the entry callback is invoked when transitioning from PM_STATE_ACTIVE to another state, and the exit callback is invoked when transitioning from a non-active state to PM_STATE_ACTIVE. This behavior may change in the future.

Note

These callbacks can be called from the ISR of the event that caused the kernel exit from idling.

> **Note**
>
> It is not allowed to call *pm_notifier_unregister* or *pm_notifier_register* from these callbacks because they are called with the spin locked in those functions.

Public Members

void (*`state_entry`)(enum *pm_state* state)

Application defined function for doing any target specific operations for power state entry.

void (*`state_exit`)(enum *pm_state* state)

Application defined function for doing any target specific operations for power state exit.

States

group `subsys_pm_states`

System Power Management States.

Defines

`PM_STATE_INFO_DT_INIT`(node_id)

Initializer for struct *pm_state_info* given a DT node identifier with zephyr,power-state compatible.

Parameters

- `node_id` – A node identifier with compatible zephyr,power-state

`PM_STATE_DT_INIT`(node_id)

Initializer for enum pm_state given a DT node identifier with zephyr,power-state compatible.

Parameters

- `node_id` – A node identifier with compatible zephyr,power-state

`DT_NUM_CPU_POWER_STATES`(node_id)

Obtain number of CPU power states supported and enabled by the given CPU node identifier.

Parameters

- `node_id` – A CPU node identifier.

Returns

Number of supported and enabled CPU power states.

`PM_STATE_INFO_LIST_FROM_DT_CPU`(node_id)

Initialize an array of struct *pm_state_info* with information from all the states present and enabled in the given CPU node identifier.

Example devicetree fragment:

```
cpus {
   ...
   cpu0: cpu@0 {
           device_type = "cpu";
           ...
           cpu-power-states = <&state0 &state1>;
   };

   power-states {
           state0: state0 {
                   compatible = "zephyr,power-state";
                   power-state-name = "suspend-to-idle";
                   min-residency-us = <10000>;
                   exit-latency-us = <100>;
           };

           state1: state1 {
                   compatible = "zephyr,power-state";
                   power-state-name = "suspend-to-ram";
                   min-residency-us = <50000>;
                   exit-latency-us = <500>;
                   zephyr,pm-device-disabled;
           };
   };
};
```

Example usage:

```
const struct pm_state_info states[] =
     PM_STATE_INFO_LIST_FROM_DT_CPU(DT_NODELABEL(cpu0));
```

Parameters

- **node_id** – A CPU node identifier.

PM_STATE_LIST_FROM_DT_CPU(node_id)

Initialize an array of struct pm_state with information from all the states present and enabled in the given CPU node identifier.

Example devicetree fragment:

```
cpus {
   ...
   cpu0: cpu@0 {
           device_type = "cpu";
           ...
           cpu-power-states = <&state0 &state1>;
   };

   power-states {
           state0: state0 {
                   compatible = "zephyr,power-state";
                   power-state-name = "suspend-to-idle";
                   min-residency-us = <10000>;
                   exit-latency-us = <100>;
           };

           state1: state1 {
                   compatible = "zephyr,power-state";
                   power-state-name = "suspend-to-ram";
                   min-residency-us = <50000>;
                   exit-latency-us = <500>;
```

(continues on next page)

(continued from previous page)

```
            };
    };
};
```

Example usage:

```
const enum pm_state states[] = PM_STATE_LIST_FROM_DT_CPU(DT_NODELABEL(cpu0));
```

Parameters

- `node_id` – A CPU node identifier.

Enums

enum `pm_state`

Power management state.

Values:

enumerator `PM_STATE_ACTIVE`

Runtime active state.

The system is fully powered and active.

> **Note**
>
> This state is correlated with ACPI G0/S0 state

enumerator `PM_STATE_RUNTIME_IDLE`

Runtime idle state.

Runtime idle is a system sleep state in which all of the cores enter deepest possible idle state and wait for interrupts, no requirements for the devices, leaving them at the states where they are.

> **Note**
>
> This state is correlated with ACPI S0ix state

enumerator `PM_STATE_SUSPEND_TO_IDLE`

Suspend to idle state.

The system goes through a normal platform suspend where it puts all of the cores in deepest possible idle state and *may* puts peripherals into low-power states. No operating state is lost (ie. the cpu core does not lose execution context), so the system can go back to where it left off easily enough.

> **Note**
>
> This state is correlated with ACPI S1 state

enumerator PM_STATE_STANDBY

Standby state.

In addition to putting peripherals into low-power states all non-boot CPUs are powered off. It should allow more energy to be saved relative to suspend to idle, but the resume latency will generally be greater than for that state. But it should be the same state with suspend to idle state on uniprocessor system.

Note

This state is correlated with ACPI S2 state

enumerator PM_STATE_SUSPEND_TO_RAM

Suspend to ram state.

This state offers significant energy savings by powering off as much of the system as possible, where memory should be placed into the self-refresh mode to retain its contents. The state of devices and CPUs is saved and held in memory, and it may require some boot- strapping code in ROM to resume the system from it.

Note

This state is correlated with ACPI S3 state

enumerator PM_STATE_SUSPEND_TO_DISK

Suspend to disk state.

This state offers significant energy savings by powering off as much of the system as possible, including the memory. The contents of memory are written to disk or other non-volatile storage, and on resume it's read back into memory with the help of boot-strapping code, restores the system to the same point of execution where it went to suspend to disk.

Note

This state is correlated with ACPI S4 state

enumerator PM_STATE_SOFT_OFF

Soft off state.

This state consumes a minimal amount of power and requires a large latency in order to return to runtime active state. The contents of system(CPU and memory) will not be preserved, so the system will be restarted as if from initial power-up and kernel boot.

Note

This state is correlated with ACPI G2/S5 state

enumerator PM_STATE_COUNT

Number of power management states (internal use)

Functions

uint8_t **pm_state_cpu_get_all**(uint8_t cpu, const struct *pm_state_info* **states)

Obtain information about all supported states by a CPU.

Parameters

- **cpu** – CPU index.
- **states** – Where to store the list of supported states.

Returns

Number of supported states.

struct **pm_state_info**

#include <*state.h*> Information about a power management state.

Public Members

uint8_t **substate_id**

Some platforms have multiple states that map to one Zephyr power state.

This property allows the platform distinguish them. e.g:

```
power-states {
   state0: state0 {
           compatible = "zephyr,power-state";
           power-state-name = "suspend-to-idle";
           substate-id = <1>;
           min-residency-us = <10000>;
           exit-latency-us = <100>;
   };
   state1: state1 {
           compatible = "zephyr,power-state";
           power-state-name = "suspend-to-idle";
           substate-id = <2>;
           min-residency-us = <20000>;
           exit-latency-us = <200>;
           zephyr,pm-device-disabled;
   };
};
```

bool **pm_device_disabled**

Whether or not this state triggers device power management.

When this property is false the power management subsystem will suspend devices before entering this state and will properly resume them when leaving it.

uint32_t **min_residency_us**

Minimum residency duration in microseconds.

It is the minimum time for a given idle state to be worthwhile energywise.

> **Note**
>
> 0 means that this property is not available for this state.

uint32_t **exit_latency_us**

Worst case latency in microseconds required to exit the idle state.

> **Note**
>
> 0 means that this property is not available for this state.

struct **pm_state_constraint**

#include <*state.h*> Power state information needed to lock a power state.

Public Members

enum *pm_state* **state**

Power management state.

> **See also**
>
> *pm_state*

uint8_t **substate_id**

Power management sub-state.

> **See also**
>
> *pm_state*

Policy

group **subsys_pm_sys_policy**

System Power Management Policy API.

Defines

PM_ALL_SUBSTATES

Special value for 'all substates'.

Typedefs

typedef void (***pm_policy_latency_changed_cb_t**)(int32_t latency)

Callback to notify when maximum latency changes.

Param latency
New maximum latency. Positive value represents latency in microseconds. SYS_FOREVER_US value lifts the latency constraint. Other values are forbidden.

Functions

void **pm_policy_state_lock_get**(enum *pm_state* state, uint8_t substate_id)

Increase a power state lock counter.

A power state will not be allowed on the first call of *pm_policy_state_lock_get()*. Subsequent calls will just increase a reference count, thus meaning this API can be safely used concurrently. A state will be allowed again after *pm_policy_state_lock_put()* is called as many times as *pm_policy_state_lock_get()*.

Note that the PM_STATE_ACTIVE state is always allowed, so calling this API with PM_STATE_ACTIVE will have no effect.

See also

pm_policy_state_lock_put()

Parameters

- **state** – Power state.
- **substate_id** – Power substate ID. Use PM_ALL_SUBSTATES to affect all the substates in the given power state.

void **pm_policy_state_lock_put**(enum *pm_state* state, uint8_t substate_id)

Decrease a power state lock counter.

See also

pm_policy_state_lock_get()

Parameters

- **state** – Power state.
- **substate_id** – Power substate ID. Use PM_ALL_SUBSTATES to affect all the substates in the given power state.

bool **pm_policy_state_lock_is_active**(enum *pm_state* state, uint8_t substate_id)

Check if a power state lock is active (not allowed).

Parameters

- **state** – Power state.
- **substate_id** – Power substate ID. Use PM_ALL_SUBSTATES to affect all the substates in the given power state.

Return values

- **true** – if power state lock is active.

- `false` – if power state lock is not active.

void **pm_policy_latency_request_add**(struct *pm_policy_latency_request* *req, uint32_t value_us)

Add a new latency requirement.

The system will not enter any power state that would make the system to exceed the given latency value.

Parameters

- `req` – Latency request.
- `value_us` – Maximum allowed latency in microseconds.

void **pm_policy_latency_request_update**(struct *pm_policy_latency_request* *req, uint32_t value_us)

Update a latency requirement.

Parameters

- `req` – Latency request.
- `value_us` – New maximum allowed latency in microseconds.

void **pm_policy_latency_request_remove**(struct *pm_policy_latency_request* *req)

Remove a latency requirement.

Parameters

- `req` – Latency request.

void **pm_policy_latency_changed_subscribe**(struct *pm_policy_latency_subscription* *req, *pm_policy_latency_changed_cb_t* cb)

Subscribe to maximum latency changes.

Parameters

- `req` – Subscription request.
- `cb` – Callback function (NULL to disable).

void **pm_policy_latency_changed_unsubscribe**(struct *pm_policy_latency_subscription* *req)

Unsubscribe to maximum latency changes.

Parameters

- `req` – Subscription request.

void **pm_policy_event_register**(struct *pm_policy_event* *evt, uint32_t time_us)

Register an event.

Events in the power-management policy context are defined as any source that will wake up the system at a known time in the future. By registering such event, the policy manager will be able to decide whether certain power states are worth entering or not.

See also

pm_policy_event_unregister

Note

It is mandatory to unregister events once they have happened by using *pm_policy_event_unregister()*. Not doing so is an API contract violation, because the system would continue to consider them as valid events in the *far* future, that is, after the cycle counter rollover.

Parameters

- `evt` – Event.
- `time_us` – When the event will occur, in microseconds from now.

void `pm_policy_event_update`(struct *pm_policy_event* *evt, uint32_t time_us)

Update an event.

See also

pm_policy_event_register

Parameters

- `evt` – Event.
- `time_us` – When the event will occur, in microseconds from now.

void `pm_policy_event_unregister`(struct *pm_policy_event* *evt)

Unregister an event.

See also

pm_policy_event_register

Parameters

- `evt` – Event.

void `pm_policy_device_power_lock_get`(const struct *device* *dev)

Increase power state locks.

Set power state locks in all power states that disable power in the given device.

See also

pm_policy_device_power_lock_put()

See also

pm_policy_state_lock_get()

Parameters

- `dev` – Device reference.

void `pm_policy_device_power_lock_put`(const struct *device* *dev)

Decrease power state locks.

Remove power state locks in all power states that disable power in the given device.

See also

pm_policy_device_power_lock_get()

See also

pm_policy_state_lock_put()

Parameters

- `dev` – Device reference.

struct `pm_policy_latency_subscription`

#include <*policy.h*> Latency change subscription.

Note

All fields in this structure are meant for private usage.

struct `pm_policy_latency_request`

#include <*policy.h*> Latency request.

Note

All fields in this structure are meant for private usage.

struct `pm_policy_event`

#include <*policy.h*> Event.

Note

All fields in this structure are meant for private usage.

Hooks

group `subsys_pm_sys_hooks`

System Power Management Hooks.

Functions

void **pm_state_set**(enum *pm_state* state, uint8_t substate_id)

Put processor into a power state.

This function implements the SoC specific details necessary to put the processor into available power states.

Parameters

- **state** – Power state.
- **substate_id** – Power substate id.

void **pm_state_exit_post_ops**(enum *pm_state* state, uint8_t substate_id)

Do any SoC or architecture specific post ops after sleep state exits.

This function is a place holder to do any operations that may be needed to be done after sleep state exits. Currently it enables interrupts after resuming from sleep state. In future, the enabling of interrupts may be moved into the kernel.

Parameters

- **state** – Power state.
- **substate_id** – Power substate id.

Device PM APIs

group **subsys_pm_device**

Device Power Management API.

Defines

PM_DEVICE_ISR_SAFE

Flag indicating that runtime PM API for the device can be called from any context.

If *PM_DEVICE_ISR_SAFE* flag is used for device definition, it indicates that PM actions are synchronous and can be executed from any context. This approach can be used for cases where suspending and resuming is short as it is executed in the critical section. This mode requires less resources (~80 byte less RAM) and allows to use device runtime PM from any context (including interrupts).

PM_DEVICE_DEFINE(dev_id, pm_action_cb, ...)

Define device PM resources for the given device name.

See also

PM_DEVICE_DT_DEFINE, *PM_DEVICE_DT_INST_DEFINE*

Note

This macro is a no-op if CONFIG_PM_DEVICE is not enabled.

Parameters

- `dev_id` – Device id.
- `pm_action_cb` – PM control callback.
- `...` – Optional flag to indicate that ISR safe. Use *PM_DEVICE_ISR_SAFE* or 0.

`PM_DEVICE_DT_DEFINE`(node_id, pm_action_cb, ...)

Define device PM resources for the given node identifier.

See also

PM_DEVICE_DT_INST_DEFINE, *PM_DEVICE_DEFINE*

Note

This macro is a no-op if `CONFIG_PM_DEVICE` is not enabled.

Parameters

- `node_id` – Node identifier.
- `pm_action_cb` – PM control callback.
- `...` – Optional flag to indicate that device is isr_ok. Use *PM_DEVICE_ISR_SAFE* or 0.

`PM_DEVICE_DT_INST_DEFINE`(idx, pm_action_cb, ...)

Define device PM resources for the given instance.

See also

PM_DEVICE_DT_DEFINE, *PM_DEVICE_DEFINE*

Note

This macro is a no-op if `CONFIG_PM_DEVICE` is not enabled.

Parameters

- `idx` – Instance index.
- `pm_action_cb` – PM control callback.
- `...` – Optional flag to indicate that device is isr_ok. Use *PM_DEVICE_ISR_SAFE* or 0.

`PM_DEVICE_GET`(dev_id)

Obtain a reference to the device PM resources for the given device.

Parameters

- `dev_id` – Device id.

Returns
Reference to the device PM resources (NULL if device CONFIG_PM_DEVICE is disabled).

PM_DEVICE_DT_GET(node_id)

Obtain a reference to the device PM resources for the given node.

Parameters

- node_id – Node identifier.

Returns
Reference to the device PM resources (NULL if device CONFIG_PM_DEVICE is disabled).

PM_DEVICE_DT_INST_GET(idx)

Obtain a reference to the device PM resources for the given instance.

Parameters

- idx – Instance index.

Returns
Reference to the device PM resources (NULL if device CONFIG_PM_DEVICE is disabled).

Typedefs

typedef int (*pm_device_action_cb_t)(const struct *device* *dev, enum *pm_device_action* action)

Device PM action callback.

Param dev
Device instance.

Param action
Requested action.

Retval 0
If successful.

Retval -ENOTSUP
If the requested action is not supported.

Retval Errno
Other negative errno on failure.

typedef bool (*pm_device_action_failed_cb_t)(const struct *device* *dev, int err)

Device PM action failed callback.

Param dev
Device that failed the action.

Param err
Return code of action failure.

Return
True to continue iteration, false to halt iteration.

Enums

enum `pm_device_state`

Device power states.

Values:

enumerator `PM_DEVICE_STATE_ACTIVE`

Device is in active or regular state.

enumerator `PM_DEVICE_STATE_SUSPENDED`

Device is suspended.

> **Note**
>
> Device context may be lost.

enumerator `PM_DEVICE_STATE_SUSPENDING`

Device is being suspended.

enumerator `PM_DEVICE_STATE_OFF`

Device is turned off (power removed).

> **Note**
>
> Device context is lost.

enum `pm_device_action`

Device PM actions.

Values:

enumerator `PM_DEVICE_ACTION_SUSPEND`

Suspend.

enumerator `PM_DEVICE_ACTION_RESUME`

Resume.

enumerator `PM_DEVICE_ACTION_TURN_OFF`

Turn off.

> **Note**
>
> Action triggered only by a power domain.

enumerator `PM_DEVICE_ACTION_TURN_ON`

Turn on.

> **Note**
>
> Action triggered only by a power domain.

Functions

const char *`pm_device_state_str`(enum *pm_device_state* state)

Get name of device PM state.

Parameters

- `state` – State id which name should be returned

int `pm_device_action_run`(const struct *device* *dev, enum *pm_device_action* action)

Run a pm action on a device.

This function calls the device PM control callback so that the device does the necessary operations to execute the given action.

Parameters

- `dev` – Device instance.
- `action` – Device pm action.

Return values

- `0` – If successful.
- `-ENOTSUP` – If requested state is not supported.
- `-EALREADY` – If device is already at the requested state.
- `-EBUSY` – If device is changing its state.
- `-ENOSYS` – If device does not support PM.
- `-EPERM` – If device has power state locked.
- `Errno` – Other negative errno on failure.

void `pm_device_children_action_run`(const struct *device* *dev, enum *pm_device_action* action, *pm_device_action_failed_cb_t* failure_cb)

Run a pm action on all children of a device.

This function calls all child devices PM control callback so that the device does the necessary operations to execute the given action.

Parameters

- `dev` – Device instance.
- `action` – Device pm action.
- `failure_cb` – Function to call if a child fails the action, can be NULL.

int `pm_device_state_get`(const struct *device* *dev, enum *pm_device_state* *state)

Obtain the power state of a device.

Parameters

- `dev` – Device instance.
- `state` – Pointer where device power state will be stored.

Return values

- `0` – If successful.

- `-ENOSYS` – If device does not implement power management.

static inline void `pm_device_init_suspended`(const struct *device* *dev)

Initialize a device state to *PM_DEVICE_STATE_SUSPENDED*.

By default device state is initialized to *PM_DEVICE_STATE_ACTIVE*. However in order to save power some drivers may choose to only initialize the device to the suspended state, or actively put the device into the suspended state. This function can therefore be used to notify the PM subsystem that the device is in *PM_DEVICE_STATE_SUSPENDED* instead of the default.

Parameters

- `dev` – Device instance.

static inline void `pm_device_init_off`(const struct *device* *dev)

Initialize a device state to *PM_DEVICE_STATE_OFF*.

By default device state is initialized to *PM_DEVICE_STATE_ACTIVE*. In general, this makes sense because the device initialization function will resume and configure a device, leaving it operational. However, when power domains are enabled, the device may be connected to a switchable power source, in which case it won't be powered at boot. This function can therefore be used to notify the PM subsystem that the device is in *PM_DEVICE_STATE_OFF* instead of the default.

Parameters

- `dev` – Device instance.

void `pm_device_busy_set`(const struct *device* *dev)

Mark a device as busy.

Devices marked as busy will not be suspended when the system goes into low-power states. This can be useful if, for example, the device is in the middle of a transaction.

See also

pm_device_busy_clear()

Parameters

- `dev` – Device instance.

void `pm_device_busy_clear`(const struct *device* *dev)

Clear a device busy status.

See also

pm_device_busy_set()

Parameters

- `dev` – Device instance.

bool `pm_device_is_any_busy`(void)

Check if any device is busy.

Return values

- `false` – If no device is busy
- `true` – If one or more devices are busy

bool `pm_device_is_busy`(const struct *device* *dev)

Check if a device is busy.

Parameters

- `dev` – Device instance.

Return values

- `false` – If the device is not busy
- `true` – If the device is busy

bool `pm_device_wakeup_enable`(const struct *device* *dev, bool enable)

Enable or disable a device as a wake up source.

A device marked as a wake up source will not be suspended when the system goes into low-power modes, thus allowing to use it as a wake up source for the system.

Parameters

- `dev` – Device instance.
- `enable` – `true` to enable or `false` to disable

Return values

- `true` – If the wakeup source was successfully enabled.
- `false` – If the wakeup source was not successfully enabled.

bool `pm_device_wakeup_is_enabled`(const struct *device* *dev)

Check if a device is enabled as a wake up source.

Parameters

- `dev` – Device instance.

Return values

- `true` – if the wakeup source is enabled.
- `false` – if the wakeup source is not enabled.

bool `pm_device_wakeup_is_capable`(const struct *device* *dev)

Check if a device is wake up capable.

Parameters

- `dev` – Device instance.

Return values

- `true` – If the device is wake up capable.
- `false` – If the device is not wake up capable.

bool `pm_device_on_power_domain`(const struct *device* *dev)

Check if the device is on a switchable power domain.

Parameters

- `dev` – Device instance.

Return values

- `true` – If device is on a switchable power domain.
- `false` – If device is not on a switchable power domain.

int `pm_device_power_domain_add`(const struct *device* *dev, const struct *device* *domain)

Add a device to a power domain.

This function adds a device to a given power domain.

Parameters

- `dev` – Device to be added to the power domain.
- `domain` – Power domain.

Return values

- `0` – If successful.
- `-EALREADY` – If device is already part of the power domain.
- `-ENOSYS` – If the application was built without power domain support.
- `-ENOSPC` – If there is no space available in the power domain to add the device.

int `pm_device_power_domain_remove`(const struct *device* *dev, const struct *device* *domain)

Remove a device from a power domain.

This function removes a device from a given power domain.

Parameters

- `dev` – Device to be removed from the power domain.
- `domain` – Power domain.

Return values

- `0` – If successful.
- `-ENOSYS` – If the application was built without power domain support.
- `-ENOENT` – If device is not in the given domain.

bool `pm_device_is_powered`(const struct *device* *dev)

Check if the device is currently powered.

Parameters

- `dev` – Device instance.

Return values

- `true` – If device is currently powered, or is assumed to be powered (i.e. it does not support PM or is not under a PM domain)
- `false` – If device is not currently powered

int `pm_device_driver_init`(const struct *device* *dev, *pm_device_action_cb_t* action_cb)

Setup a device driver into the lowest valid power mode.

This helper function is intended to be called at the end of a driver init function to automatically setup the device into the lowest power mode. It assumes that the device has been configured as if it is in *PM_DEVICE_STATE_OFF*.

Parameters

- `dev` – Device instance.
- `action_cb` – Device PM control callback function.

Return values

- `0` – On success.

- `-errno` – Error code from *action_cb* on failure.

struct `pm_device_base`

#include <*device.h*> Device PM info.

Structure holds fields which are common for two PM devices: generic and synchronous.

Public Members

atomic_t `flags`

Device PM status flags.

enum *pm_device_state* `state`

Device power state.

pm_device_action_cb_t `action_cb`

Device PM action callback.

uint32_t `usage`

Device usage count.

struct `pm_device`

#include <*device.h*> Runtime PM info for device with generic PM.

Generic PM involves suspending and resuming operations which can be blocking, long lasting or asynchronous. Runtime PM API is limited when used from interrupt context.

Public Members

struct *pm_device_base* `base`

Base info.

const struct *device* *`dev`

Pointer to the device.

struct k_sem `lock`

Lock to synchronize the get/put operations.

struct *k_event* `event`

Event var to listen to the sync request events.

struct *k_work_delayable* `work`

Work object for asynchronous calls.

struct `pm_device_isr`

#include <*device.h*> Runtime PM info for device with synchronous PM.

Synchronous PM can be used with devices which suspend and resume operations can be performed in the critical section as they are short and non-blocking. Runtime PM API can be used from any context in that case.

Public Members

struct *pm_device_base* **base**
Base info.

struct *k_spinlock* **lock**
Lock to synchronize the synchronous get/put operations.

Device Runtime PM APIs

group **subsys_pm_device_runtime**
Device Runtime Power Management API.

Functions

int **pm_device_runtime_auto_enable**(const struct *device* *dev)
Automatically enable device runtime based on devicetree properties.

> **Note**
> Must not be called from application code. See the zephyr,pm-device-runtime-auto property in pm.yaml and z_sys_init_run_level.

Parameters
- **dev** – Device instance.

Return values
- **0** – If the device runtime PM is enabled successfully or it has not been requested for this device in devicetree.
- **-errno** – Other negative errno, result of enabled device runtime PM.

int **pm_device_runtime_enable**(const struct *device* *dev)
Enable device runtime PM.

This function will enable runtime PM on the given device. If the device is in *PM_DEVICE_STATE_ACTIVE* state, the device will be suspended.

> **See also**
> *pm_device_init_suspended()*

Function properties (list may not be complete)
pre-kernel-ok

Parameters
- **dev** – Device instance.

Return values

- `0` – If the device runtime PM is enabled successfully.
- `-EBUSY` – If device is busy.
- `-ENOTSUP` – If the device does not support PM.
- `-errno` – Other negative errno, result of suspending the device.

int `pm_device_runtime_disable`(const struct *device* *dev)

Disable device runtime PM.

If the device is currently suspended it will be resumed.

Function properties (list may not be complete)

pre-kernel-ok

Parameters

- `dev` – Device instance.

Return values

- `0` – If the device runtime PM is disabled successfully.
- `-ENOTSUP` – If the device does not support PM.
- `-errno` – Other negative errno, result of resuming the device.

int `pm_device_runtime_get`(const struct *device* *dev)

Resume a device based on usage count.

This function will resume the device if the device is suspended (usage count equal to 0). In case of a resume failure, usage count and device state will be left unchanged. In all other cases, usage count will be incremented.

If the device is still being suspended as a result of calling *pm_device_runtime_put_async()*, this function will wait for the operation to finish to then resume the device.

Function properties (list may not be complete)

pre-kernel-ok

Note

It is safe to use this function in contexts where blocking is not allowed, e.g. ISR, provided the device PM implementation does not block.

Parameters

- `dev` – Device instance.

Return values

- `0` – If it succeeds. In case device runtime PM is not enabled or not available this function will be a no-op and will also return 0.
- `-EWOUDBLOCK` – If call would block but it is not allowed (e.g. in ISR).
- `-errno` – Other negative errno, result of the PM action callback.

int `pm_device_runtime_put`(const struct *device* *dev)

Suspend a device based on usage count.

This function will suspend the device if the device is no longer required (usage count equal to 0). In case of suspend failure, usage count and device state will be left unchanged. In all other cases, usage count will be decremented (down to 0).

See also

pm_device_runtime_put_async()

Function properties (list may not be complete)

pre-kernel-ok

Parameters

- `dev` – Device instance.

Return values

- `0` – If it succeeds. In case device runtime PM is not enabled or not available this function will be a no-op and will also return 0.
- `-EALREADY` – If device is already suspended (can only happen if get/put calls are unbalanced).
- `-errno` – Other negative errno, result of the action callback.

int `pm_device_runtime_put_async`(const struct *device* *dev, *k_timeout_t* delay)

Suspend a device based on usage count (asynchronously).

This function will schedule the device suspension if the device is no longer required (usage count equal to 0). In all other cases, usage count will be decremented (down to 0).

See also

pm_device_runtime_put()

Function properties (list may not be complete)

pre-kernel-ok , *async* , *isr-ok*

Note

Asynchronous operations are not supported when in pre-kernel mode. In this case, the function will be blocking (equivalent to *pm_device_runtime_put()*).

Parameters

- `dev` – Device instance.
- `delay` – Minimum amount of time before triggering the action.

Return values

- `0` – If it succeeds. In case device runtime PM is not enabled or not available this function will be a no-op and will also return 0.
- `-EBUSY` – If the device is busy.
- `-EALREADY` – If device is already suspended (can only happen if get/put calls are unbalanced).

bool `pm_device_runtime_is_enabled`(const struct *device* *dev)

Check if device runtime is enabled for a given device.

See also

pm_device_runtime_enable()

Function properties (list may not be complete)

pre-kernel-ok

Parameters

- `dev` – Device instance.

Return values

- `true` – If device has device runtime PM enabled.
- `false` – If the device has device runtime PM disabled.

int `pm_device_runtime_usage`(const struct *device* *dev)

Return the current device usage counter.

Parameters

- `dev` – Device instance.

Return values

- `-ENOTSUP` – If the device is not using runtime PM.
- `-ENOSYS` – If the runtime PM is not enabled at all.

Returns

the current usage counter.

4.20 OS Abstraction

OS abstraction layers (OSAL) provide wrapper function APIs that encapsulate common system functions offered by any operating system. These APIs make it easier and quicker to develop for, and port code to multiple software and hardware platforms.

These sections describe the software and hardware abstraction layers supported by the Zephyr RTOS.

4.20.1 POSIX

Overview

The Portable Operating System Interface (POSIX) is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems. Zephyr implements a subset of the standard POSIX API specified by IEEE 1003.1-2017 (also known as POSIX-1.2017).

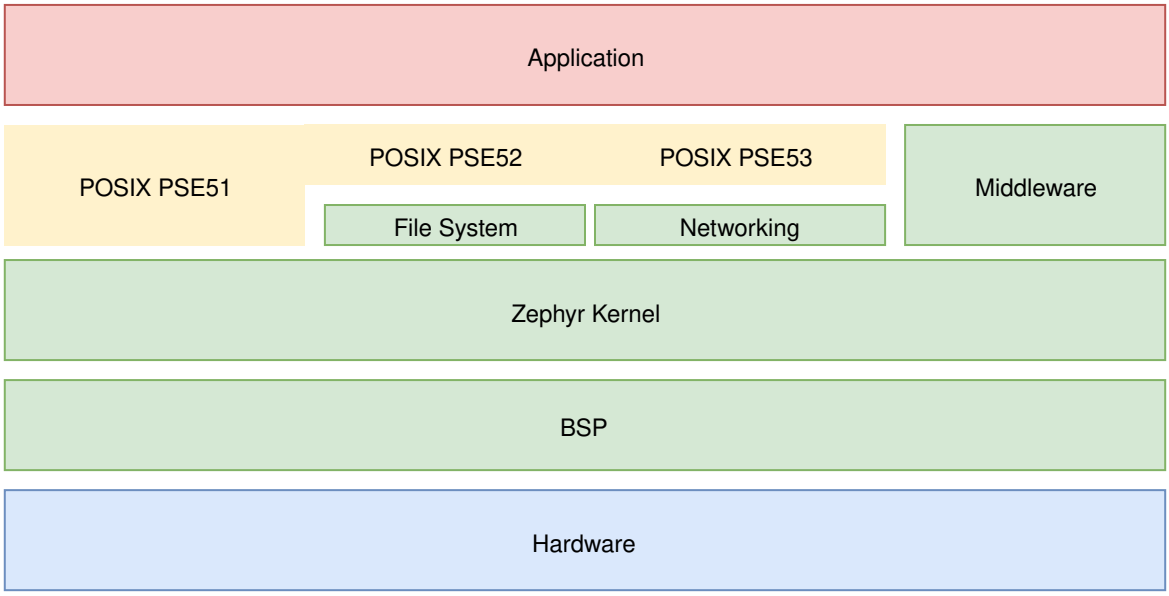


Fig. 13: POSIX support in Zephyr

Note

This page does not document Zephyr's POSIX architecture, which is used to run Zephyr as a native application under the host operating system for prototyping, test, and diagnostic purposes.

With the POSIX support available in Zephyr, an existing POSIX conformant application can be ported to run on the Zephyr kernel, and therefore leverage Zephyr features and functionality. Additionally, a library designed to be POSIX conformant can be ported to Zephyr kernel based applications with no changes.

The POSIX API is an increasingly popular OSAL (operating system abstraction layer) for IoT and embedded applications, as can be seen in Zephyr, AWS:FreeRTOS, TI-RTOS, and NuttX.

Benefits of POSIX support in Zephyr include:

- Offering a familiar API to non-embedded programmers, especially from Linux
- Enabling reuse (portability) of existing libraries based on POSIX APIs
- Providing an efficient API subset appropriate for small (MCU) embedded systems

POSIX Subprofiles While Zephyr supports running multiple *threads* (possibly in an *SMP* configuration), as well as *Virtual Memory and MMUs*, Zephyr code and data normally share a common address space that is partitioned into separate *Memory Domains*. The Zephyr kernel executable code and the application executable code are typically compiled into the same binary artifact. From that perspective, Zephyr apps can be seen as running in the context of a single process.

While multi-purpose operating systems (OS) offer full POSIX conformance, Real-Time Operating Systems (RTOS) such as Zephyr typically serve a fixed-purpose, have limited hardware resources,

and experience limited user interaction. In such systems, full POSIX conformance can be impractical and unnecessary.

For that reason, POSIX defined the following *Application Environment Profiles (AEP)* as part of IEEE 1003.13-2003 (also known as POSIX.13-2003). Each AEP adds incrementally more features over the required *POSIX System Interfaces*.

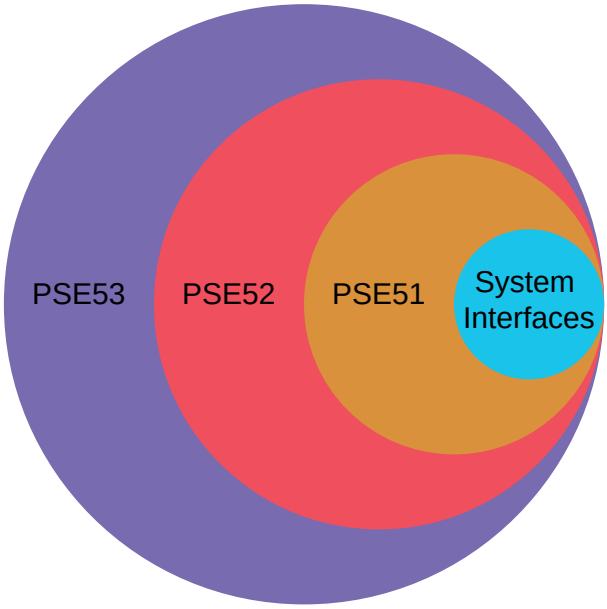


Fig. 14: POSIX Application Environment Profiles (AEP)

- Minimal Realtime System Profile (*PSE51*)
- Realtime Controller System Profile (*PSE52*)
- Dedicated Realtime System Profile (*PSE53*)
- Multi-Purpose Realtime System (PSE54)

POSIX.13-2003 AEP were formalized in 2003 via "Units of Functionality" but the specification is now inactive (for reference only). Nevertheless, the intent is still captured as part of POSIX-1.2017 via *Options* and *Option Groups*.

For more information, please see IEEE 1003.1-2017, Section E, Subprofiling Considerations.

POSIX Applications in Zephyr A POSIX app in Zephyr is *built like any other app* and therefore requires the usual `prj.conf`, `CMakeLists.txt`, and source code. For example, the app below leverages the `nanosleep()` and `perror()` POSIX functions.

Listing 1: *prj.conf* for a simple POSIX app in Zephyr

```
CONFIG_POSIX_API=y
```

Listing 2: A simple app that uses Zephyr's POSIX API

```
#include <stddef.h>
#include <stdio.h>
#include <time.h>

void megasleep(size_t megaseconds)
```

(continues on next page)

(continued from previous page)

```
{
    struct timespec ts = {
        .tv_sec = megaseconds * 1000000,
        .tv_nsec = 0,
    };

    printf("See you in a while!\n");
    if (nanosleep(&ts, NULL) == -1) {
        perror("nanosleep");
    }
}

int main()
{
    megasleep(42);
    return 0;
}
```

For more examples of POSIX applications, please see the POSIX sample applications.

Configuration Like most features in Zephyr, POSIX features are *highly configurable* but disabled by default. Users must explicitly choose to enable POSIX options via *Kconfig* selection.

Subprofiles Enable one of the Kconfig options below to quickly configure a pre-defined *POSIX subprofile*.

- CONFIG_POSIX_AEP_CHOICE_BASE (*Base*)
- CONFIG_POSIX_AEP_CHOICE_PSE51 (*PSE51*)
- CONFIG_POSIX_AEP_CHOICE_PSE52 (*PSE52*)
- CONFIG_POSIX_AEP_CHOICE_PSE53 (*PSE53*)

Additional POSIX *Options and Option Groups* may be enabled as needed via Kconfig (e.g. CONFIG_POSIX_C_LIB_EXT=y). Further fine-tuning may be accomplished via *additional POSIX-related Kconfig options*.

Subprofiles, Options, and Option Groups should be considered the preferred way to configure POSIX in Zephyr going forward.

Legacy Historically, Zephyr used CONFIG_POSIX_API to configure a set of POSIX features that was overloaded and always increasing in size.

- CONFIG_POSIX_API

The option is now frozen, and can be considered equivalent to the following:

- CONFIG_POSIX_AEP_CHOICE_PSE51
- CONFIG_POSIX_FD_MGMT
- CONFIG_POSIX_MESSAGE_PASSING
- CONFIG_POSIX_NETWORKING

However, CONFIG_POSIX_API should be considered legacy and should not be used for new Zephyr applications.

POSIX Conformance

As per *IEEE 1003.1-2017*, this section details Zephyr's POSIX conformance.

POSIX System Interfaces

Table 5: POSIX System Interfaces

Symbol	Support	Remarks
_POSIX_CHOWN_RESTRICTED	0	
_POSIX_NO_TRUNC	0	
_POSIX_VDISABLE	'\0'	

Table 6: POSIX System Interfaces

Symbol	Support	Remarks
_POSIX_VERSION	200809L	
_POSIX_ASYNCHRONOUS_IO	200809L	CONFIG_POSIX_ASYNCHRONOUS_IO†
_POSIX_BARRIERS	200809L	CONFIG_POSIX_BARRIERS
_POSIX_CLOCK_SELECTION	200809L	CONFIG_POSIX_CLOCK_SELECTION
_POSIX_MAPPED_FILES	200809L	CONFIG_POSIX_MAPPED_FILES
_POSIX_MEMORY_PROTECTION	200809L	CONFIG_POSIX_MEMORY_PROTECTION †
_POSIX_READER_WRITER_LOCKS	200809L	CONFIG_POSIX_READER_WRITER_LOCKS
_POSIX_REALTIME_SIGNALS	-1	CONFIG_POSIX_REALTIME_SIGNALS
_POSIX_SEMAPHORES	200809L	CONFIG_POSIX_SEMAPHORES
_POSIX_SPIN_LOCKS	200809L	CONFIG_POSIX_SPIN_LOCKS
_POSIX_THREAD_SAFE_FUNCTIONS	-1	CONFIG_POSIX_THREAD_SAFE_FUNCTIONS
_POSIX_THREADS	-1	CONFIG_POSIX_THREADS
_POSIX_TIMEOUTS	200809L	CONFIG_POSIX_TIMEOUTS
_POSIX_TIMERS	200809L	CONFIG_POSIX_TIMERS
_POSIX2_C_BIND	200809L	

Table 7: POSIX System Interfaces (Unsupported)

Symbol	Support	Remarks
_POSIX_JOB_CONTROL	-1	†
_POSIX_REGEXP	-1	†
_POSIX_SAVED_IDS	-1	†
_POSIX_SHELL	-1	†

Table 8: POSIX System Interfaces (Optional)

Symbol	Support	Remarks
_POSIX_ADVISORY_INFO	-1	
_POSIX_CPUTIME	200809L	CONFIG_POSIX_CPUTIME
_POSIX_FSYNC	200809L	CONFIG_POSIX_FSYNC
_POSIX_IPV6	200809L	CONFIG_POSIX_IPV6
_POSIX_MEMLOCK	200809L	CONFIG_POSIX_MEMLOCK †
_POSIX_MEMLOCK_RANGE	200809L	CONFIG_POSIX_MEMLOCK_RANGE

continues on next page

Table 8 – continued from previous page

Symbol	Support	Remarks
_POSIX_MESSAGE_PASSING	200809	CONFIG_POSIX_MESSAGE_PASSING
_POSIX_MONOTONIC_CLOCK	200809	CONFIG_POSIX_MONOTONIC_CLOCK
_POSIX_PRIORITIZED_IO	-1	
_POSIX_PRIORITY_SCHEDULING	200809	CONFIG_POSIX_PRIORITY_SCHEDULING
_POSIX_RAW_SOCKETS	200809	CONFIG_POSIX_RAW_SOCKETS
_POSIX_SHARED_MEMORY_OBJECTS	200809	CONFIG_POSIX_SHARED_MEMORY_OBJECTS
_POSIX_SPAWN	-1	†
_POSIX_SPORADIC_SERVER	-1	†
_POSIX_SYNCHRONIZED_IO	-1	CONFIG_POSIX_SYNCHRONIZED_IO
_POSIX_THREAD_ATTR_STACKADDR	200809	CONFIG_POSIX_THREAD_ATTR_STACKADDR
_POSIX_THREAD_ATTR_STACKSIZE	200809	CONFIG_POSIX_THREAD_ATTR_STACKSIZE
_POSIX_THREAD_CPUTIME	200809	CONFIG_POSIX_CPUTIME
_POSIX_THREAD_PRIO_INHERIT	200809	CONFIG_POSIX_THREAD_PRIO_INHERIT
_POSIX_THREAD_PRIO_PROTECT	-1	CONFIG_POSIX_THREAD_PRIO_PROTECT
_POSIX_THREAD_PRIORITY_SCHEDULING	200809	CONFIG_POSIX_THREAD_PRIORITY_SCHEDULING
_POSIX_THREAD_PROCESS_SHARED	-1	
_POSIX_THREAD_SPORADIC_SERVER	-1	
_POSIX_TRACE	-1	
_POSIX_TRACE_EVENT_FILTER	-1	
_POSIX_TRACE_INHERIT	-1	
_POSIX_TRACE_LOG	-1	
_POSIX_TYPED_MEMORY_OBJECTS	-1	
_XOPEN_CRYPT	-1	
_XOPEN_REALTIME	-1	
_XOPEN_REALTIME_THREADS	-1	
_XOPEN_STREAMS	200809	CONFIG_XOPEN_STREAMS
_XOPEN_UNIX	-1	

POSIX Shell and Utilities Zephyr does not support a POSIX shell or utilities at this time.

Table 9: POSIX Shell and Utilities

Symbol	Support	Remarks
_POSIX2_C_DEV	-1	†
_POSIX2_CHAR_TERM	-1	†
_POSIX2_FORT_DEV	-1	†
_POSIX2_FORT_RUN	-1	†
_POSIX2_LOCALEDEF	-1	†
_POSIX2_PBS	-1	†
_POSIX2_PBS_ACCOUNTING	-1	†
_POSIX2_PBS_LOCATE	-1	†
_POSIX2_PBS_MESSAGE	-1	†
_POSIX2_PBS_TRACK	-1	†
_POSIX2_SW_DEV	-1	†
_POSIX2_UPE	-1	†
_POSIX2_UNIX	-1	†
_POSIX2_UUCP	-1	†

XSI Conformance

X/Open System Interfaces

Table 10: X/Open System Interfaces

Symbol	Support	Remarks
_POSIX_FSYNC	200809	CONFIG_POSIX_FSYNC
_POSIX_THREAD_ATTR_STACKADDR	200809	CONFIG_POSIX_THREAD_ATTR_STACKADDR
_POSIX_THREAD_ATTR_STACKSIZE	200809	CONFIG_POSIX_THREAD_ATTR_STACKSIZE
_POSIX_THREAD_PROCESS_SHARED	-1	

Note

Some features may exhibit undefined behaviour as they fall beyond the scope of Zephyr's current design and capabilities. For example, multi-processing, ad-hoc memory-mapping, multiple users, or regular expressions are features that are uncommon in low-footprint embedded systems. Such undefined behaviour is denoted with the † (obelus) symbol. Additional details *here*.

Note

Features listed in various POSIX Options or Option Groups may be provided in whole or in part by a conformant C library implementation. This includes (but is not limited to) POSIX Extensions to the ISO C Standard (CX).

POSIX Application Environment Profiles (AEP)

Although inactive, IEEE 1003.13-2003 defined a number of AEP that inspired the modern subprofiling options of IEEE 1003.1-2017. The single-purpose realtime system profiles are listed below, for reference, in terms that agree with the current POSIX-1 standard. PSE54 is not considered at this time.

System Interfaces The required POSIX *System Interfaces* are supported for each Application Environment Profile.

Minimal Realtime System Profile (PSE51) The *Minimal Realtime System Profile* (PSE51) includes all of the *System Interfaces* along with several additional features.

Table 11: PSE51 System Interfaces

Symbol	Support	Remarks
_POSIX_AEP_REALTIME_MINIMAL	-1	CONFIG_POSIX_AEP_REALTIME_MINIMAL

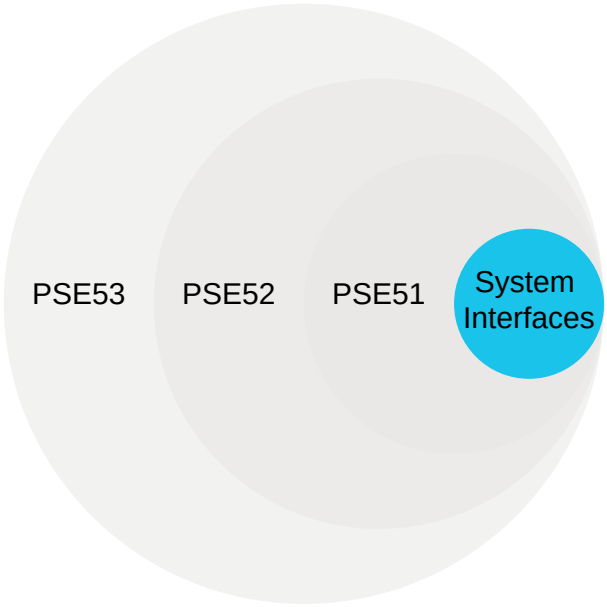


Fig. 15: System Interfaces

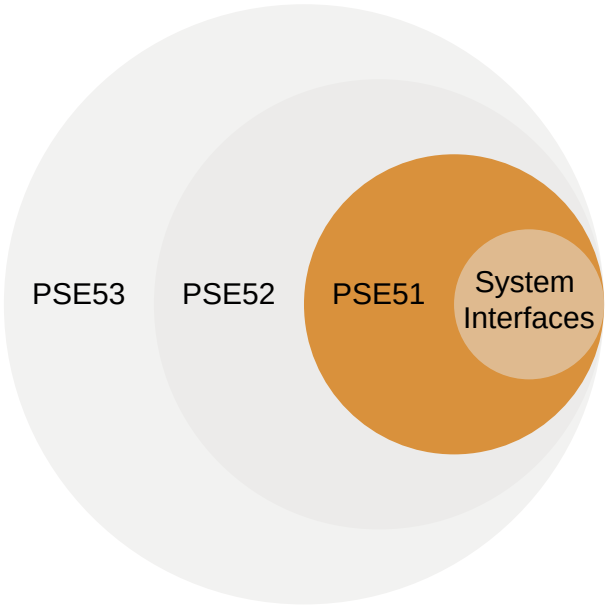


Fig. 16: Minimal Realtime System Profile (PSE51)

Table 12: PSE51 Option Groups

Symbol	Support	Remarks
POSIX_C_LANG_JUMP	yes	
POSIX_C_LANG_SUPPORT	yes	
POSIX_DEVICE_IO		CONFIG_POSIX_DEVICE_IO
POSIX_SIGNALS		CONFIG_POSIX_SIGNALS
POSIX_SINGLE_PROCESS	yes	CONFIG_POSIX_SINGLE_PROCESS
XSI_THREADS_EXT	yes	CONFIG_XSI_THREADS_EXT

Table 13: PSE51 Option Requirements

Symbol	Support	Remarks
_POSIX_FSYNC	200809	CONFIG_POSIX_FSYNC
_POSIX_MEMLOCK	200809	CONFIG_POSIX_MEMLOCK †
_POSIX_MEMLOCK_RANGE	200809	CONFIG_POSIX_MEMLOCK_RANGE
_POSIX_MONOTONIC_CLOCK	200809	CONFIG_POSIX_MONOTONIC_CLOCK
_POSIX_SHARED_MEMORY_OBJECTS	200809	CONFIG_POSIX_SHARED_MEMORY_OBJECTS
_POSIX_SYNCHRONIZED_IO	-1	CONFIG_POSIX_SYNCHRONIZED_IO
_POSIX_THREAD_ATTR_STACKADDR	200809	CONFIG_POSIX_THREAD_ATTR_STACKADDR
_POSIX_THREAD_ATTR_STACKSIZE	200809	CONFIG_POSIX_THREAD_ATTR_STACKSIZE
_POSIX_THREAD_CPUTIME	200809	CONFIG_POSIX_CPUTIME
_POSIX_THREAD_PRIO_INHERIT	200809	CONFIG_POSIX_THREAD_PRIO_INHERIT
_POSIX_THREAD_PRIO_PROTECT	-1	CONFIG_POSIX_THREAD_PRIO_PROTECT
_POSIX_THREAD_PRIORITY_SCHEDULING	200809	CONFIG_POSIX_THREAD_PRIORITY_SCHEDULING
_POSIX_THREAD_SPORADIC_SERVER	-1	

Realtime Controller System Profile (PSE52) The *Realtime Controller System Profile* (PSE52) includes all features from PSE51 and the *System Interfaces*.

Table 14: PSE52 System Interfaces

Symbol	Support	Remarks
_POSIX_AEP_REALTIME_CONTROLLER	-1	CONFIG_POSIX_AEP_REALTIME_CONTROLLER

Table 15: PSE52 Option Groups

Symbol	Support	Remarks
POSIX_C_LANG_MATH	yes	
POSIX_FD_MGMT		CONFIG_POSIX_FD_MGMT
POSIX_FILE_SYSTEM		CONFIG_POSIX_FILE_SYSTEM

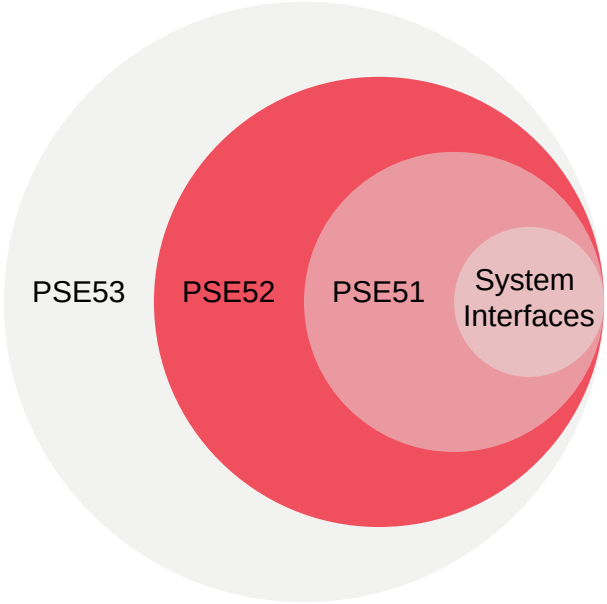


Fig. 17: Realtime Controller System Profile (PSE52)

Table 16: PSE52 Option Requirements

Symbol	Support	Remarks
_POSIX_MESSAGE_PASSING	200809	CONFIG_POSIX_MESSAGE_PASSING
_POSIX_TRACE	-1	
_POSIX_TRACE_EVENT_FILTER	-1	
_POSIX_TRACE_LOG	-1	

Dedicated Realtime System Profile (PSE53) The *Dedicated Realtime System Profile* (PSE53) includes all features from PSE52, PSE51, and the *System Interfaces*.

Table 17: PSE53 System Interfaces

Symbol	Support	Remarks
_POSIX_AEP_REALTIME_DEDICATED	-1	CONFIG_POSIX_AEP_REALTIME_DEDICATED

Table 18: PSE53 Option Groups

Symbol	Support	Remarks
POSIX_MULTI_PROCESS		CONFIG_POSIX_MULTI_PROCESS†
POSIX_NETWORKING	yes	CONFIG_POSIX_NETWORKING
POSIX_PIPE		
POSIX_SIGNAL_JUMP		

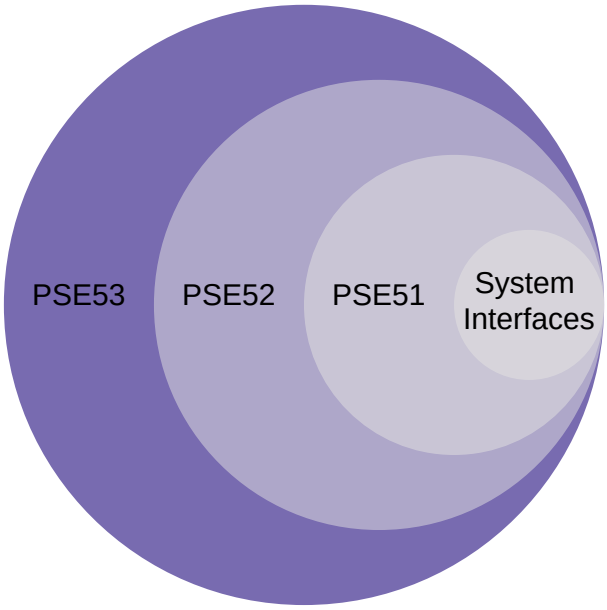


Fig. 18: Dedicated Realtime System Profile (PSE53)

Table 19: PSE53 Option Requirements

Symbol	Support	Remarks
_POSIX_CPUTIME	200809	CONFIG_POSIX_CPUTIME
_POSIX_PRIORITIZED_IO	-1	
_POSIX_PRIORITY_SCHEDULING	-1	
_POSIX_RAW_SOCKETS	200809	CONFIG_POSIX_RAW_SOCKETS
_POSIX_SPAWN	-1	†
_POSIX_SPORADIC_SERVER	-1	†

Implementation Details

In many ways, Zephyr provides support like any POSIX OS; API bindings are provided in the C programming language, POSIX headers are available in the standard include path, when configured.

Unlike other multi-purpose POSIX operating systems

- Zephyr is not "a POSIX OS". The Zephyr kernel was not designed around the POSIX standard, and POSIX support is an opt-in feature
- Zephyr apps are not linked separately, nor do they execute as subprocesses
- Zephyr, libraries, and application code are compiled and linked together, running similarly to a single-process application, in a single (possibly virtual) address space
- Zephyr does not provide a POSIX shell, compiler, utilities, and is not self-hosting.

Note

Unlike the Linux kernel or FreeBSD, Zephyr does not maintain a static table of system call

numbers for each supported architecture, but instead generates system calls dynamically at build time. See *System Calls* for more information.

Design As a library, Zephyr's POSIX API implementation makes an effort to be a thin abstraction layer between the application, middleware, and the Zephyr kernel.

Some general design considerations:

- The POSIX interface and implementations should be part of Zephyr's POSIX library, and not elsewhere, unless required both by the POSIX API implementation and some other feature. An example where the implementation should remain part of the POSIX implementation is `getopt()`. Examples where the implementation should be part of separate libraries are multithreading and networking.
- When the POSIX API and another Zephyr subsystem both rely on a feature, the implementation of that feature should be as a separate Zephyr library that can be used by both the POSIX API and the other library or subsystem. This reduces the likelihood of dependency cycles in code. When practical, that rule should expand to include macros. In the example below, `libposix` depends on `libzfoo` for the implementation of some functionality "foo" in Zephyr. If `libzfoo` also depends on `libposix`, then there is a dependency cycle. The cycle can be removed via mutual dependency, `libcommon`.

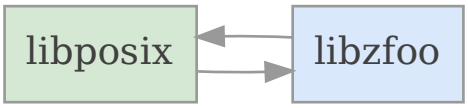


Fig. 19: Dependency cycle between POSIX and another Zephyr library

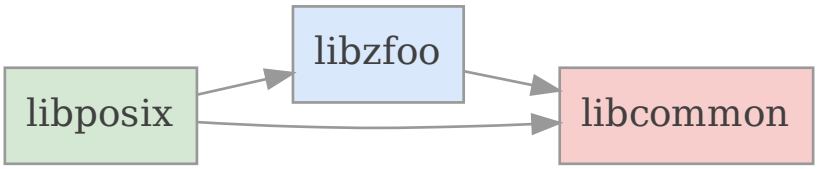


Fig. 20: Mutual dependencies between POSIX and other Zephyr libraries

- POSIX API calls should be provided as regular callable C functions; if a Zephyr *System Call* is needed as part of the implementation, the declaration and the implementation of that system call should be hidden behind the POSIX API.

POSIX Option and Option Group Details

POSIX Option Groups

POSIX_BARRIERS Enable this option group with `CONFIG_POSIX_BARRIERS`.

Table 20: POSIX_BARRIERS

API	Supported
pthread_barrier_destroy()	yes
pthread_barrier_init()	yes
pthread_barrier_wait()	yes
pthread_barrierattr_destroy()	yes
pthread_barrierattr_init()	yes

POSIX_C_LANG_JUMP The `POSIX_C_LANG_JUMP` Option Group is included in the ISO C standard.

> **Note**
>
> When using Newlib, Picolibc, or other C libraries conforming to the ISO C Standard, the `POSIX_C_LANG_JUMP` Option Group is considered supported.

Table 21: POSIX_C_LANG_JUMP

API	Supported
setjmp()	yes
longjmp()	yes

POSIX_C_LANG_MATH The `POSIX_C_LANG_MATH` Option Group is included in the ISO C standard.

> **Note**
>
> When using Newlib, Picolibc, or other C libraries conforming to the ISO C Standard, the `POSIX_C_LANG_MATH` Option Group is considered supported.

Please refer to Subprofiling Considerations for details on the `POSIX_C_LANG_MATH` Option Group.

POSIX_C_LANG_SUPPORT The POSIX_C_LANG_SUPPORT option group contains the general ISO C Library.

> **Note**
>
> When using Newlib, Picolibc, or other C libraries conforming to the ISO C Standard, the entire `POSIX_C_LANG_SUPPORT` Option Group is considered supported.

Please refer to Subprofiling Considerations for details on the `POSIX_C_LANG_SUPPORT` Option Group.

For more information on developing Zephyr applications in the C programming language, please refer to *details*.

POSIX_C_LIB_EXT Enable this option group with `CONFIG_POSIX_C_LIB_EXT`.

Table 22: POSIX_C_LIB_EXT

API	Supported
fnmatch()	yes
getopt()	yes
getsubopt()	
optarg	yes
opterr	yes
optind	yes
optopt	yes
stpcpy()	
stpncpy()	
strcasecmp()	
strdup()	
strfmon()	
strncasecmp()	yes
strndup()	
strnlen()	yes

POSIX_CLOCK_SELECTION Enable this option group with `CONFIG_POSIX_CLOCK_SELECTION`.

Table 23: POSIX_CLOCK_SELECTION

API	Supported
pthread_condattr_getclock()	yes
pthread_condattr_setclock()	yes
clock_nanosleep()	yes

POSIX_DEVICE_IO Enable this option group with `CONFIG_POSIX_DEVICE_IO`.

Table 24: POSIX_DEVICE_IO

API	Supported
FD_CLR()	yes
FD_ISSET()	yes
FD_SET()	yes
FD_ZERO()	yes
clearerr()	yes
close()	yes
fclose()	
fdopen()	
feof()	
ferror()	
fflush()	
fgetc()	
fgets()	
fileno()	
fopen()	
fprintf()	yes
fputc()	yes
fputs()	yes
fread()	

continues on next page

Table 24 – continued from previous page

API	Supported
freopen()	
fscanf()	
fwrite()	yes
getc()	
getchar()	
gets()	
open()	yes
perror()	yes
poll()	yes
printf()	yes
pread()	
pselect()	
putc()	yes
putchar()	yes
puts()	yes
pwrite()	
read()	yes
scanf()	
select()	yes
setbuf()	
setvbuf()	
stderr	
stdin	
stdout	
ungetc()	
vfprintf()	yes
vfscanf()	
vprintf()	yes
vscanf()	
write()	yes

POSIX_FD_MGMT Enable this option group with `CONFIG_POSIX_FD_MGMT`.

Table 25: POSIX_FD_MGMT

API	Supported
dup()	
dup2()	
fcntl()	
fgetpos()	
fseek()	
fseeko()	
fsetpos()	
ftell()	
ftello()	
ftruncate()	yes
lseek()	
rewind()	

POSIX_FILE_LOCKING

Table 26: POSIX_FILE_LOCKING

API	Supported
flockfile()	
ftrylockfile()	
funlockfile()	
getc_unlocked()	
getchar_unlocked()	
putc_unlocked()	
putchar_unlocked()	

POSIX_FILE_SYSTEM Enable this option group with `CONFIG_POSIX_FILE_SYSTEM`.

Table 27: POSIX_FILE_SYSTEM

API	Supported
access()	
chdir()	
closedir()	yes
creat()	
fchdir()	
fpathconf()	
fstat()	yes
fstatvfs()	
getcwd()	
link()	
mkdir()	yes
mkstemp()	
opendir()	yes
pathconf()	
readdir()	yes
remove()	
rename()	yes
rewinddir()	
rmdir()	
stat()	yes
statvfs()	
tmpfile()	
tmpnam()	
truncate()	
unlink()	yes
utime()	

POSIX_MAPPED_FILES Enable this option group with `CONFIG_POSIX_MAPPED_FILES`.

Table 28: POSIX_MAPPED_FILES

API	Supported
mmap()	yes
msync()	yes
munmap()	yes

POSIX_MEMORY_PROTECTION Enable this option group with `CONFIG_POSIX_MEMORY_PROTECTION`.

Table 29: POSIX_MEMORY_PROTECTION

API	Supported
mprotect()	yes †

POSIX_MULTI_PROCESS Enable this option group with `CONFIG_POSIX_MULTI_PROCESS`.

Table 30: POSIX_MULTI_PROCESS

API	Supported
_Exit()	yes
_exit()	yes
assert()	yes
atexit()	†
clock()	
execl()	†
execle()	†
execlp()	†
execv()	†
execve()	†
execvp()	†
exit()	yes
fork()	†
getpgrp()	†
getpgid()	†
getpid()	yes †
getppid()	†
getsid()	†
setsid()	†
sleep()	yes
times()	
wait()	†
waitid()	†
waitpid()	†

POSIX_NETWORKING The function `sockatmark()` is not yet supported and is expected to fail setting `errno` to `ENOSYS` †.

Enable this option group with `CONFIG_POSIX_NETWORKING`.

Table 31: POSIX_NETWORKING

API	Supported
accept()	yes
bind()	yes
connect()	yes
endhostent()	yes
endnetent()	yes
endprotoent()	yes
endservent()	yes
freeaddrinfo()	yes

continues on next page

Table 31 – continued from previous page

API	Supported
gai_strerror()	yes
getaddrinfo()	yes
gethostent()	yes
gethostname()	yes
getnameinfo()	yes
getnetbyaddr()	yes
getnetbyname()	yes
getnetent()	yes
getpeername()	yes
getprotobyname()	yes
getprotobynumber()	yes
getprotoent()	yes
getservbyname()	yes
getservbyport()	yes
getservent()	yes
getsockname()	yes
getsockopt()	yes
htonl()	yes
htons()	yes
if_freenameindex()	yes
if_indextoname()	yes
if_nameindex()	yes
if_nametoindex()	yes
inet_addr()	yes
inet_ntoa()	yes
inet_ntop()	yes
inet_pton()	yes
listen()	yes
ntohl()	yes
ntohs()	yes
recv()	yes
recvfrom()	yes
recvmsg()	yes
send()	yes
sendmsg()	yes
sendto()	yes
sethostent()	yes
setnetent()	yes
setprotoent()	yes
setservent()	yes
setsockopt()	yes
shutdown()	yes
socket()	yes
sockatmark()	yes †
socketpair()	yes

POSIX_PIPE

Table 32: POSIX_PIPE

API	Supported
pipe()	

POSIX_REALTIME_SIGNALS Enable this option group with `CONFIG_POSIX_REALTIME_SIGNALS`.

Table 33: POSIX_REALTIME_SIGNALS

API	Supported
sigqueue()	
sigtimedwait()	
sigwaitinfo()	

POSIX_SEMAPHORES Enable this option group with `CONFIG_POSIX_SEMAPHORES`.

Table 34: POSIX_SEMAPHORES

API	Supported
sem_close()	yes
sem_destroy()	yes
sem_getvalue()	yes
sem_init()	yes
sem_open()	yes
sem_post()	yes
sem_trywait()	yes
sem_unlink()	yes
sem_wait()	yes

POSIX_SIGNAL_JUMP

Table 35: POSIX_SIGNAL_JUMP

API	Supported
siglongjmp()	
sigsetjmp()	

POSIX_SIGNALS Signal services are a basic mechanism within POSIX-based systems and are required for error and event handling.

Enable this option group with `CONFIG_POSIX_SIGNALS`.

Table 36: POSIX_SIGNALS

API	Supported
abort()	yes
alarm()	
kill()	
pause()	
raise()	
sigaction()	
sigaddset()	yes
sigdelset()	yes
sigemptyset()	yes
sigfillset()	yes
sigismember()	yes
signal()	
sigpending()	
sigprocmask()	yes
sigsuspend()	
sigwait()	
strsignal()	yes

POSIX_SINGLE_PROCESS The POSIX_SINGLE_PROCESS option group contains services for single process applications.

Enable this option group with `CONFIG_POSIX_SINGLE_PROCESS`.

Table 37: POSIX_SINGLE_PROCESS

API	Supported
confstr()	yes
environ	yes
errno	yes
getenv()	yes
setenv()	yes
sysconf()	yes
uname()	yes
unsetenv()	yes

POSIX_SPIN_LOCKS Enable this option group with `CONFIG_POSIX_SPIN_LOCKS`.

Table 38: POSIX_SPIN_LOCKS

API	Supported
pthread_spin_destroy()	yes
pthread_spin_init()	yes
pthread_spin_lock()	yes
pthread_spin_trylock()	yes
pthread_spin_unlock()	yes

POSIX_THREADS_BASE The basic assumption in this profile is that the system consists of a single (implicit) process with multiple threads. Therefore, the standard requires all basic thread services, except those related to multiple processes.

Enable this option group with `CONFIG_POSIX_THREADS`.

Table 39: POSIX_THREADS_BASE

API	Supported
pthread_atfork()	yes
pthread_attr_destroy()	yes
pthread_attr_getdetachstate()	yes
pthread_attr_getschedparam()	yes
pthread_attr_init()	yes
pthread_attr_setdetachstate()	yes
pthread_attr_setschedparam()	yes
pthread_barrier_destroy()	yes
pthread_barrier_init()	yes
pthread_barrier_wait()	yes
pthread_barrierattr_destroy()	yes
pthread_barrierattr_getpshared()	yes
pthread_barrierattr_init()	yes
pthread_barrierattr_setpshared()	yes
pthread_cancel()	yes
pthread_cleanup_pop()	yes
pthread_cleanup_push()	yes
pthread_cond_broadcast()	yes
pthread_cond_destroy()	yes
pthread_cond_init()	yes
pthread_cond_signal()	yes
pthread_cond_timedwait()	yes
pthread_cond_wait()	yes
pthread_condattr_destroy()	yes
pthread_condattr_init()	yes
pthread_create()	yes
pthread_detach()	yes
pthread_equal()	yes
pthread_exit()	yes
pthread_getspecific()	yes
pthread_join()	yes
pthread_key_create()	yes
pthread_key_delete()	yes
pthread_kill()	
pthread_mutex_destroy()	yes
pthread_mutex_init()	yes
pthread_mutex_lock()	yes
pthread_mutex_trylock()	yes
pthread_mutex_unlock()	yes
pthread_mutexattr_destroy()	yes
pthread_mutexattr_init()	yes
pthread_once()	yes
pthread_self()	yes
pthread_setcancelstate()	yes
pthread_setcanceltype()	yes
pthread_setspecific()	yes
pthread_sigmask()	yes
pthread_testcancel()	yes

POSIX_THREADS_EXT Enable this option group with `CONFIG_POSIX_THREADS_EXT`.

Table 40: POSIX_THREADS_EXT

API	Supported
pthread_attr_getguardsize()	yes
pthread_attr_setguardsize()	yes
pthread_mutexattr_gettype()	yes
pthread_mutexattr_settype()	yes

POSIX_TIMERS Enable this option group with `CONFIG_POSIX_TIMERS`.

Table 41: POSIX_TIMERS

API	Supported
clock_getres()	yes
clock_gettime()	yes
clock_settime()	yes
nanosleep()	yes
timer_create()	yes
timer_delete()	yes
timer_gettime()	yes
timer_getoverrun()	yes
timer_settime()	yes

XSI_SYSTEM_LOGGING Enable this option group with `CONFIG_XSI_SYSTEM_LOGGING`.

Table 42: XSI_SYSTEM_LOGGING

API	Supported
closelog()	yes
openlog()	yes
setlogmask()	yes
syslog()	yes

XSI_THREADS_EXT The XSI_THREADS_EXT option group is required because it provides functions to control a thread's stack. This is considered useful for any real-time application.

Enable this option group with `CONFIG_XSI_THREADS_EXT`.

Table 43: XSI_THREADS_EXT

API	Supported
pthread_attr_getstack()	yes
pthread_attr_setstack()	yes
pthread_getconcurrency()	yes
pthread_setconcurrency()	yes

POSIX Options

_POSIX_ASYNCHRONOUS_IO Functions part of the `_POSIX_ASYNCHRONOUS_IO` Option are not implemented in Zephyr but are provided so that conformant applications can still link. These functions will fail, setting `errno` to `ENOSYS`†.

Enable this option with CONFIG_POSIX_ASYNCHRONOUS_IO.

Table 44: _POSIX_ASYNCHRONOUS_IO

API	Supported
aio_cancel()	yes †
aio_error()	yes †
aio_fsync()	yes †
aio_read()	yes †
aio_return()	yes †
aio_suspend()	yes †
aio_write()	yes †
lio_listio()	yes †

_POSIX_CPUTIME Enable this option with CONFIG_POSIX_CPUTIME.

Table 45: _POSIX_CPUTIME

API	Supported
CLOCK_PROCESS_CPUTIME_ID	yes

_POSIX_FSYNC Enable this option with CONFIG_POSIX_FSYNC.

Table 46: _POSIX_FSYNC

API	Supported
fsync()	yes

_POSIX_IPV6 Internet Protocol Version 6 is supported.

For more information, please refer to *Networking*.

Enable this option with CONFIG_POSIX_IPV6.

_POSIX_MEMLOCK Zephyr's *Demand Paging API* does not yet support pinning or unpinning all virtual memory regions. The functions below are expected to fail and set errno to ENOSYS †.

Enable this option with CONFIG_POSIX_MEMLOCK.

Table 47: _POSIX_MEMLOCK

API	Supported
mlockall()	yes
munlockall()	yes

_POSIX_MEMLOCK_RANGE Enable this option with CONFIG_POSIX_MEMLOCK_RANGE.

Table 48: _POSIX_MEMLOCK_RANGE

API	Supported
mlock()	yes
munlock()	yes

_POSIX_MESSAGE_PASSING Enable this option with `CONFIG_POSIX_MESSAGE_PASSING`.

Table 49: _POSIX_MESSAGE_PASSING

API	Supported
mq_close()	yes
mq_getattr()	yes
mq_notify()	yes
mq_open()	yes
mq_receive()	yes
mq_send()	yes
mq_setattr()	yes
mq_unlink()	yes

_POSIX_MONOTONIC_CLOCK Enable this option with `CONFIG_POSIX_MONOTONIC_CLOCK`.

Table 50: _POSIX_MONOTONIC_CLOCK

API	Supported
CLOCK_MONOTONIC	yes

_POSIX_PRIORITY_SCHEDULING As processes are not yet supported in Zephyr, the functions `sched_rr_get_interval()`, `sched_setparam()`, and `sched_setscheduler()` are expected to fail setting `errno` to `ENOSYS`*†*.

Enable this option with `CONFIG_POSIX_PRIORITY_SCHEDULING`.

Table 51: _POSIX_PRIORITY_SCHEDULING

API	Supported
sched_get_priority_max()	yes
sched_get_priority_min()	yes
sched_getparam()	yes
sched_getscheduler()	yes
sched_rr_get_interval()	yes *†*
sched_setparam()	yes *†*
sched_setscheduler()	yes *†*
sched_yield()	yes

_POSIX_RAW_SOCKETS Raw sockets are supported.

For more information, please refer to `CONFIG_NET_SOCKETS_PACKET`.

Enable this option with `CONFIG_POSIX_RAW_SOCKETS`.

_POSIX_READER_WRITER_LOCKS Enable this option with `CONFIG_POSIX_READER_WRITER_LOCKS`.

Table 52: _POSIX_READER_WRITER_LOCKS

API	Supported
pthread_rwlock_destroy()	yes
pthread_rwlock_init()	yes
pthread_rwlock_rdlock()	yes
pthread_rwlock_tryrdlock()	yes
pthread_rwlock_trywrlock()	yes
pthread_rwlock_unlock()	yes
pthread_rwlock_wrlock()	yes
pthread_rwlockattr_destroy()	yes
pthread_rwlockattr_getpshared()	yes
pthread_rwlockattr_init()	yes
pthread_rwlockattr_setpshared()	yes

_POSIX_SHARED_MEMORY_OBJECTS Enable this option with `CONFIG_POSIX_SHARED_MEMORY_OBJECTS`.

Table 53: _POSIX_SHARED_MEMORY_OBJECTS

API	Supported
mmap()	yes
munmap()	yes
shm_open()	yes
shm_unlink()	yes

_POSIX_SYNCHRONIZED_IO Enable this option with `CONFIG_POSIX_SYNCHRONIZED_IO`.

Table 54: _POSIX_SYNCHRONIZED_IO

API	Supported
fdatasync()	
fsync()	yes
msync()	

_POSIX_THREAD_ATTR_STACKADDR Enable this option with `CONFIG_POSIX_THREAD_ATTR_STACKADDR`.

Table 55: _POSIX_THREAD_ATTR_STACKADDR

API	Supported
pthread_attr_getstackaddr()	yes
pthread_attr_setstackaddr()	yes

_POSIX_THREAD_ATTR_STACKSIZE Enable this option with `CONFIG_POSIX_THREAD_ATTR_STACKSIZE`.

Table 56: _POSIX_THREAD_ATTR_STACKSIZE

API	Supported
pthread_attr_getstacksize()	yes
pthread_attr_setstacksize()	yes

_POSIX_THREAD_CPUTIME Enable this option with `CONFIG_POSIX_THREAD_CPUTIME`.

Table 57: _POSIX_THREAD_CPUTIME

API	Supported
CLOCK_THREAD_CPUTIME_ID	yes
pthread_getcpuclockid()	yes

_POSIX_THREAD_PRIO_INHERIT Enable this option with `CONFIG_POSIX_THREAD_PRIO_INHERIT`.

Table 58: _POSIX_THREAD_PRIO_INHERIT

API	Supported
pthread_mutexattr_getprotocol()	yes
pthread_mutexattr_setprotocol()	yes

_POSIX_THREAD_PRIO_PROTECT Enable this option with `CONFIG_POSIX_THREAD_PRIO_PROTECT`.

Table 59: _POSIX_THREAD_PRIO_PROTECT

API	Supported
pthread_mutex_getprioceiling()	
pthread_mutex_setprioceiling()	
pthread_mutexattr_getprioceiling()	
pthread_mutexattr_getprotocol()	yes
pthread_mutexattr_setprioceiling()	
pthread_mutexattr_setprotocol()	yes

_POSIX_THREAD_PRIORITY_SCHEDULING Enable this option with `CONFIG_POSIX_THREAD_PRIORITY_SCHEDULING`.

Table 60: _POSIX_THREAD_PRIORITY_SCHEDULING

API	Supported
pthread_attr_getinheritsched()	yes
pthread_attr_getschedpolicy()	yes
pthread_attr_getscope()	yes
pthread_attr_setinheritsched()	yes
pthread_attr_setschedpolicy()	yes
pthread_attr_setscope()	yes
pthread_getschedparam()	yes
pthread_setschedparam()	yes
pthread_setschedprio()	yes

_POSIX_THREAD_SAFE_FUNCTIONS Enable this option with CONFIG_POSIX_THREAD_SAFE_FUNCTIONS.

Table 61: _POSIX_THREAD_SAFE_FUNCTIONS

API	Supported
asctime_r()	
ctime_r()	
flockfile()	
ftrylockfile()	
funlockfile()	
getc_unlocked()	
getchar_unlocked()	
getgrgid_r()	
getgrnam_r()	
getpwnam_r()	
getpwuid_r()	
gmtime_r()	yes
localtime_r()	
putc_unlocked()	
putchar_unlocked()	
rand_r()	yes
readdir_r()	
strerror_r()	yes
strtok_r()	yes

_POSIX_TIMEOUTS Enable this option with CONFIG_POSIX_TIMEOUTS.

Table 62: _POSIX_TIMEOUTS

API	Supported
mq_timedreceive()	yes
mq_timedsend()	yes
pthread_mutex_timedlock()	yes
pthread_rwlock_timedrdlock()	yes
pthread_rwlock_timedwrlock()	yes
sem_timedwait()	yes
posix_trace_timedgetnext_event()	

_XOPEN_STREAMS With the exception of ioctl(), functions in the _XOPEN_STREAMS option group are not implemented in Zephyr but are provided so that conformant applications can still link. Unimplemented functions in this option group will fail, setting errno to ENOSYS †.

Enable this option with CONFIG_XOPEN_STREAMS.

Table 63: _XOPEN_STREAMS

API	Supported
fattach()	yes †
fdetach()	yes †
getmsg()	yes †
getpmsg()	yes †
ioctl()	yes
isastream()	yes †
putmsg()	yes †
putpmsg()	yes †

Additional Configuration Options

Below is a non-exhaustive list of additional *Configuration System (Kconfig)* options relating to Zephyr's implementation of the POSIX API.

- CONFIG_DYNAMIC_THREAD
- CONFIG_DYNAMIC_THREAD_POOL_SIZE
- CONFIG_EVENTFD
- CONFIG_FDTABLE
- CONFIG_GETOPT_LONG
- CONFIG_MAX_PTHREAD_SPINLOCK_COUNT
- CONFIG_MQUEUE_NAMELEN_MAX
- CONFIG_POSIX_MQ_OPEN_MAX
- CONFIG_MSG_SIZE_MAX
- CONFIG_NET_SOCKETPAIR
- CONFIG_NET_SOCKETS
- CONFIG_NET_SOCKETS_POLL_MAX
- CONFIG_ZVFS_OPEN_MAX
- CONFIG_POSIX_API
- CONFIG_POSIX_OPEN_MAX
- CONFIG_POSIX_PTHREAD_ATTR_GUARDSIZE_BITS
- CONFIG_POSIX_PTHREAD_ATTR_GUARDSIZE_DEFAULT
- CONFIG_POSIX_PTHREAD_ATTR_STACKSIZE_BITS
- CONFIG_POSIX_RTSIG_MAX
- CONFIG_POSIX_SIGNAL_STRING_DESC
- CONFIG_POSIX_THREAD_KEYS_MAX
- CONFIG_POSIX_THREAD_THREADS_MAX
- CONFIG_POSIX_UNAME_NODENAME_LEN
- CONFIG_POSIX_UNAME_VERSION_LEN
- CONFIG_PTHREAD_CREATE_BARRIER
- CONFIG_PTHREAD_RECYCLER_DELAY_MS

- CONFIG_POSIX_SEM_NAMELEN_MAX
- CONFIG_POSIX_SEM_NSEMS_MAX
- CONFIG_POSIX_SEM_VALUE_MAX
- CONFIG_TIMER_CREATE_WAIT
- CONFIG_THREAD_STACK_INFO
- CONFIG_ZVFS_EVENTFD_MAX

4.20.2 CMSIS RTOS v1

Cortex-M Software Interface Standard (CMSIS) RTOS is a vendor-independent hardware abstraction layer for the ARM Cortex-M processor series and defines generic tool interfaces. Though it was originally defined for ARM Cortex-M microcontrollers alone, it could be easily extended to other microcontrollers making it generic. For more information on CMSIS RTOS v1, please refer http://www.keil.com/pack/doc/CMSIS/RTOS/html/index.html

4.20.3 CMSIS RTOS v2

Cortex-M Software Interface Standard (CMSIS) RTOS is a vendor-independent hardware abstraction layer for the ARM Cortex-M processor series and defines generic tool interfaces. Though it was originally defined for ARM Cortex-M microcontrollers alone, it could be easily extended to other microcontrollers making it generic. For more information on CMSIS RTOS v2, please refer to the CMSIS-RTOS2 Documentation.

Features not supported in Zephyr implementation

Kernel
: `osKernelGetState`, `osKernelSuspend`, `osKernelResume`, `osKernelInitialize` and `osKernelStart` are not supported.

Mutex
: `osMutexPrioInherit` is supported by default and is not configurable, you cannot select/unselect this attribute.

  `osMutexRecursive` is also supported by default. If this attribute is not set, an error is thrown when the same thread tries to acquire it the second time.

  `osMutexRobust` is not supported in Zephyr.

Return values not supported in the Zephyr implementation

`osKernelUnlock`, `osKernelLock`, `osKernelRestoreLock`
: `osError` (Unspecified error) is not supported.

`osSemaphoreDelete`
: `osErrorResource` (the semaphore specified by parameter semaphore_id is in an invalid semaphore state) is not supported.

`osMutexDelete`
: `osErrorResource` (mutex specified by parameter mutex_id is in an invalid mutex state) is not supported.

`osTimerDelete`
: `osErrorResource` (the timer specified by parameter timer_id is in an invalid timer state) is not supported.

osMessageQueueReset
: osErrorResource (the message queue specified by parameter msgq_id is in an invalid message queue state) is not supported.

osMessageQueueDelete
: osErrorResource (the message queue specified by parameter msgq_id is in an invalid message queue state) is not supported.

osMemoryPoolFree
: osErrorResource (the memory pool specified by parameter mp_id is in an invalid memory pool state) is not supported.

osMemoryPoolDelete
: osErrorResource (the memory pool specified by parameter mp_id is in an invalid memory pool state) is not supported.

osEventFlagsSet, osEventFlagsClear
: osFlagsErrorUnknown (Unspecified error) and osFlagsErrorResource (Event flags object specified by parameter ef_id is not ready to be used) are not supported.

osEventFlagsDelete
: osErrorParameter (the value of the parameter ef_id is incorrect) is not supported.

osThreadFlagsSet
: osFlagsErrorUnknown (Unspecified error) and osFlagsErrorResource (Thread specified by parameter thread_id is not active to receive flags) are not supported.

osThreadFlagsClear
: osFlagsErrorResource (Running thread is not active to receive flags) is not supported.

osDelayUntil
: osParameter (the time cannot be handled) is not supported.

4.21 Power off

group sys_poweroff

Functions

FUNC_NORETURN void **sys_poweroff**(void)

Perform a system power off.

This function will perform an immediate power off of the system. It is the responsibility of the caller to ensure that the system is in a safe state to be powered off. Any required wake up sources must be enabled before calling this function.

CONFIG_POWEROFF needs to be enabled to use this API.

4.22 Shell

- *Overview*
- *Backends*
  - *Telnet*

  - *USB CDC ACM*
  - *Bluetooth LE (NUS)*
  - *Segget RTT*
- *Commands*
  - *Commonly-used command groups*
  - *Creating commands*
  - *Dictionary commands*
  - *Commands execution*
  - *Built-in commands*
- *Tab Feature*
- *History Feature*
- *Wildcards Feature*
- *Meta Keys Feature*
- *Getopt Feature*
- *Obscured Input Feature*
- *Shell Logger Backend Feature*
- *RTT Backend Channel Selection*
- *Usage*
- *API Reference*

4.22.1 Overview

This module allows you to create and handle a shell with a user-defined command set. You can use it in examples where more than simple button or LED user interaction is required. This module is a Unix-like shell with these features:

- Support for multiple instances.
- Advanced cooperation with the *Logging*.
- Support for static and dynamic commands.
- Support for dictionary commands.
- Smart command completion with the `Tab` key.
- Built-in commands: **`clear`**, **`shell`**, **`colors`**, **`echo`**, **`history`** and **`resize`**.
- Viewing recently executed commands using keys: ↑ ↓ or meta keys.
- Text edition using keys: ←, →, `Backspace`, `Delete`, `End`, `Home`, `Insert`.
- Support for ANSI escape codes: `VT100` and `ESC[n~` for cursor control and color printing.
- Support for editing multiline commands.
- Built-in handler to display help for the commands.
- Support for wildcards: `*` and `?`.
- Support for meta keys.
- Support for getopt and getopt_long.

- Kconfig configuration to optimize memory usage.

Note

Some of these features have a significant impact on RAM and flash usage, but many can be disabled when not needed. To default to options which favor reduced RAM and flash requirements instead of features, you should enable `CONFIG_SHELL_MINIMAL` and selectively enable just the features you want.

4.22.2 Backends

The module can be connected to any transport for command input and output. At this point, the following transport layers are implemented:

- MQTT
- Segger RTT
- SMP
- Telnet
- UART
- USB
- Bluetooth LE (NUS)
- RPMSG
- DUMMY - not a physical transport layer.

Telnet

Enabling `CONFIG_SHELL_BACKEND_TELNET` will allow users to use telnet as a shell backend. Connecting to it can be done using PuTTY or any `telnet` client. For example:

```
telnet <ip address> <port>
```

By default the telnet client won't handle telnet commands and configuration. Although command support can be enabled with `CONFIG_SHELL_TELNET_SUPPORT_COMMAND`. This will give the telnet client access to a very limited set of supported commands but still can be turned on if needed. One of the command options it supports is the `ECHO` option. This will allow the client to be in character mode (character at a time), similar to a UART backend in that regard. This will make the client send a character as soon as it is typed having the effect of increasing the network traffic considerably. For that cost, it will enable the line editing, *tab completion*, and *history* features of the shell.

USB CDC ACM

To configure Shell USB CDC ACM backend, simply add the snippet `cdc-acm-console` to your build:

```
west build -S cdc-acm-console [...]
```

Details on the configuration settings are captured in the following files:

- snippets/cdc-acm-console/cdc-acm-console.conf.
- snippets/cdc-acm-console/cdc-acm-console.overlay.

Bluetooth LE (NUS)

To configure Bluetooth LE (NUS) backend, simply add the snippet `nus-console` to your build:

```
west build -S nus-console [...]
```

Details on the configuration settings are captured in the following files:

- snippets/nus-console/nus-console.conf.
- snippets/nus-console/nus-console.overlay.

Segget RTT

To configure Segger RTT backend, add the following configurations to your build:

- `CONFIG_USE_SEGGER_RTT`
- `CONFIG_SHELL_BACKEND_RTT`
- `CONFIG_SHELL_BACKEND_SERIAL`

Details on additional configuration settings are captured in: samples/subsys/shell/shell_module/prj_minimal_rtt.conf.

Connecting to Segger RTT via TCP (on macOS, for example) On macOS JLinkRTTClient won't let you enter input. Instead, please use following procedure:

- Open up a first Terminal window and enter:

  ```
  JLinkRTTLogger -Device NRF52840_XXAA -RTTChannel 1 -if SWD -Speed 4000 ~/rtt.log
  ```

  (change device if required)

- Open up a second Terminal window and enter:

  ```
  nc localhost 19021
  ```

- Now you should have a network connection to RTT that will let you enter input to the shell.

4.22.3 Commands

Shell commands are organized in a tree structure and grouped into the following types:

- Root command (level 0): Gathered and alphabetically sorted in a dedicated memory section.
- Static subcommand (level > 0): Number and syntax must be known during compile time. Created in the software module.
- Dynamic subcommand (level > 0): Number and syntax does not need to be known during compile time. Created in the software module.

Commonly-used command groups

The following list is a set of useful command groups and how to enable them:

GPIO

- CONFIG_GPIO
- CONFIG_GPIO_SHELL

I2C

- CONFIG_I2C
- CONFIG_I2C_SHELL

Sensor

- CONFIG_SENSOR
- CONFIG_SENSOR_SHELL

Flash

- CONFIG_FLASH
- CONFIG_FLASH_SHELL

File-System

- CONFIG_FILE_SYSTEM
- CONFIG_FILE_SYSTEM_SHELL

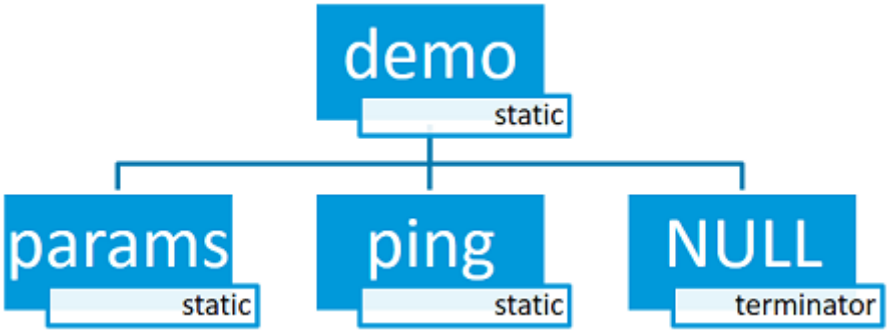
Creating commands

Use the following macros for adding shell commands:

- *SHELL_CMD_REGISTER* - Create root command. All root commands must have different name.
- *SHELL_COND_CMD_REGISTER* - Conditionally (if compile time flag is set) create root command. All root commands must have different name.
- *SHELL_CMD_ARG_REGISTER* - Create root command with arguments. All root commands must have different name.
- *SHELL_COND_CMD_ARG_REGISTER* - Conditionally (if compile time flag is set) create root command with arguments. All root commands must have different name.
- *SHELL_CMD* - Initialize a command.
- *SHELL_COND_CMD* - Initialize a command if compile time flag is set.
- *SHELL_EXPR_CMD* - Initialize a command if compile time expression is non-zero.
- *SHELL_CMD_ARG* - Initialize a command with arguments.
- *SHELL_COND_CMD_ARG* - Initialize a command with arguments if compile time flag is set.
- *SHELL_EXPR_CMD_ARG* - Initialize a command with arguments if compile time expression is non-zero.
- *SHELL_STATIC_SUBCMD_SET_CREATE* - Create a static subcommands array.
- *SHELL_SUBCMD_DICT_SET_CREATE* - Create a dictionary subcommands array.
- *SHELL_DYNAMIC_CMD_CREATE* - Create a dynamic subcommands array.

Commands can be created in any file in the system that includes include/zephyr/shell/shell.h. All created commands are available for all shell instances.

Static commands Example code demonstrating how to create a root command with static subcommands.




```
/* Creating subcommands (level 1 command) array for command "demo". */
SHELL_STATIC_SUBCMD_SET_CREATE(sub_demo,
        SHELL_CMD(params, NULL, "Print params command.",
                                                cmd_demo_params),
        SHELL_CMD(ping,   NULL, "Ping command.", cmd_demo_ping),
        SHELL_SUBCMD_SET_END
);
/* Creating root (level 0) command "demo" */
SHELL_CMD_REGISTER(demo, &sub_demo, "Demo commands", NULL);
```

Example implementation can be found under following location: samples/subsys/shell/shell_module/src/main.c.

Dictionary commands

This is a special kind of static commands. Dictionary commands can be used every time you want to use a pair: (string <-> corresponding data) in a command handler. The string is usually a verbal description of a given data. The idea is to use the string as a command syntax that can be prompted by the shell and corresponding data can be used to process the command.

Let's use an example. Suppose you created a command to set an ADC gain. It is a perfect place where a dictionary can be used. The dictionary would be a set of pairs: (string: gain_value, int: value) where int value could be used with the ADC driver API.

Abstract code for this task would look like this:

```
static int gain_cmd_handler(const struct shell *sh,
                            size_t argc, char **argv, void *data)
{
        int gain;

        /* data is a value corresponding to called command syntax */
        gain = (int)data;
        adc_set_gain(gain);

        shell_print(sh, "ADC gain set to: %s\n"
                        "Value send to ADC driver: %d",
                        argv[0],
                        gain);

        return 0;
}

SHELL_SUBCMD_DICT_SET_CREATE(sub_gain, gain_cmd_handler,
        (gain_1, 1, "gain 1"), (gain_2, 2, "gain 2"),
```

(continues on next page)

(continued from previous page)

```
        (gain_1_2, 3, "gain 1/2"), (gain_1_4, 4, "gain 1/4")
);
SHELL_CMD_REGISTER(gain, &sub_gain, "Set ADC gain", NULL);
```

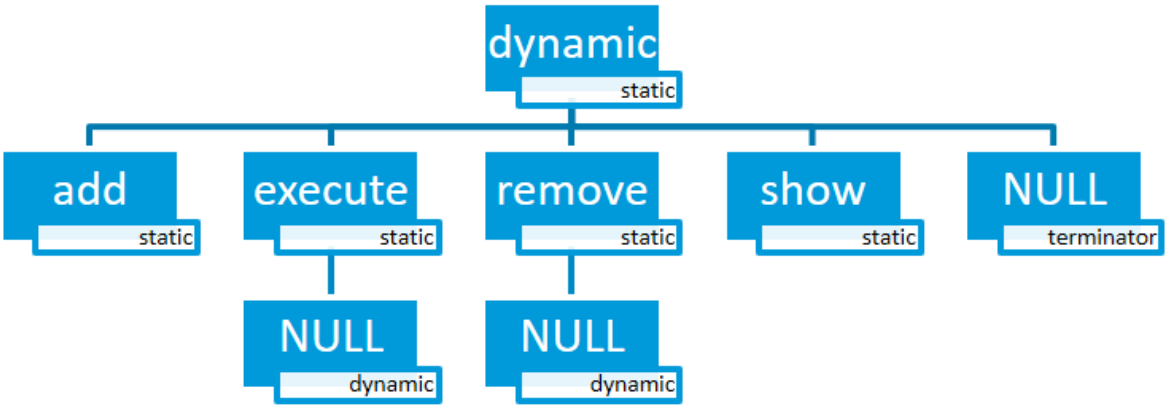
This is how it would look like in the shell:

```
uart:~$ gain ga
  gain_1    gain_2    gain_1_2  gain_1_4
uart:~$ gain gain_1
ADC gain set to: gain_1
Value send to ADC driver: 1
uart:~$ gain gain_2
ADC gain set to: gain_2
Value send to ADC driver: 2
uart:~$ gain gain_1_2
ADC gain set to: gain_1_2
Value send to ADC driver: 3
uart:~$ gain gain_1_4
ADC gain set to: gain_1_4
Value send to ADC driver: 4
uart:~$
```

Dynamic commands Example code demonstrating how to create a root command with static and dynamic subcommands. At the beginning dynamic command list is empty. New commands can be added by typing:

```
dynamic add <new_dynamic_command>
```

Newly added commands can be prompted or autocompleted with the Tab key.




```
/* Buffer for 10 dynamic commands */
static char dynamic_cmd_buffer[10][50];
```

(continues on next page)

(continued from previous page)

```
/* commands counter */
static uint8_t dynamic_cmd_cnt;

/* Function returning command dynamically created
 * in  dynamic_cmd_buffer.
 */
static void dynamic_cmd_get(size_t idx,
                            struct shell_static_entry *entry)
{
        if (idx < dynamic_cmd_cnt) {
                entry->syntax = dynamic_cmd_buffer[idx];
                entry->handler  = NULL;
                entry->subcmd = NULL;
                entry->help = "Show dynamic command name.";
        } else {
                /* if there are no more dynamic commands available
                 * syntax must be set to NULL.
                 */
                entry->syntax = NULL;
        }
}

SHELL_DYNAMIC_CMD_CREATE(m_sub_dynamic_set, dynamic_cmd_get);
SHELL_STATIC_SUBCMD_SET_CREATE(m_sub_dynamic,
        SHELL_CMD(add, NULL,"Add new command to dynamic_cmd_buffer and"
                  " sort them alphabetically.",
                  cmd_dynamic_add),
        SHELL_CMD(execute, &m_sub_dynamic_set,
                  "Execute a command.", cmd_dynamic_execute),
        SHELL_CMD(remove, &m_sub_dynamic_set,
                  "Remove a command from dynamic_cmd_buffer.",
                  cmd_dynamic_remove),
        SHELL_CMD(show, NULL,
                  "Show all commands in dynamic_cmd_buffer.",
                  cmd_dynamic_show),
        SHELL_SUBCMD_SET_END
);
SHELL_CMD_REGISTER(dynamic, &m_sub_dynamic,
           "Demonstrate dynamic command usage.", cmd_dynamic);
```

Example implementation can be found under following location: samples/subsys/shell/shell_module/src/dynamic_cmd.c.

Commands execution

Each command or subcommand may have a handler. The shell executes the handler that is found deepest in the command tree and further subcommands (without a handler) are passed as arguments. Characters within parentheses are treated as one argument. If shell won't find a handler it will display an error message.

Commands can be also executed from a user application using any active backend and a function *shell_execute_cmd()*, as shown in this example:

```
int main(void)
{
        /* Below code will execute "clear" command on a DUMMY backend */
        shell_execute_cmd(NULL, "clear");

        /* Below code will execute "shell colors off" command on
```

(continues on next page)

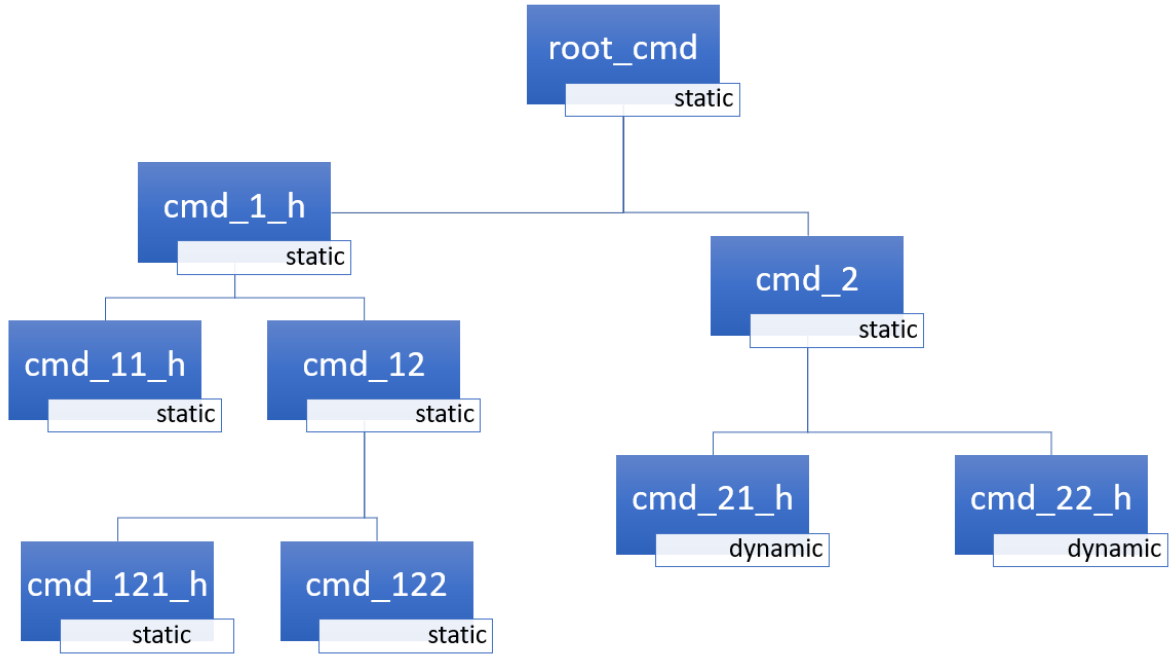
(continued from previous page)

```
         * an UART backend
         */
        shell_execute_cmd(shell_backend_uart_get_ptr(),
                          "shell colors off");
}
```

Enable the DUMMY backend by setting the Kconfig `CONFIG_SHELL_BACKEND_DUMMY` option.

Commands execution example Let's assume a command structure as in the following figure, where:

- `root_cmd` - root command without a handler
- `cmd_xxx_h` - command has a handler
- `cmd_xxx` - command does not have a handler



Example 1 Sequence: `root_cmd cmd_1_h cmd_12_h cmd_121_h parameter` will execute command `cmd_121_h` and `parameter` will be passed as an argument.

Example 2 Sequence: `root_cmd cmd_2 cmd_22_h parameter1 parameter2` will execute command `cmd_22_h` and `parameter1 parameter2` will be passed as an arguments.

Example 3 Sequence: `root_cmd cmd_1_h parameter1 cmd_121_h parameter2` will execute command `cmd_1_h` and `parameter1`, `cmd_121_h` and `parameter2` will be passed as an arguments.

Example 4 Sequence: `root_cmd parameter cmd_121_h parameter2` will not execute any command.

Command handler Simple command handler implementation:

```
static int cmd_handler(const struct shell *sh, size_t argc,
                       char **argv)
{
        ARG_UNUSED(argc);
        ARG_UNUSED(argv);

        shell_fprintf(shell, SHELL_INFO, "Print info message\n");

        shell_print(sh, "Print simple text.");

        shell_warn(sh, "Print warning text.");

        shell_error(sh, "Print error text.");

        return 0;
}
```

Function *shell_fprintf()* or the shell print macros: *shell_print*, *shell_info*, *shell_warn* and *shell_error* can be used from the command handler or from threads, but not from an interrupt context. Instead, interrupt handlers should use *Logging* for printing.

Command help Every user-defined command or subcommand can have its own help description. The help for commands and subcommands can be created with respective macros: *SHELL_CMD_REGISTER*, *SHELL_CMD_ARG_REGISTER*, *SHELL_CMD*, and *SHELL_CMD_ARG*.

Shell prints this help message when you call a command or subcommand with `-h` or `--help` parameter.

Parent commands In the subcommand handler, you can access both the parameters passed to commands or the parent commands, depending on how you index `argv`.

- When indexing `argv` with positive numbers, you can access the parameters.
- When indexing `argv` with negative numbers, you can access the parent commands.
- The subcommand to which the handler belongs has the `argv` index of 0.

```
static int cmd_handler(const struct shell *sh, size_t argc,
                       char **argv)
{
        ARG_UNUSED(argc);

        /* If it is a subcommand handler parent command syntax
         * can be found using argv[-1].
         */
        shell_print(sh, "This command has a parent command: %s",
                    argv[-1]);

        /* Print this command syntax */
        shell_print(sh, "This command syntax is: %s", argv[0]);

        /* Print first argument */
        shell_print(sh, "%s", argv[1]);

        return 0;
}
```

Built-in commands

These commands are activated by CONFIG_SHELL_CMDS set to y.

- **clear** - Clears the screen.
- **history** - Shows the recently entered commands.
- **resize** - Must be executed when terminal width is different than 80 characters or after each change of terminal width. It ensures proper multiline text display and ←, →, End, Home keys handling. Currently this command works only with UART flow control switched on. It can be also called with a subcommand:
  - **default** - Shell will send terminal width = 80 to the terminal and assume successful delivery.

  These command needs extra activation: CONFIG_SHELL_CMDS_RESIZE set to y.
- **select** - It can be used to set new root command. Exit to main command tree is with alt+r. This command needs extra activation: CONFIG_SHELL_CMDS_SELECT set to y.
- **shell** - Root command with useful shell-related subcommands like:
  - **echo** - Toggles shell echo.
  - **colors** - Toggles colored syntax. This might be helpful in case of Bluetooth shell to limit the amount of transferred bytes.
  - **stats** - Shows shell statistics.

4.22.4 Tab Feature

The Tab button can be used to suggest commands or subcommands. This feature is enabled by CONFIG_SHELL_TAB set to y. It can also be used for partial or complete auto-completion of commands. This feature is activated by CONFIG_SHELL_TAB_AUTOCOMPLETION set to y. When user starts writing a command and presses the Tab button then the shell will do one of 3 possible things:

- Autocomplete the command.
- Prompts available commands and if possible partly completes the command.
- Will not do anything if there are no available or matching commands.

```
uart:~$ log
  backend        disable        enable         go             halt
  list_backends  status
uart:~$ log enable
  dbg   err   inf   none  wrn
uart:~$ log enable err
  app               app_test          log              os
  shell.shell_uart  shell_uart
uart:~$ log enable err shell
  shell.shell_uart  shell_uart
uart:~$ log enable err shell
```

4.22.5 History Feature

This feature enables commands history in the shell. It is activated by: CONFIG_SHELL_HISTORY set to y. History can be accessed using keys: ↑ ↓ or Ctrl+n and Ctrl+p if meta keys are active. Number of commands that can be stored depends on size of CONFIG_SHELL_HISTORY_BUFFER parameter.

4.22.6 Wildcards Feature

The shell module can handle wildcards. Wildcards are interpreted correctly when expanded command and its subcommands do not have a handler. For example, if you want to set logging level to `err` for the `app` and `app_test` modules you can execute the following command:

```
log enable err a*
```

```
uart:~$ log status
module_name                                     | current | built-in
------------------------------------------------------------------
app                                             | inf     | inf
app_test                                        | inf     | inf
shell.uart_shell                                | inf     | inf
uart:~$ log enable err a*
uart:~$ log status
module_name                                     | current | built-in
------------------------------------------------------------------
app                                             | err     | inf
app_test                                        | err     | inf
shell.uart_shell                                | inf     | inf
uart:~$
```

This feature is activated by `CONFIG_SHELL_WILDCARD` set to `y`.

4.22.7 Meta Keys Feature

The shell module supports the following meta keys:

Table 64: Implemented meta keys

Meta keys	Action
`Ctrl+a`	Moves the cursor to the beginning of the line.
`Ctrl+b`	Moves the cursor backward one character.
`Ctrl+c`	Preserves the last command on the screen and starts a new command in a new line.
`Ctrl+d`	Deletes the character under the cursor.
`Ctrl+e`	Moves the cursor to the end of the line.
`Ctrl+f`	Moves the cursor forward one character.
`Ctrl+k`	Deletes from the cursor to the end of the line.
`Ctrl+l`	Clears the screen and leaves the currently typed command at the top of the screen.
`Ctrl+n`	Moves in history to next entry.
`Ctrl+p`	Moves in history to previous entry.
`Ctrl+u`	Clears the currently typed command.
`Ctrl+w`	Removes the word or part of the word to the left of the cursor. Words separated by period instead of space are treated as one word.
`Alt+b`	Moves the cursor backward one word.
`Alt+f`	Moves the cursor forward one word.

This feature is activated by `CONFIG_SHELL_METAKEYS` set to `y`.

4.22.8 Getopt Feature

Some shell users apart from subcommands might need to use options as well. the arguments string, looking for supported options. Typically, this task is accomplished by the `getopt` family functions.

For this purpose shell supports the getopt and getopt_long libraries available in the FreeBSD project. This feature is activated by: `CONFIG_POSIX_C_LIB_EXT` set to `y` and `CONFIG_GETOPT_LONG` set to `y`.

This feature can be used in thread safe as well as non thread safe manner. The former is full compatible with regular getopt usage while the latter a bit differs.

An example non-thread safe usage:

```
char *cvalue = NULL;
while ((char c = getopt(argc, argv, "abhc:")) != -1) {
      switch (c) {
      case 'c':
              cvalue = optarg;
              break;
      default:
              break;
      }
}
```

An example thread safe usage:

```
char *cvalue = NULL;
struct getopt_state *state;
while ((char c = getopt(argc, argv, "abhc:")) != -1) {
      state = getopt_state_get();
      switch (c) {
      case 'c':
              cvalue = state->optarg;
              break;
      default:
              break;
      }
}
```

Thread safe getopt functionality is activated by `CONFIG_SHELL_GETOPT` set to `y`.

4.22.9 Obscured Input Feature

With the obscured input feature, the shell can be used for implementing a login prompt or other user interaction whereby the characters the user types should not be revealed on screen, such as when entering a password.

Once the obscured input has been accepted, it is normally desired to return the shell to normal operation. Such runtime control is possible with the `shell_obscure_set` function.

An example of login and logout commands using this feature is located in samples/subsys/shell/shell_module/src/main.c and the config file samples/subsys/shell/shell_module/prj_login.conf.

This feature is activated upon startup by `CONFIG_SHELL_START_OBSCURED` set to `y`. With this set either way, the option can still be controlled later at runtime. `CONFIG_SHELL_CMDS_SELECT` is useful to prevent entry of any other command besides a login command, by means of the `shell_set_root_cmd` function. Likewise, `CONFIG_SHELL_PROMPT_UART` allows you to set the prompt upon startup, but it can be changed later with the `shell_prompt_change` function.

4.22.10 Shell Logger Backend Feature

Shell instance can act as the *Logging* backend. Shell ensures that log messages are correctly multiplexed with shell output. Log messages from logger thread are enqueued and processed in the shell thread. Logger thread will block for configurable amount of time if queue is full, blocking logger thread context for that time. Oldest log message is removed from the queue after timeout and new message is enqueued. Use the `shell stats show` command to retrieve number of log messages dropped by the shell instance. Log queue size and timeout are *SHELL_DEFINE* arguments.

This feature is activated by: `CONFIG_SHELL_LOG_BACKEND` set to `y`.

Warning

Enqueuing timeout must be set carefully when multiple backends are used in the system. The shell instance could have a slow transport or could block, for example, by a UART with hardware flow control. If timeout is set too high, the logger thread could be blocked and impact other logger backends.

Warning

As the shell is a complex logger backend, it can not output logs if the application crashes before the shell thread is running. In this situation, you can enable one of the simple logging backends instead, such as UART (`CONFIG_LOG_BACKEND_UART`) or RTT (`CONFIG_LOG_BACKEND_RTT`), which are available earlier during system initialization.

4.22.11 RTT Backend Channel Selection

Instead of using the shell as a logger backend, RTT shell backend and RTT log backend can also be used simultaneously, but over different channels. By separating them, the log can be captured or monitored without shell output or the shell may be scripted without log interference. Enabling both the Shell RTT backend and the Log RTT backend does not work by default, because both default to channel `0`. There are two options:

1. The Shell buffer can use an alternate channel, for example using `CONFIG_SHELL_BACKEND_RTT_BUFFER` set to `1`. This allows monitoring the log using JLinkRTTViewer while a script interfaces over channel 1.

2. The Log buffer can use an alternate channel, for example using `CONFIG_LOG_BACKEND_RTT_BUFFER` set to `1`. This allows interactive use of the shell through JLinkRTTViewer, while the log is written to file.

See *shell backends* for details on how to enable RTT as a Shell backend.

4.22.12 Usage

The following code shows a simple use case of this library:

```
int main(void)
{

}

static int cmd_demo_ping(const struct shell *sh, size_t argc,
                         char **argv)
```

(continues on next page)

(continued from previous page)

```
{
        ARG_UNUSED(argc);
        ARG_UNUSED(argv);

        shell_print(sh, "pong");
        return 0;
}

static int cmd_demo_params(const struct shell *sh, size_t argc,
                           char **argv)
{
        int cnt;

        shell_print(sh, "argc = %d", argc);
        for (cnt = 0; cnt < argc; cnt++) {
                shell_print(sh, "  argv[%d] = %s", cnt, argv[cnt]);
        }
        return 0;
}

/* Creating subcommands (level 1 command) array for command "demo". */
SHELL_STATIC_SUBCMD_SET_CREATE(sub_demo,
        SHELL_CMD(params, NULL, "Print params command.",
                                                cmd_demo_params),
        SHELL_CMD(ping,   NULL, "Ping command.", cmd_demo_ping),
        SHELL_SUBCMD_SET_END
);
/* Creating root (level 0) command "demo" without a handler */
SHELL_CMD_REGISTER(demo, &sub_demo, "Demo commands", NULL);

/* Creating root (level 0) command "version" */
SHELL_CMD_REGISTER(version, NULL, "Show kernel version", cmd_version);
```

Users may use the Tab key to complete a command/subcommand or to see the available subcommands for the currently entered command level. For example, when the cursor is positioned at the beginning of the command line and the Tab key is pressed, the user will see all root (level 0) commands:

```
clear  demo  shell  history  log  resize  version
```

Note

To view the subcommands that are available for a specific command, you must first type a space after this command and then hit Tab.

These commands are registered by various modules, for example:

- **clear**, **shell**, **history**, and **resize** are built-in commands which have been registered by subsys/shell/shell.c
- **demo** and **version** have been registered in example code above by main.c
- **log** has been registered by subsys/logging/log_cmds.c

Then, if a user types a **demo** command and presses the Tab key, the shell will only print the subcommands registered for this command:

```
params  ping
```

4.22.13 API Reference

Related code samples

Custom Shell module
Register shell commands using the Shell API

Telnet console
Access Zephyr shell over telnet.

group `shell_api`

Shell API.

Since
1.14

Version
1.0.0

Defines

`SHELL_CMD_ARG_REGISTER`(syntax, subcmd, help, handler, mandatory, optional)

Macro for defining and adding a root command (level 0) with required number of arguments.

Note

Each root command shall have unique syntax. If a command will be called with wrong number of arguments shell will print an error message and command handler will not be called.

Parameters

- `syntax` – **[in]** Command syntax (for example: history).
- `subcmd` – **[in]** Pointer to a subcommands array.
- `help` – **[in]** Pointer to a command help string.
- `handler` – **[in]** Pointer to a function handler.
- `mandatory` – **[in]** Number of mandatory arguments including command name.
- `optional` – **[in]** Number of optional arguments.

`SHELL_COND_CMD_ARG_REGISTER`(flag, syntax, subcmd, help, handler, mandatory, optional)

Macro for defining and adding a conditional root command (level 0) with required number of arguments.

Macro can be used to create a command which can be conditionally present. It is and alternative to #ifdefs around command registration and command handler. If command is disabled handler and subcommands are removed from the application.

See also

SHELL_CMD_ARG_REGISTER for details.

Parameters

- `flag` – **[in]** Compile time flag. Command is present only if flag exists and equals 1.
- `syntax` – **[in]** Command syntax (for example: history).
- `subcmd` – **[in]** Pointer to a subcommands array.
- `help` – **[in]** Pointer to a command help string.
- `handler` – **[in]** Pointer to a function handler.
- `mandatory` – **[in]** Number of mandatory arguments including command name.
- `optional` – **[in]** Number of optional arguments.

`SHELL_CMD_REGISTER`(syntax, subcmd, help, handler)

Macro for defining and adding a root command (level 0) with arguments.

Note

All root commands must have different name.

Parameters

- `syntax` – **[in]** Command syntax (for example: history).
- `subcmd` – **[in]** Pointer to a subcommands array.
- `help` – **[in]** Pointer to a command help string.
- `handler` – **[in]** Pointer to a function handler.

`SHELL_COND_CMD_REGISTER`(flag, syntax, subcmd, help, handler)

Macro for defining and adding a conditional root command (level 0) with arguments.

See also

SHELL_COND_CMD_ARG_REGISTER.

Parameters

- `flag` – **[in]** Compile time flag. Command is present only if flag exists and equals 1.
- `syntax` – **[in]** Command syntax (for example: history).
- `subcmd` – **[in]** Pointer to a subcommands array.
- `help` – **[in]** Pointer to a command help string.
- `handler` – **[in]** Pointer to a function handler.

SHELL_STATIC_SUBCMD_SET_CREATE(name, ...)

Macro for creating a subcommand set.

It must be used outside of any function body.

Example usage:

```
SHELL_STATIC_SUBCMD_SET_CREATE(
   foo,
   SHELL_CMD(abc, ...),
   SHELL_CMD(def, ...),
   SHELL_SUBCMD_SET_END
)
```

Parameters

- **name** – **[in]** Name of the subcommand set.
- **...** – **[in]** List of commands created with *SHELL_CMD_ARG* or or *SHELL_CMD*

SHELL_SUBCMD_SET_CREATE(_name, _parent)

Create set of subcommands.

Commands to this set are added using *SHELL_SUBCMD_ADD* and *SHELL_SUBCMD_COND_ADD*. Commands can be added from multiple files.

Parameters

- **_name** – **[in]** Name of the set. _name is used to refer the set in the parent command.
- **_parent** – **[in]** Set of comma separated parent commands in parenthesis, e.g. (foo_cmd) if subcommands are for the root command "foo_cmd".

SHELL_SUBCMD_COND_ADD(_flag, _parent, _syntax, _subcmd, _help, _handler, _mand, _opt)

Conditionally add command to the set of subcommands.

Add command to the set created with *SHELL_SUBCMD_SET_CREATE*.

Note

The name of the section is formed as concatenation of number of parent commands, names of all parent commands and own syntax. Number of parent commands is added to ensure that section prefix is unique. Without it subcommands of (foo) and (foo, cmd1) would mix.

Parameters

- **_flag** – **[in]** Compile time flag. Command is present only if flag exists and equals 1.
- **_parent** – **[in]** Parent command sequence. Comma separated in parenthesis.
- **_syntax** – **[in]** Command syntax (for example: history).
- **_subcmd** – **[in]** Pointer to a subcommands array.
- **_help** – **[in]** Pointer to a command help string.
- **_handler** – **[in]** Pointer to a function handler.
- **_mand** – **[in]** Number of mandatory arguments including command name.

- **_opt** – **[in]** Number of optional arguments.

SHELL_SUBCMD_ADD(_parent, _syntax, _subcmd, _help, _handler, _mand, _opt)

Add command to the set of subcommands.

Add command to the set created with *SHELL_SUBCMD_SET_CREATE*.

Parameters

- **_parent** – **[in]** Parent command sequence. Comma separated in parenthesis.
- **_syntax** – **[in]** Command syntax (for example: history).
- **_subcmd** – **[in]** Pointer to a subcommands array.
- **_help** – **[in]** Pointer to a command help string.
- **_handler** – **[in]** Pointer to a function handler.
- **_mand** – **[in]** Number of mandatory arguments including command name.
- **_opt** – **[in]** Number of optional arguments.

SHELL_SUBCMD_SET_END

Define ending subcommands set.

SHELL_DYNAMIC_CMD_CREATE(name, get)

Macro for creating a dynamic entry.

Parameters

- **name** – **[in]** Name of the dynamic entry.
- **get** – **[in]** Pointer to the function returning dynamic commands array

SHELL_CMD_ARG(syntax, subcmd, help, handler, mand, opt)

Initializes a shell command with arguments.

> **Note**
>
> If a command will be called with wrong number of arguments shell will print an error message and command handler will not be called.

Parameters

- **syntax** – **[in]** Command syntax (for example: history).
- **subcmd** – **[in]** Pointer to a subcommands array.
- **help** – **[in]** Pointer to a command help string.
- **handler** – **[in]** Pointer to a function handler.
- **mand** – **[in]** Number of mandatory arguments including command name.
- **opt** – **[in]** Number of optional arguments.

SHELL_COND_CMD_ARG(flag, syntax, subcmd, help, handler, mand, opt)

Initializes a conditional shell command with arguments.

See also

SHELL_CMD_ARG. Based on the flag, creates a valid entry or an empty command which is ignored by the *shell*. It is an alternative to #ifdefs around command registration and command handler. However, empty structure is present in the flash even if command is disabled (subcommands and handler are removed). Macro internally handles case if flag is not defined so flag must be provided without any wrapper, e.g.: *SHELL_COND_CMD_ARG(CONFIG_FOO, ...)*

Parameters

- `flag` – **[in]** Compile time flag. Command is present only if flag exists and equals 1.
- `syntax` – **[in]** Command syntax (for example: history).
- `subcmd` – **[in]** Pointer to a subcommands array.
- `help` – **[in]** Pointer to a command help string.
- `handler` – **[in]** Pointer to a function handler.
- `mand` – **[in]** Number of mandatory arguments including command name.
- `opt` – **[in]** Number of optional arguments.

`SHELL_EXPR_CMD_ARG`(_expr, _syntax, _subcmd, _help, _handler, _mand, _opt)

Initializes a conditional shell command with arguments if expression gives non-zero result at compile time.

See also

SHELL_CMD_ARG. Based on the expression, creates a valid entry or an empty command which is ignored by the *shell*. It should be used instead of *SHELL_COND_CMD_ARG* if condition is not a single configuration flag, e.g.: *SHELL_EXPR_CMD_ARG*(*IS_ENABLED(CONFIG_FOO)* && *IS_ENABLED(CONFIG_FOO_SETTING_1)*, ...)

Parameters

- `_expr` – **[in]** Expression.
- `_syntax` – **[in]** Command syntax (for example: history).
- `_subcmd` – **[in]** Pointer to a subcommands array.
- `_help` – **[in]** Pointer to a command help string.
- `_handler` – **[in]** Pointer to a function handler.
- `_mand` – **[in]** Number of mandatory arguments including command name.
- `_opt` – **[in]** Number of optional arguments.

`SHELL_CMD`(_syntax, _subcmd, _help, _handler)

Initializes a shell command.

Parameters

- `_syntax` – **[in]** Command syntax (for example: history).

- **_subcmd** – **[in]** Pointer to a subcommands array.
- **_help** – **[in]** Pointer to a command help string.
- **_handler** – **[in]** Pointer to a function handler.

SHELL_COND_CMD(_flag, _syntax, _subcmd, _help, _handler)

Initializes a conditional shell command.

See also

SHELL_COND_CMD_ARG.

Parameters

- **_flag** – **[in]** Compile time flag. Command is present only if flag exists and equals 1.
- **_syntax** – **[in]** Command syntax (for example: history).
- **_subcmd** – **[in]** Pointer to a subcommands array.
- **_help** – **[in]** Pointer to a command help string.
- **_handler** – **[in]** Pointer to a function handler.

SHELL_EXPR_CMD(_expr, _syntax, _subcmd, _help, _handler)

Initializes shell command if expression gives non-zero result at compile time.

See also

SHELL_EXPR_CMD_ARG.

Parameters

- **_expr** – **[in]** Compile time expression. Command is present only if expression is non-zero.
- **_syntax** – **[in]** Command syntax (for example: history).
- **_subcmd** – **[in]** Pointer to a subcommands array.
- **_help** – **[in]** Pointer to a command help string.
- **_handler** – **[in]** Pointer to a function handler.

SHELL_CMD_DICT_CREATE(_data, _handler)

SHELL_SUBCMD_DICT_SET_CREATE(_name, _handler, ...)

Initializes shell dictionary commands.

This is a special kind of static commands. Dictionary commands can be used every time you want to use a pair: (string <-> corresponding data) in a command handler. The string is usually a verbal description of a given data. The idea is to use the string as a command syntax that can be prompted by the shell and corresponding data can be used to process the command.

Example usage:

```
static int my_handler(const struct shell *sh,
                      size_t argc, char **argv, void *data)
{
   int val = (int)data;

   shell_print(sh, "(syntax, value) : (%s, %d)", argv[0], val);
   return 0;
}

SHELL_SUBCMD_DICT_SET_CREATE(sub_dict_cmds, my_handler,
   (value_0, 0, "value 0"), (value_1, 1, "value 1"),
   (value_2, 2, "value 2"), (value_3, 3, "value 3")
);
SHELL_CMD_REGISTER(dictionary, &sub_dict_cmds, NULL, NULL);
```

See also

shell_dict_cmd_handler

Parameters

- **_name** – **[in]** Name of the dictionary subcommand set
- **_handler** – **[in]** Command handler common for all dictionary commands.
- **...** – **[in]** Dictionary triplets: (command_syntax, value, helper). Value will be passed to the _handler as user data.

SHELL_DEFAULT_BACKEND_CONFIG_FLAGS

SHELL_DEFINE(_name, _prompt, _transport_iface, _log_queue_size, _log_timeout, _shell_flag)

Macro for defining a shell instance.

Parameters

- **_name** – **[in]** Instance name.
- **_prompt** – **[in]** Shell default prompt string.
- **_transport_iface** – **[in]** Pointer to the transport interface.
- **_log_queue_size** – **[in]** Logger processing queue size.
- **_log_timeout** – **[in]** Logger thread timeout in milliseconds on full log queue. If queue is full logger thread is blocked for given amount of time before log message is dropped.
- **_shell_flag** – **[in]** Shell output newline sequence.

SHELL_NORMAL

Terminal default text color for shell_fprintf function.

SHELL_INFO

Green text color for shell_fprintf function.

SHELL_OPTION

Cyan text color for shell_fprintf function.

SHELL_WARNING
: Yellow text color for shell_fprintf function.

SHELL_ERROR
: Red text color for shell_fprintf function.

shell_fprintf(sh, color, fmt, ...)

shell_info(_sh, _ft, ...)
: Print info message to the shell.

  See shell_fprintf.

  Parameters

  - **_sh** – **[in]** Pointer to the shell instance.
  - **_ft** – **[in]** Format string.
  - **...** – **[in]** List of parameters to print.

shell_print(_sh, _ft, ...)
: Print normal message to the shell.

  See shell_fprintf.

  Parameters

  - **_sh** – **[in]** Pointer to the shell instance.
  - **_ft** – **[in]** Format string.
  - **...** – **[in]** List of parameters to print.

shell_warn(_sh, _ft, ...)
: Print warning message to the shell.

  See shell_fprintf.

  Parameters

  - **_sh** – **[in]** Pointer to the shell instance.
  - **_ft** – **[in]** Format string.
  - **...** – **[in]** List of parameters to print.

shell_error(_sh, _ft, ...)
: Print error message to the shell.

  See shell_fprintf.

  Parameters

  - **_sh** – **[in]** Pointer to the shell instance.
  - **_ft** – **[in]** Format string.
  - **...** – **[in]** List of parameters to print.

SHELL_CMD_HELP_PRINTED
: Command's help has been printed.

Typedefs

typedef void (*`shell_dynamic_get`)(size_t idx, struct *shell_static_entry* *entry)

Shell dynamic command descriptor.

Function shall fill the received *shell_static_entry* structure with requested (idx) dynamic subcommand data. If there is more than one dynamic subcommand available, the function shall ensure that the returned commands: entry->syntax are sorted in alphabetical order. If idx exceeds the available dynamic subcommands, the function must write to entry->syntax NULL value. This will indicate to the shell module that there are no more dynamic commands to read.

typedef bool (*`shell_device_filter_t`)(const struct *device* *dev)

Filter callback type, for use with shell_device_lookup_filter.

This is used as an argument of shell_device_lookup_filter to only return devices that match a specific condition, implemented by the filter.

Param dev
pointer to a struct device.

Return
bool, true if the filter matches the device type.

typedef int (*`shell_cmd_handler`)(const struct *shell* *sh, size_t argc, char **argv)

Shell command handler prototype.

Param sh
Shell instance.

Param argc
Arguments count.

Param argv
Arguments.

Retval 0
Successful command execution.

Retval 1
Help printed and command not executed.

Retval -EINVAL
Argument validation failed.

Retval -ENOEXEC
Command not executed.

typedef int (*`shell_dict_cmd_handler`)(const struct *shell* *sh, size_t argc, char **argv, void *data)

Shell dictionary command handler prototype.

Param sh
Shell instance.

Param argc
Arguments count.

Param argv
Arguments.

Param data
Pointer to the user data.

Retval 0
Successful command execution.

Retval 1
Help printed and command not executed.

Retval -EINVAL
Argument validation failed.

Retval -ENOEXEC
Command not executed.

typedef void (*`shell_transport_handler_t`)(enum *shell_transport_evt* evt, void *context)

typedef void (*`shell_uninit_cb_t`)(const struct *shell* *sh, int res)

typedef void (*`shell_bypass_cb_t`)(const struct *shell* *sh, uint8_t *data, size_t len)

Bypass callback.

Param sh
Shell instance.

Param data
Raw data from transport.

Param len
Data length.

Enums

enum `shell_receive_state`

Values:

enumerator `SHELL_RECEIVE_DEFAULT`

enumerator `SHELL_RECEIVE_ESC`

enumerator `SHELL_RECEIVE_ESC_SEQ`

enumerator `SHELL_RECEIVE_TILDE_EXP`

enum `shell_state`

Values:

enumerator `SHELL_STATE_UNINITIALIZED`

enumerator `SHELL_STATE_INITIALIZED`

enumerator `SHELL_STATE_ACTIVE`

enumerator `SHELL_STATE_PANIC_MODE_ACTIVE`

Panic activated.

enumerator SHELL_STATE_PANIC_MODE_INACTIVE
Panic requested, not supported.

enum shell_transport_evt
Shell transport event.

Values:

enumerator SHELL_TRANSPORT_EVT_RX_RDY

enumerator SHELL_TRANSPORT_EVT_TX_RDY

enum shell_signal
Values:

enumerator SHELL_SIGNAL_RXRDY

enumerator SHELL_SIGNAL_LOG_MSG

enumerator SHELL_SIGNAL_KILL

enumerator SHELL_SIGNAL_TXDONE

enumerator SHELL_SIGNALS

enum shell_flag
Flags for setting shell output newline sequence.

Values:

enumerator SHELL_FLAG_CRLF_DEFAULT = (1 « 0)
Do not map CR or LF.

enumerator SHELL_FLAG_OLF_CRLF = (1 « 1)
Map LF to CRLF on output.

Functions

const struct *device* *shell_device_lookup(size_t idx, const char *prefix)
Get by index a device that matches .

This can be used, for example, to identify I2C_1 as the second I2C device.

Devices that failed to initialize or do not have a non-empty name are excluded from the candidates for a match.

Parameters

- idx – the device number starting from zero.
- prefix – optional name prefix used to restrict candidate devices. Indexing is done relative to devices with names that start with this text. Pass null if no prefix match is required.

const struct *device* *`shell_device_filter`(size_t idx, *shell_device_filter_t* filter)

Get a device by index and filter.

This can be used to return devices matching a specific type.

Devices that the filter returns false for, failed to initialize or do not have a non-empty name are excluded from the candidates for a match.

Parameters

- `idx` – the device number starting from zero.
- `filter` – a pointer to a shell_device_filter_t function that returns true if the device matches the filter.

int `shell_init`(const struct *shell* *sh, const void *transport_config, struct *shell_backend_config_flags* cfg_flags, bool log_backend, uint32_t init_log_level)

Function for initializing a transport layer and internal shell state.

Parameters

- `sh` – **[in]** Pointer to shell instance.
- `transport_config` – **[in]** Transport configuration during initialization.
- `cfg_flags` – **[in]** Initial backend configuration flags. Shell will copy this data.
- `log_backend` – If true, the console will be used as logger backend.
- `init_log_level` – **[in]** Default severity level for the logger.

Returns

Standard error code.

void `shell_uninit`(const struct *shell* *sh, *shell_uninit_cb_t* cb)

Uninitializes the transport layer and the internal shell state.

Parameters

- `sh` – Pointer to shell instance.
- `cb` – Callback called when uninitialization is completed.

int `shell_start`(const struct *shell* *sh)

Function for starting shell processing.

Parameters

- `sh` – Pointer to the shell instance.

Returns

Standard error code.

int `shell_stop`(const struct *shell* *sh)

Function for stopping shell processing.

Parameters

- `sh` – Pointer to shell instance.

Returns

Standard error code.

void `shell_fprintf_impl`(const struct *shell* *sh, enum shell_vt100_color color, const char *fmt, ...)

printf-like function which sends formatted data stream to the shell.

This function can be used from the command handler or from threads, but not from an interrupt context.

Parameters

- `sh` – **[in]** Pointer to the shell instance.
- `color` – **[in]** Printed text color.
- `fmt` – **[in]** Format string.
- `...` – **[in]** List of parameters to print.

void `shell_vfprintf`(const struct *shell* *sh, enum shell_vt100_color color, const char *fmt, va_list args)

vprintf-like function which sends formatted data stream to the shell.

This function can be used from the command handler or from threads, but not from an interrupt context. It is similar to *shell_fprintf()* but takes a va_list instead of variable arguments.

Parameters

- `sh` – **[in]** Pointer to the shell instance.
- `color` – **[in]** Printed text color.
- `fmt` – **[in]** Format string.
- `args` – **[in]** List of parameters to print.

void `shell_hexdump_line`(const struct *shell* *sh, unsigned int offset, const uint8_t *data, size_t len)

Print a line of data in hexadecimal format.

Each line shows the offset, bytes and then ASCII representation.

For example:

00008010: 20 25 00 20 2f 48 00 08 80 05 00 20 af 46 00 | %. /H.. ... .F. |

Parameters

- `sh` – **[in]** Pointer to the shell instance.
- `offset` – **[in]** Offset to show for this line.
- `data` – **[in]** Pointer to data.
- `len` – **[in]** Length of data.

void `shell_hexdump`(const struct *shell* *sh, const uint8_t *data, size_t len)

Print data in hexadecimal format.

Parameters

- `sh` – **[in]** Pointer to the shell instance.
- `data` – **[in]** Pointer to data.
- `len` – **[in]** Length of data.

void `shell_process`(const struct *shell* *sh)

Process function, which should be executed when data is ready in the transport interface.

To be used if shell thread is disabled.

Parameters

- **sh** – **[in]** Pointer to the shell instance.

int **shell_prompt_change**(const struct *shell* *sh, const char *prompt)

Change displayed shell prompt.

Parameters

- **sh** – **[in]** Pointer to the shell instance.
- **prompt** – **[in]** New shell prompt.

Returns
0 Success.

Returns
-EINVAL Pointer to new prompt is not correct.

void **shell_help**(const struct *shell* *sh)

Prints the current command help.

Function will print a help string with: the currently entered command and subcommands (if they exist).

Parameters

- **sh** – **[in]** Pointer to the shell instance.

int **shell_execute_cmd**(const struct *shell* *sh, const char *cmd)

Execute command.

Pass command line to shell to execute.

Note: This by no means makes any of the commands a stable interface, so this function should only be used for debugging/diagnostic.

This function must not be called from shell command context!

Parameters

- **sh** – **[in]** Pointer to the shell instance. It can be NULL when the CONFIG_SHELL_BACKEND_DUMMY option is enabled.
- **cmd** – **[in]** Command to be executed.

Returns
Result of the execution

int **shell_set_root_cmd**(const char *cmd)

Set root command for all shell instances.

It allows setting from the code the root command. It is an equivalent of calling select command with one of the root commands as the argument (e.g “select log”) except it sets command for all shell instances.

Parameters

- **cmd** – String with one of the root commands or null pointer to reset.

Return values

- **0** – if root command is set.
- **-EINVAL** – if invalid root command is provided.

void **shell_set_bypass**(const struct *shell* *sh, *shell_bypass_cb_t* bypass)

Set bypass callback.

Bypass callback is called whenever data is received. Shell is bypassed and data is passed directly to the callback. Use null to disable bypass functionality.

Parameters

- **sh** – **[in]** Pointer to the shell instance.
- **bypass** – **[in]** Bypass callback or null to disable.

bool **shell_ready**(const struct *shell* *sh)

Get shell readiness to execute commands.

Parameters

- **sh** – **[in]** Pointer to the shell instance.

Return values

- **true** – Shell backend is ready to execute commands.
- **false** – Shell backend is not initialized or not started.

int **shell_insert_mode_set**(const struct *shell* *sh, bool val)

Allow application to control text insert mode.

Value is modified atomically and the previous value is returned.

Parameters

- **sh** – **[in]** Pointer to the shell instance.
- **val** – **[in]** Insert mode.

Return values

- **0** – or 1: previous value
- **-EINVAL** – if shell is NULL.

int **shell_use_colors_set**(const struct *shell* *sh, bool val)

Allow application to control whether terminal output uses colored syntax.

Value is modified atomically and the previous value is returned.

Parameters

- **sh** – **[in]** Pointer to the shell instance.
- **val** – **[in]** Color mode.

Return values

- **0** – or 1: previous value
- **-EINVAL** – if shell is NULL.

int **shell_use_vt100_set**(const struct *shell* *sh, bool val)

Allow application to control whether terminal is using vt100 commands.

Value is modified atomically and the previous value is returned.

Parameters

- **sh** – **[in]** Pointer to the shell instance.
- **val** – **[in]** vt100 mode.

Return values

- **0** – or 1: previous value
- **-EINVAL** – if shell is NULL.

int **shell_echo_set**(const struct *shell* *sh, bool val)

Allow application to control whether user input is echoed back.

Value is modified atomically and the previous value is returned.

Parameters

- `sh` – **[in]** Pointer to the shell instance.
- `val` – **[in]** Echo mode.

Return values

- `0` – or 1: previous value
- `-EINVAL` – if shell is NULL.

int `shell_obscure_set`(const struct *shell* *sh, bool obscure)

Allow application to control whether user input is obscured with asterisks — useful for implementing passwords.

Value is modified atomically and the previous value is returned.

Parameters

- `sh` – **[in]** Pointer to the shell instance.
- `obscure` – **[in]** Obscure mode.

Return values

- `0` – or 1: previous value.
- `-EINVAL` – if shell is NULL.

int `shell_mode_delete_set`(const struct *shell* *sh, bool val)

Allow application to control whether the delete key backspaces or deletes.

Value is modified atomically and the previous value is returned.

Parameters

- `sh` – **[in]** Pointer to the shell instance.
- `val` – **[in]** Delete mode.

Return values

- `0` – or 1: previous value
- `-EINVAL` – if shell is NULL.

int `shell_get_return_value`(const struct *shell* *sh)

Retrieve return value of most recently executed shell command.

Parameters

- `sh` – **[in]** Pointer to the shell instance

Return values

`return` – value of previous command

Variables

const struct *log_backend_api* `log_backend_shell_api`

union `shell_cmd_entry`

#include <*shell.h*> Shell command descriptor.

Public Members

shell_dynamic_get **dynamic_get**
Pointer to function returning dynamic commands.

const struct *shell_static_entry* ***entry**
Pointer to array of static commands.

struct **shell_static_args**
#include <*shell.h*>

Public Members

uint8_t **mandatory**
Number of mandatory arguments.

uint8_t **optional**
Number of optional arguments.

struct **shell_static_entry**
#include <*shell.h*>

Public Members

const char ***syntax**
Command syntax strings.

const char ***help**
Command help string.

const union *shell_cmd_entry* ***subcmd**
Pointer to subcommand.

shell_cmd_handler **handler**
Command handler.

struct *shell_static_args* **args**
Command arguments.

struct **shell_transport_api**
#include <*shell.h*> Unified shell transport interface.

Public Members

int (*`init`)(const struct *shell_transport* *transport, const void *config, *shell_transport_handler_t* evt_handler, void *context)

Function for initializing the shell transport interface.

Param transport
[in] Pointer to the transfer instance.
Param config
[in] Pointer to instance configuration.
Param evt_handler
[in] Event handler.
Param context
[in] Pointer to the context passed to event handler.
Return
Standard error code.

int (*`uninit`)(const struct *shell_transport* *transport)

Function for uninitializing the shell transport interface.

Param transport
[in] Pointer to the transfer instance.
Return
Standard error code.

int (*`enable`)(const struct *shell_transport* *transport, bool blocking_tx)

Function for enabling transport in given TX mode.

Function can be used to reconfigure TX to work in blocking mode.

Param transport
Pointer to the transfer instance.
Param blocking_tx
If true, the transport TX is enabled in blocking mode.
Return
NRF_SUCCESS on successful enabling, error otherwise (also if not supported).

int (*`write`)(const struct *shell_transport* *transport, const void *data, size_t length, size_t *cnt)

Function for writing data to the transport interface.

Param transport
[in] Pointer to the transfer instance.
Param data
[in] Pointer to the source buffer.
Param length
[in] Source buffer length.
Param cnt
[out] Pointer to the sent bytes counter.
Return
Standard error code.

int (*`read`)(const struct *shell_transport* *transport, void *data, size_t length, size_t *cnt)

Function for reading data from the transport interface.

Param transport
[in] Pointer to the transfer instance.
Param data
[in] Pointer to the destination buffer.
Param length
[in] Destination buffer length.

Param cnt
[out] Pointer to the received bytes counter.
Return
Standard error code.

void (*update)(const struct *shell_transport* *transport)

Function called in shell thread loop.

Can be used for backend operations that require longer execution time
Param transport
[in] Pointer to the transfer instance.

struct shell_transport

#include <*shell.h*>

struct shell_stats

#include <*shell.h*> Shell statistics structure.

Public Members

atomic_t log_lost_cnt

Lost log counter.

struct shell_backend_config_flags

#include <*shell.h*>

Public Members

uint32_t insert_mode

Controls insert mode for text introduction.

uint32_t echo

Controls shell echo.

uint32_t obscure

If echo on, print asterisk instead.

uint32_t mode_delete

Operation mode of backspace key.

uint32_t use_colors

Controls colored syntax.

uint32_t use_vt100

Controls VT100 commands usage in shell.

struct shell_backend_ctx_flags

#include <*shell.h*>

Public Members

uint32_t **processing**
Shell is executing process function.

uint32_t **history_exit**
Request to exit history mode.

uint32_t **last_nl**
Last received new line character.

uint32_t **cmd_ctx**
Shell is executing command.

uint32_t **print_noinit**
Print request from not initialized shell.

uint32_t **sync_mode**
Shell in synchronous mode.

uint32_t **handle_log**
Shell is handling logger backend.

union **shell_backend_cfg**
#include <*shell.h*>

Public Members

atomic_t **value**

struct *shell_backend_config_flags* **flags**

union **shell_backend_ctx**
#include <*shell.h*>

Public Members

uint32_t **value**

struct *shell_backend_ctx_flags* **flags**

struct **shell_ctx**
#include <*shell.h*> Shell instance context.

Public Members

enum *shell_state* **state**

Internal module state.

enum *shell_receive_state* **receive_state**

Escape sequence indicator.

struct *shell_static_entry* **active_cmd**

Currently executed command.

const struct *shell_static_entry* ***selected_cmd**

New root command.

If NULL shell uses default root commands.

struct shell_vt100_ctx **vt100_ctx**

VT100 color and cursor position, terminal width.

shell_uninit_cb_t **uninit_cb**

Callback called from shell thread context when unitialization is completed just before aborting shell thread.

shell_bypass_cb_t **bypass**

When bypass is set, all incoming data is passed to the callback.

Logging level for a backend.

uint16_t **cmd_buff_len**

Command length.

uint16_t **cmd_buff_pos**

Command buffer cursor position.

uint16_t **cmd_tmp_buff_len**

Command length in tmp buffer.

char **cmd_buff**[0]

Command input buffer.

char **temp_buff**[0]

Command temporary buffer.

char **printf_buff**[0]

Printf buffer size.

struct *k_poll_event* **events**[*SHELL_SIGNALS*]

Events that should be used only internally by shell thread.

Event for SHELL_SIGNAL_TXDONE is initialized but unused.

struct **shell**

#include <shell.h> Shell instance internals.

Public Members

const char ***default_prompt**

shell default prompt.

const struct *shell_transport* ***iface**

Transport interface.

struct *shell_ctx* ***ctx**

Internal context.

4.23 Serialization

Zephyr has support for several data serialization subsystems. These can be used to encode/decode structured data with a known format on-the-wire.

4.23.1 Nanopb

Nanopb is a C implementation of Google's Protocol Buffers.

Requirements

Nanopb uses the protocol buffer compiler to generate source and header files, make sure the `protoc` executable is installed and available.

Ubuntu

Use `apt` to install dependency:

```
sudo apt install protobuf-compiler
```

macOS

Use `brew` to install dependency:

```
brew install protobuf
```

Windows

Use `choco` to install dependency:

```
choco install protoc
```

Additionally, Nanopb is an optional module and needs to be added explicitly to the workspace:

```
west config manifest.project-filter -- +nanopb
west update
```

Configuration

Make sure to include nanopb within your CMakeLists.txt file as follows:

```
list(APPEND CMAKE_MODULE_PATH ${ZEPHYR_BASE}/modules/nanopb)
include(nanopb)
```

Adding proto files can be done with the zephyr_nanopb_sources() CMake function which ensures the generated header and source files are created before building the specified target.

Nanopb has generator options that can be used to configure messages or fields. This allows to set fixed sizes or skip fields entirely.

The internal CMake generator has an extension to configure *.options.in files automatically with CMake variables.

See samples/modules/nanopb/src/simple.options.in and samples/modules/nanopb/CMakeLists.txt for usage example.

4.24 Settings

The settings subsystem gives modules a way to store persistent per-device configuration and runtime state. A variety of storage implementations are provided behind a common API using FCB, NVS, or a file system. These different implementations give the application developer flexibility to select an appropriate storage medium, and even change it later as needs change. This subsystem is used by various Zephyr components and can be used simultaneously by user applications.

Settings items are stored as key-value pair strings. By convention, the keys can be organized by the package and subtree defining the key, for example the key id/serial would define the serial configuration element for the package id.

Convenience routines are provided for converting a key value to and from a string type.

For an example of the settings subsystem refer to settings sample.

Note

As of Zephyr release 2.1 the recommended backend for non-filesystem storage is *NVS*.

4.24.1 Handlers

Settings handlers for subtree implement a set of handler functions. These are registered using a call to settings_register().

h_get
: This gets called when asking for a settings element value by its name using settings_runtime_get() from the runtime backend.

h_set
: This gets called when the value is loaded from persisted storage with settings_load(), or when using settings_runtime_set() from the runtime backend.

h_commit
: This gets called after the settings have been loaded in full. Sometimes you don't want an individual setting value to take effect right away, for example if there are multiple settings which are interdependent.

h_export
: This gets called to write all current settings. This happens when `settings_save()` tries to save the settings or transfer to any user-implemented back-end.

4.24.2 Backends

Backends are meant to load and save data to/from setting handlers, and implement a set of handler functions. These are registered using a call to `settings_src_register()` for backends that can load data, and/or `settings_dst_register()` for backends that can save data. The current implementation allows for multiple source backends but only a single destination backend.

csi_load
: This gets called when loading values from persistent storage using `settings_load()`.

csi_save
: This gets called when saving a single setting to persistent storage using `settings_save_one()`.

csi_save_start
: This gets called when starting a save of all current settings using `settings_save()`.

csi_save_end
: This gets called after having saved of all current settings using `settings_save()`.

4.24.3 Zephyr Storage Backends

Zephyr has three storage backends: a Flash Circular Buffer (`CONFIG_SETTINGS_FCB`), a file in the filesystem (`CONFIG_SETTINGS_FILE`), or non-volatile storage (`CONFIG_SETTINGS_NVS`).

You can declare multiple sources for settings; settings from all of these are restored when `settings_load()` is called.

There can be only one target for writing settings; this is where data is stored when you call `settings_save()`, or `settings_save_one()`.

FCB read target is registered using `settings_fcb_src()`, and write target using `settings_fcb_dst()`. As a side-effect, `settings_fcb_src()` initializes the FCB area, so it must be called before calling `settings_fcb_dst()`. File read target is registered using `settings_file_src()`, and write target by using `settings_file_dst()`. Non-volatile storage read target is registered using `settings_nvs_src()`, and write target by using `settings_nvs_dst()`.

4.24.4 Storage Location

The FCB and non-volatile storage (NVS) backends both look for a fixed partition with label "storage" by default. A different partition can be selected by setting the `zephyr,settings-partition` property of the chosen node in the devicetree.

The file path used by the file backend to store settings is selected via the option `CONFIG_SETTINGS_FILE_PATH`.

4.24.5 Loading data from persisted storage

A call to `settings_load()` uses an `h_set` implementation to load settings data from storage to volatile memory. After all data is loaded, the `h_commit` handler is issued, signalling the application that the settings were successfully retrieved.

Technically FCB and file backends may store some history of the entities. This means that the newest data entity is stored after any older existing data entities. Starting with Zephyr 2.1, the back-end must filter out all old entities and call the callback with only the newest entity.

4.24.6 Storing data to persistent storage

A call to `settings_save_one()` uses a backend implementation to store settings data to the storage medium. A call to `settings_save()` uses an `h_export` implementation to store different data in one operation using `settings_save_one()`. A key need to be covered by a `h_export` only if it is supposed to be stored by `settings_save()` call.

For both FCB and file back-end only storage requests with data which changes most actual key's value are stored, therefore there is no need to check whether a value changed by the application. Such a storage mechanism implies that storage can contain multiple value assignments for a key , while only the last is the current value for the key.

Garbage collection

When storage becomes full (FCB) or consumes too much space (file), the backend removes non-recent key-value pairs records and unnecessary key-delete records.

4.24.7 Secure domain settings

Currently settings doesn't provide scheme of being secure, and non-secure configuration storage simultaneously for the same instance. It is recommended that secure domain uses its own settings instance and it might provide data for non-secure domain using dedicated interface if needed (case dependent).

4.24.8 Example: Device Configuration

This is a simple example, where the settings handler only implements `h_set` and `h_export`. `h_set` is called when the value is restored from storage (or when set initially), and `h_export` is used to write the value to storage thanks to `storage_func()`. The user can also implement some other export functionality, for example, writing to the shell console).

```
#define DEFAULT_FOO_VAL_VALUE 1

static int8 foo_val = DEFAULT_FOO_VAL_VALUE;

static int foo_settings_set(const char *name, size_t len,
                            settings_read_cb read_cb, void *cb_arg)
{
    const char *next;
    int rc;

    if (settings_name_steq(name, "bar", &next) && !next) {
        if (len != sizeof(foo_val)) {
            return -EINVAL;
        }

        rc = read_cb(cb_arg, &foo_val, sizeof(foo_val));
        if (rc >= 0) {
            /* key-value pair was properly read.
             * rc contains value length.
             */
```

(continues on next page)

(continued from previous page)

```
            return 0;
        }
        /* read-out error */
        return rc;
    }

    return -ENOENT;
}

static int foo_settings_export(int (*storage_func)(const char *name,
                                                   const void *value,
                                                   size_t val_len))
{
    return storage_func("foo/bar", &foo_val, sizeof(foo_val));
}

struct settings_handler my_conf = {
    .name = "foo",
    .h_set = foo_settings_set,
    .h_export = foo_settings_export
};
```

4.24.9 Example: Persist Runtime State

This is a simple example showing how to persist runtime state. In this example, only `h_set` is defined, which is used when restoring value from persisted storage.

In this example, the `main` function increments `foo_val`, and then persists the latest number. When the system restarts, the application calls `settings_load()` while initializing, and `foo_val` will continue counting up from where it was before restart.

```
#include <zephyr/kernel.h>
#include <zephyr/sys/reboot.h>
#include <zephyr/settings/settings.h>
#include <zephyr/sys/printk.h>
#include <inttypes.h>

#define DEFAULT_FOO_VAL_VALUE 0

static uint8_t foo_val = DEFAULT_FOO_VAL_VALUE;

static int foo_settings_set(const char *name, size_t len,
                            settings_read_cb read_cb, void *cb_arg)
{
    const char *next;
    int rc;

    if (settings_name_steq(name, "bar", &next) && !next) {
        if (len != sizeof(foo_val)) {
            return -EINVAL;
        }

        rc = read_cb(cb_arg, &foo_val, sizeof(foo_val));
        if (rc >= 0) {
            return 0;
        }

        return rc;
    }
```

(continues on next page)

(continued from previous page)

```
    return -ENOENT;
}

struct settings_handler my_conf = {
    .name = "foo",
    .h_set = foo_settings_set
};

int main(void)
{
    settings_subsys_init();
    settings_register(&my_conf);
    settings_load();

    foo_val++;
    settings_save_one("foo/bar", &foo_val, sizeof(foo_val));

    printk("foo: %d\n", foo_val);

    k_msleep(1000);
    sys_reboot(SYS_REBOOT_COLD);
}
```

4.24.10 Example: Custom Backend Implementation

This is a simple example showing how to register a simple custom backend handler (CONFIG_SETTINGS_CUSTOM).

```
static int settings_custom_load(struct settings_store *cs,
                                const struct settings_load_arg *arg)
{
    //...
}

static int settings_custom_save(struct settings_store *cs, const char *name,
                                const char *value, size_t val_len)
{
    //...
}

/* custom backend interface */
static struct settings_store_itf settings_custom_itf = {
    .csi_load = settings_custom_load,
    .csi_save = settings_custom_save,
};

/* custom backend node */
static struct settings_store settings_custom_store = {
    .cs_itf = &settings_custom_itf
};

int settings_backend_init(void)
{
    /* register custom backend */
    settings_dst_register(&settings_custom_store);
    settings_src_register(&settings_custom_store);
    return 0;
```

(continues on next page)

(continued from previous page)

```
}
```

4.24.11 API Reference

The Settings subsystem APIs are provided by `settings.h`:

API for general settings usage

Related code samples

Settings API
Load and save configuration values using the settings API.

group `settings`

Since
1.12

Version
1.0.0

Defines

`SETTINGS_MAX_DIR_DEPTH`

`SETTINGS_MAX_NAME_LEN`

`SETTINGS_MAX_VAL_LEN`

`SETTINGS_NAME_SEPARATOR`

`SETTINGS_NAME_END`

`SETTINGS_EXTRA_LEN`

`SETTINGS_STATIC_HANDLER_DEFINE`(_hname, _tree, _get, _set, _commit, _export)
Define a static handler for settings items.

This creates a variable *hname prepended by* settings_handler.

Parameters

- **`_hname`** – handler name
- **`_tree`** – subtree name
- **`_get`** – get routine (can be NULL)
- **`_set`** – set routine (can be NULL)

- `_commit` – commit routine (can be NULL)
- `_export` – export routine (can be NULL)

Typedefs

typedef ssize_t (*`settings_read_cb`)(void *cb_arg, void *data, size_t len)

Function used to read the data from the settings storage in h_set handler implementations.

Param cb_arg
[in] arguments for the read function. Appropriate cb_arg is transferred to h_set handler implementation by the backend.

Param data
[out] the destination buffer

Param len
[in] length of read

Return
positive: Number of bytes read, 0: key-value pair is deleted. On error returns -ERRNO code.

typedef int (*`settings_load_direct_cb`)(const char *key, size_t len, *settings_read_cb* read_cb, void *cb_arg, void *param)

Callback function used for direct loading.

Used by *settings_load_subtree_direct* function.

Param key
[in] the name with skipped part that was used as name in handler registration

Param len
[in] the size of the data found in the backend.

Param read_cb
[in] function provided to read the data from the backend.

Param cb_arg
[inout] arguments for the read function provided by the backend.

Param param
[inout] parameter given to the *settings_load_subtree_direct* function.

Return
When nonzero value is returned, further subtree searching is stopped.

Functions

int `settings_subsys_init`(void)

Initialization of settings and backend.

Can be called at application startup. In case the backend is a FS Remember to call it after the FS was mounted. For FCB backend it can be called without such a restriction.

Returns
0 on success, non-zero on failure.

int `settings_register`(struct *settings_handler* *cf)

Register a handler for settings items stored in RAM.

Parameters

- `cf` – Structure containing registration info.

Returns

0 on success, non-zero on failure.

int `settings_load`(void)

Load serialized items from registered persistence sources.

Handlers for serialized item subtrees registered earlier will be called for encountered values.

Returns

0 on success, non-zero on failure.

int `settings_load_subtree`(const char *subtree)

Load limited set of serialized items from registered persistence sources.

Handlers for serialized item subtrees registered earlier will be called for encountered values that belong to the subtree.

Parameters

- `subtree` – **[in]** name of the subtree to be loaded.

Returns

0 on success, non-zero on failure.

int `settings_load_subtree_direct`(const char *subtree, *settings_load_direct_cb* cb, void *param)

Load limited set of serialized items using given callback.

This function bypasses the normal data workflow in settings module. All the settings values that are found are passed to the given callback.

> **Note**
>
> This function does not call commit function. It works as a blocking function, so it is up to the user to call any kind of commit function when this operation ends.

Parameters

- `subtree` – **[in]** subtree name of the subtree to be loaded.
- `cb` – **[in]** pointer to the callback function.
- `param` – **[inout]** parameter to be passed when callback function is called.

Returns

0 on success, non-zero on failure.

int `settings_save`(void)

Save currently running serialized items.

All serialized items which are different from currently persisted values will be saved.

Returns

0 on success, non-zero on failure.

int `settings_save_subtree`(const char *subtree)

Save limited set of currently running serialized items.

All serialized items that belong to subtree and which are different from currently persisted values will be saved.

Parameters

- `subtree` – **[in]** name of the subtree to be loaded.

Returns

0 on success, non-zero on failure.

int `settings_save_one`(const char *name, const void *value, size_t val_len)

Write a single serialized value to persisted storage (if it has changed value).

Parameters

- `name` – Name/key of the settings item.
- `value` – Pointer to the value of the settings item. This value will be transferred to the *settings_handler::h_export* handler implementation.
- `val_len` – Length of the value.

Returns

0 on success, non-zero on failure.

int `settings_delete`(const char *name)

Delete a single serialized in persisted storage.

Deleting an existing key-value pair in the settings mean to set its value to NULL.

Parameters

- `name` – Name/key of the settings item.

Returns

0 on success, non-zero on failure.

int `settings_commit`(void)

Call commit for all settings handler.

This should apply all settings which has been set, but not applied yet.

Returns

0 on success, non-zero on failure.

int `settings_commit_subtree`(const char *subtree)

Call commit for settings handler that belong to subtree.

This should apply all settings which has been set, but not applied yet.

Parameters

- `subtree` – **[in]** name of the subtree to be committed.

Returns

0 on success, non-zero on failure.

struct `settings_handler`

#include <*settings.h*> Config handlers for subtree implement a set of handler functions.

These are registered using a call to *settings_register*.

Public Members

const char *name

Name of subtree.

int (*h_get)(const char *key, char *val, int val_len_max)

Get values handler of settings items identified by keyword names.

Parameters:
- key[in] the name with skipped part that was used as name in handler registration
- val[out] buffer to receive value.
- val_len_max[in] size of that buffer.

Return: length of data read on success, negative on failure.

int (*h_set)(const char *key, size_t len, *settings_read_cb* read_cb, void *cb_arg)

Set value handler of settings items identified by keyword names.

Parameters:
- key[in] the name with skipped part that was used as name in handler registration
- len[in] the size of the data found in the backend.
- read_cb[in] function provided to read the data from the backend.
- cb_arg[in] arguments for the read function provided by the backend.

Return: 0 on success, non-zero on failure.

int (*h_commit)(void)

This handler gets called after settings has been loaded in full.

User might use it to apply setting to the application.

Return: 0 on success, non-zero on failure.

int (*h_export)(int (*export_func)(const char *name, const void *val, size_t val_len))

This gets called to dump all current settings items.

This happens when *settings_save* tries to save the settings. Parameters:
- export_func: the pointer to the internal function which appends a single key-value pair to persisted settings. Don't store duplicated value. The name is subtree/key string, val is the string with value.

Return: 0 on success, non-zero on failure.

> **Remark**
>
> The User might limit a implementations of handler to serving only one keyword at one call - what will impose limit to get/set values using full subtree/key name.

sys_snode_t node

Linked list node info for module internal usage.

struct settings_handler_static

#include <*settings.h*> Config handlers without the node element, used for static handlers.

These are registered using a call to *SETTINGS_STATIC_HANDLER_DEFINE()*.

Public Members

const char *`name`

Name of subtree.

int (*`h_get`)(const char *key, char *val, int val_len_max)

Get values handler of settings items identified by keyword names.

Parameters:
- key[in] the name with skipped part that was used as name in handler registration
- val[out] buffer to receive value.
- val_len_max[in] size of that buffer.

Return: length of data read on success, negative on failure.

int (*`h_set`)(const char *key, size_t len, *settings_read_cb* read_cb, void *cb_arg)

Set value handler of settings items identified by keyword names.

Parameters:
- key[in] the name with skipped part that was used as name in handler registration
- len[in] the size of the data found in the backend.
- read_cb[in] function provided to read the data from the backend.
- cb_arg[in] arguments for the read function provided by the backend.

Return: 0 on success, non-zero on failure.

int (*`h_commit`)(void)

This handler gets called after settings has been loaded in full.

User might use it to apply setting to the application.

int (*`h_export`)(int (*export_func)(const char *name, const void *val, size_t val_len))

This gets called to dump all current settings items.

This happens when *settings_save* tries to save the settings. Parameters:
- export_func: the pointer to the internal function which appends a single key-value pair to persisted settings. Don't store duplicated value. The name is subtree/key string, val is the string with value.

Return: 0 on success, non-zero on failure.

Remark

The User might limit a implementations of handler to serving only one keyword at one call - what will impose limit to get/set values using full subtree/key name.

API for key-name processing

Related code samples

Settings API
Load and save configuration values using the settings API.

group `settings_name_proc`

API for const name processing.

Functions

int `settings_name_steq`(const char *name, const char *key, const char **next)

Compares the start of name with a key.

Some examples: settings_name_steq("bt/btmesh/iv", "b", &next) returns 1, next="t/btmesh/iv" settings_name_steq("bt/btmesh/iv", "bt", &next) returns 1, next="btmesh/iv" settings_name_steq("bt/btmesh/iv", "bt/", &next) returns 0, next=NULL settings_name_steq("bt/btmesh/iv", "bta", &next) returns 0, next=NULL

REMARK: This routine could be simplified if the *settings_handler* names would include a separator at the end.

Parameters

- `name` – **[in]** in string format
- `key` – **[in]** comparison string
- `next` – **[out]** pointer to remaining of name, when the remaining part starts with a separator the separator is removed from next

Returns
0: no match 1: match, next can be used to check if match is full

int `settings_name_next`(const char *name, const char **next)

determine the number of characters before the first separator

Parameters

- `name` – **[in]** in string format
- `next` – **[out]** pointer to remaining of name (excluding separator)

Returns
index of the first separator, in case no separator was found this is the size of name

API for runtime settings manipulation

Related code samples

Settings API
Load and save configuration values using the settings API.

group `settings_rt`

API for runtime settings.

Functions

int **settings_runtime_set**(const char *name, const void *data, size_t len)

Set a value with a specific key to a module handler.

Parameters

- **name** – Key in string format.
- **data** – Binary value.
- **len** – Value length in bytes.

Returns

0 on success, non-zero on failure.

int **settings_runtime_get**(const char *name, void *data, size_t len)

Get a value corresponding to a key from a module handler.

Parameters

- **name** – Key in string format.
- **data** – Returned binary value.
- **len** – requested value length in bytes.

Returns

length of data read on success, negative on failure.

int **settings_runtime_commit**(const char *name)

Apply settings in a module handler.

Parameters

- **name** – Key in string format.

Returns

0 on success, non-zero on failure.

API of backend interface

group **settings_backend**

settings

Functions

void **settings_src_register**(struct *settings_store* *cs)

Register a backend handler acting as source.

Parameters

- **cs** – Backend handler node containing handler information.

void **settings_dst_register**(struct *settings_store* *cs)

Register a backend handler acting as destination.

Parameters

- **cs** – Backend handler node containing handler information.

struct settings_handler_ *settings_parse_and_lookup(const char *name, const char **next)

Parses a key to an array of elements and locate corresponding module handler.

Parameters

- name – **[in]** in string format
- next – **[out]** remaining of name after matched handler

Returns
settings_handler_static on success, NULL on failure.

int settings_call_set_handler(const char *name, size_t len, *settings_read_cb* read_cb, void *read_cb_arg, const struct *settings_load_arg* *load_arg)

Calls settings handler.

Parameters

- name – **[in]** The name of the data found in the backend.
- len – **[in]** The size of the data found in the backend.
- read_cb – **[in]** Function provided to read the data from the backend.
- read_cb_arg – **[inout]** Arguments for the read function provided by the backend.
- load_arg – **[inout]** Arguments for data loading.

Returns
0 or negative error code

struct settings_store

#include <settings.h> Backend handler node for storage handling.

Public Members

sys_snode_t cs_next

Linked list node info for internal usage.

const struct *settings_store_itf* *cs_itf

Backend handler structure.

struct settings_load_arg

#include <settings.h> Arguments for data loading.

Holds all parameters that changes the way data should be loaded from backend.

Public Members

const char *subtree

Name of the subtree to be loaded.

If NULL, all values would be loaded.

settings_load_direct_cb **cb**

Pointer to the callback function.

If NULL then matching registered function would be used.

void ***param**

Parameter for callback function.

Parameter to be passed to the callback function.

struct **settings_store_itf**

#include <*settings.h*> Backend handler functions.

Sources are registered using a call to *settings_src_register*. Destinations are registered using a call to *settings_dst_register*.

Public Members

int (***csi_load**)(struct *settings_store* *cs, const struct *settings_load_arg* *arg)

Loads values from storage limited to subtree defined by subtree.

Parameters:
- cs - Corresponding backend handler node,
- arg - Structure that holds additional data for data loading.

Note

Backend is expected not to provide duplicates of the entities. It means that if the backend does not contain any functionality to really delete old keys, it has to filter out old entities and call load callback only on the final entity.

int (***csi_save_start**)(struct *settings_store* *cs)

Handler called before an export operation.

Parameters:
- cs - Corresponding backend handler node

int (***csi_save**)(struct *settings_store* *cs, const char *name, const char *value, size_t val_len)

Save a single key-value pair to storage.

Parameters:
- cs - Corresponding backend handler node
- name - Key in string format
- value - Binary value
- val_len - Length of value in bytes.

int (***csi_save_end**)(struct *settings_store* *cs)

Handler called after an export operation.

Parameters:
- cs - Corresponding backend handler node Get pointer to the storage instance used by the backend.

Parameters:
- cs - Corresponding backend handler node

4.25 State Machine Framework

4.25.1 Overview

The State Machine Framework (SMF) is an application agnostic framework that provides an easy way for developers to integrate state machines into their application. The framework can be added to any project by enabling the `CONFIG_SMF` option.

4.25.2 State Creation

A state is represented by three functions, where one function implements the Entry actions, another function implements the Run actions, and the last function implements the Exit actions. The prototype for these functions is as follows: `void funct(void *obj)`, where the `obj` parameter is a user defined structure that has the state machine context, *smf_ctx*, as its first member. For example:

```
struct user_object {
   struct smf_ctx ctx;
   /* All User Defined Data Follows */
};
```

The *smf_ctx* member must be first because the state machine framework's functions casts the user defined object to the *smf_ctx* type with the *SMF_CTX* macro.

For example instead of doing this `(struct smf_ctx *)&user_obj`, you could use `SMF_CTX(&user_obj)`.

By default, a state can have no ancestor states, resulting in a flat state machine. But to enable the creation of a hierarchical state machine, the `CONFIG_SMF_ANCESTOR_SUPPORT` option must be enabled.

By default, the hierarchical state machines do not support initial transitions to child states on entering a superstate. To enable them the `CONFIG_SMF_INITIAL_TRANSITION` option must be enabled.

The following macro can be used for easy state creation:

- *SMF_CREATE_STATE* Create a state

4.25.3 State Machine Creation

A state machine is created by defining a table of states that's indexed by an enum. For example, the following creates three flat states:

```
enum demo_state { S0, S1, S2 };

const struct smf_state demo_states[] = {
   [S0] = SMF_CREATE_STATE(s0_entry, s0_run, s0_exit, NULL, NULL),
   [S1] = SMF_CREATE_STATE(s1_entry, s1_run, s1_exit, NULL, NULL),
   [S2] = SMF_CREATE_STATE(s2_entry, s2_run, s2_exit, NULL, NULL)
};
```

And this example creates three hierarchical states:

```
enum demo_state { S0, S1, S2 };

const struct smf_state demo_states[] = {
   [S0] = SMF_CREATE_STATE(s0_entry, s0_run, s0_exit, parent_s0, NULL),
```

(continues on next page)

(continued from previous page)

```
   [S1] = SMF_CREATE_STATE(s1_entry, s1_run, s1_exit, parent_s12, NULL),
   [S2] = SMF_CREATE_STATE(s2_entry, s2_run, s2_exit, parent_s12, NULL)
};
```

This example creates three hierarchical states with an initial transition from parent state S0 to child state S2:

```
enum demo_state { S0, S1, S2 };

/* Forward declaration of state table */
const struct smf_state demo_states[];

const struct smf_state demo_states[] = {
   [S0] = SMF_CREATE_STATE(s0_entry, s0_run, s0_exit, NULL, demo_states[S2]),
   [S1] = SMF_CREATE_STATE(s1_entry, s1_run, s1_exit, demo_states[S0], NULL),
   [S2] = SMF_CREATE_STATE(s2_entry, s2_run, s2_exit, demo_states[S0], NULL)
};
```

To set the initial state, the *smf_set_initial()* function should be called.

To transition from one state to another, the *smf_set_state()* function is used.

Note

If CONFIG_SMF_INITIAL_TRANSITION is not set, *smf_set_initial()* and *smf_set_state()* function should not be passed a parent state as the parent state does not know which child state to transition to. Transitioning to a parent state is OK if an initial transition to a child state is defined. A well-formed HSM should have initial transitions defined for all parent states.

Note

While the state machine is running, *smf_set_state()* should only be called from the Entry or Run function. Calling *smf_set_state()* from Exit functions will generate a warning in the log and no transition will occur.

4.25.4 State Machine Execution

To run the state machine, the *smf_run_state()* function should be called in some application dependent way. An application should cease calling smf_run_state if it returns a non-zero value.

4.25.5 Preventing Parent Run Actions

Calling *smf_set_handled()* prevents calling the run action of parent states. It is not required to call *smf_set_handled()* if the state calls *smf_set_state()*.

4.25.6 State Machine Termination

To terminate the state machine, the *smf_set_terminate()* function should be called. It can be called from the entry, run, or exit actions. The function takes a non-zero user defined value that will be returned by the *smf_run_state()* function.

4.25.7 UML State Machines

SMF follows UML hierarchical state machine rules for transitions i.e., the entry and exit actions of the least common ancestor are not executed on transition, unless said transition is a transition to self.

The UML Specification for StateMachines may be found in chapter 14 of the UML specification available here: https://www.omg.org/spec/UML/

SMF breaks from UML rules in:

1. Executing the actions associated with the transition within the context of the source state, rather than after the exit actions are performed.
2. Only allowing external transitions to self, not to sub-states. A transition from a superstate to a child state is treated as a local transition.
3. Prohibiting transitions using *smf_set_state()* in exit actions.

SMF also does not provide any pseudostates except the Initial Pseudostate. Terminate pseudostates can be modelled by calling *smf_set_terminate()* from the entry action of a 'terminate' state. Orthogonal regions are modelled by calling *smf_run_state()* for each region.

4.25.8 State Machine Examples

Flat State Machine Example

This example turns the following state diagram into code using the SMF, where the initial state is S0.

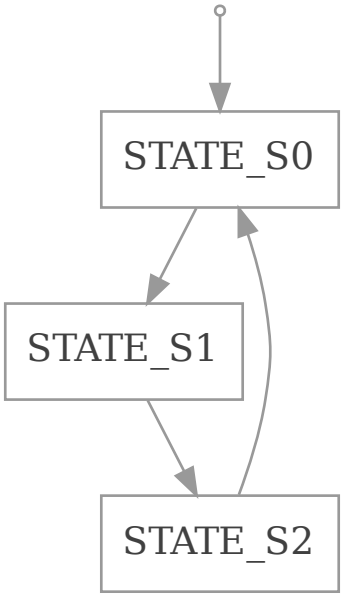


Fig. 21: Flat state machine diagram

Code:

```c
#include <zephyr/smf.h>

/* Forward declaration of state table */
static const struct smf_state demo_states[];

/* List of demo states */
enum demo_state { S0, S1, S2 };

/* User defined object */
struct s_object {
        /* This must be first */
        struct smf_ctx ctx;

        /* Other state specific data add here */
} s_obj;

/* State S0 */
static void s0_entry(void *o)
{
        /* Do something */
}
static void s0_run(void *o)
{
        smf_set_state(SMF_CTX(&s_obj), &demo_states[S1]);
}
static void s0_exit(void *o)
{
        /* Do something */
}

/* State S1 */
static void s1_run(void *o)
{
        smf_set_state(SMF_CTX(&s_obj), &demo_states[S2]);
}
static void s1_exit(void *o)
{
        /* Do something */
}

/* State S2 */
static void s2_entry(void *o)
{
        /* Do something */
}
static void s2_run(void *o)
{
        smf_set_state(SMF_CTX(&s_obj), &demo_states[S0]);
}

/* Populate state table */
static const struct smf_state demo_states[] = {
        [S0] = SMF_CREATE_STATE(s0_entry, s0_run, s0_exit, NULL, NULL),
        /* State S1 does not have an entry action */
        [S1] = SMF_CREATE_STATE(NULL, s1_run, s1_exit, NULL, NULL),
        /* State S2 does not have an exit action */
        [S2] = SMF_CREATE_STATE(s2_entry, s2_run, NULL, NULL, NULL),
};

int main(void)
{
        int32_t ret;
```

(continues on next page)

(continued from previous page)

```
        /* Set initial state */
        smf_set_initial(SMF_CTX(&s_obj), &demo_states[S0]);

        /* Run the state machine */
        while(1) {
                /* State machine terminates if a non-zero value is returned */
                ret = smf_run_state(SMF_CTX(&s_obj));
                if (ret) {
                        /* handle return code and terminate state machine */
                        break;
                }
                k_msleep(1000);
        }
}
```

Hierarchical State Machine Example

This example turns the following state diagram into code using the SMF, where S0 and S1 share a parent state and S0 is the initial state.

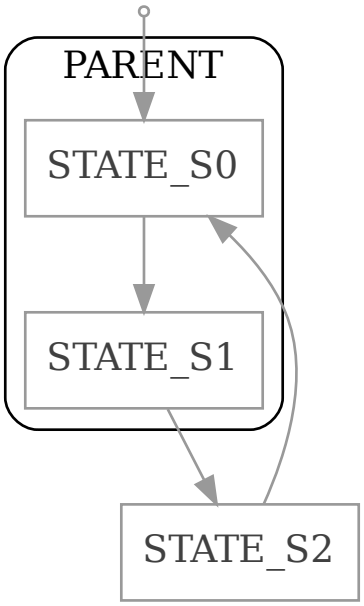


Fig. 22: Hierarchical state machine diagram

Code:

```
#include <zephyr/smf.h>

/* Forward declaration of state table */
static const struct smf_state demo_states[];
```

(continues on next page)

(continued from previous page)

```
/* List of demo states */
enum demo_state { PARENT, S0, S1, S2 };

/* User defined object */
struct s_object {
        /* This must be first */
        struct smf_ctx ctx;

        /* Other state specific data add here */
} s_obj;

/* Parent State */
static void parent_entry(void *o)
{
        /* Do something */
}
static void parent_exit(void *o)
{
        /* Do something */
}

/* State S0 */
static void s0_run(void *o)
{
        smf_set_state(SMF_CTX(&s_obj), &demo_states[S1]);
}

/* State S1 */
static void s1_run(void *o)
{
        smf_set_state(SMF_CTX(&s_obj), &demo_states[S2]);
}

/* State S2 */
static void s2_run(void *o)
{
        smf_set_state(SMF_CTX(&s_obj), &demo_states[S0]);
}

/* Populate state table */
static const struct smf_state demo_states[] = {
        /* Parent state does not have a run action */
        [PARENT] = SMF_CREATE_STATE(parent_entry, NULL, parent_exit, NULL, NULL),
        /* Child states do not have entry or exit actions */
        [S0] = SMF_CREATE_STATE(NULL, s0_run, NULL, &demo_states[PARENT], NULL),
        [S1] = SMF_CREATE_STATE(NULL, s1_run, NULL, &demo_states[PARENT], NULL),
        /* State S2 do ot have entry or exit actions and no parent */
        [S2] = SMF_CREATE_STATE(NULL, s2_run, NULL, NULL, NULL),
};

int main(void)
{
        int32_t ret;

        /* Set initial state */
        smf_set_initial(SMF_CTX(&s_obj), &demo_states[S0]);

        /* Run the state machine */
        while(1) {
                /* State machine terminates if a non-zero value is returned */
                ret = smf_run_state(SMF_CTX(&s_obj));
```

(continues on next page)

(continued from previous page)

```
                if (ret) {
                        /* handle return code and terminate state machine */
                        break;
                }
                k_msleep(1000);
        }
}
```

When designing hierarchical state machines, the following should be considered:

- Ancestor entry actions are executed before the sibling entry actions. For example, the parent_entry function is called before the s0_entry function.
- Transitioning from one sibling to another with a shared ancestry does not re-execute the ancestor's entry action or execute the exit action. For example, the parent_entry function is not called when transitioning from S0 to S1, nor is the parent_exit function called.
- Ancestor exit actions are executed after the exit action of the current state. For example, the s1_exit function is called before the parent_exit function is called.
- The parent_run function only executes if the child_run function does not call either *smf_set_state()* or *smf_set_handled()*.

Event Driven State Machine Example

Events are not explicitly part of the State Machine Framework but an event driven state machine can be implemented using Zephyr *Events*.

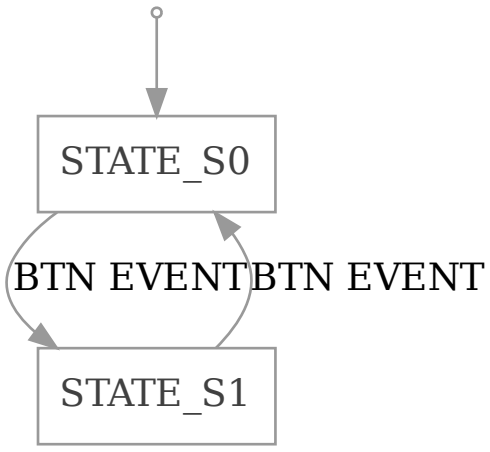


Fig. 23: Event driven state machine diagram

Code:

```
#include <zephyr/kernel.h>
#include <zephyr/drivers/gpio.h>
#include <zephyr/smf.h>
```

(continues on next page)

(continued from previous page)

```
#define SW0_NODE        DT_ALIAS(sw0)

/* List of events */
#define EVENT_BTN_PRESS BIT(0)

static const struct gpio_dt_spec button =
        GPIO_DT_SPEC_GET_OR(SW0_NODE, gpios, {0});

static struct gpio_callback button_cb_data;

/* Forward declaration of state table */
static const struct smf_state demo_states[];

/* List of demo states */
enum demo_state { S0, S1 };

/* User defined object */
struct s_object {
        /* This must be first */
        struct smf_ctx ctx;

        /* Events */
        struct k_event smf_event;
        int32_t events;

        /* Other state specific data add here */
} s_obj;

/* State S0 */
static void s0_entry(void *o)
{
        printk("STATE0\n");
}

static void s0_run(void *o)
{
        struct s_object *s = (struct s_object *)o;

        /* Change states on Button Press Event */
        if (s->events & EVENT_BTN_PRESS) {
                smf_set_state(SMF_CTX(&s_obj), &demo_states[S1]);
        }
}

/* State S1 */
static void s1_entry(void *o)
{
        printk("STATE1\n");
}

static void s1_run(void *o)
{
        struct s_object *s = (struct s_object *)o;

        /* Change states on Button Press Event */
        if (s->events & EVENT_BTN_PRESS) {
                smf_set_state(SMF_CTX(&s_obj), &demo_states[S0]);
        }
}

/* Populate state table */
```

(continues on next page)

(continued from previous page)

```
static const struct smf_state demo_states[] = {
        [S0] = SMF_CREATE_STATE(s0_entry, s0_run, NULL, NULL, NULL),
        [S1] = SMF_CREATE_STATE(s1_entry, s1_run, NULL, NULL, NULL),
};

void button_pressed(const struct device *dev,
                struct gpio_callback *cb, uint32_t pins)
{
        /* Generate Button Press Event */
        k_event_post(&s_obj.smf_event, EVENT_BTN_PRESS);
}

int main(void)
{
        int ret;

        if (!gpio_is_ready_dt(&button)) {
                printk("Error: button device %s is not ready\n",
                        button.port->name);
                return;
        }

        ret = gpio_pin_configure_dt(&button, GPIO_INPUT);
        if (ret != 0) {
                printk("Error %d: failed to configure %s pin %d\n",
                        ret, button.port->name, button.pin);
                return;
        }

        ret = gpio_pin_interrupt_configure_dt(&button,
                GPIO_INT_EDGE_TO_ACTIVE);
        if (ret != 0) {
                printk("Error %d: failed to configure interrupt on %s pin %d\n",
                        ret, button.port->name, button.pin);
                return;
        }

        gpio_init_callback(&button_cb_data, button_pressed, BIT(button.pin));
        gpio_add_callback(button.port, &button_cb_data);

        /* Initialize the event */
        k_event_init(&s_obj.smf_event);

        /* Set initial state */
        smf_set_initial(SMF_CTX(&s_obj), &demo_states[S0]);

        /* Run the state machine */
        while(1) {
                /* Block until an event is detected */
                s_obj.events = k_event_wait(&s_obj.smf_event,
                                EVENT_BTN_PRESS, true, K_FOREVER);

                /* State machine terminates if a non-zero value is returned */
                ret = smf_run_state(SMF_CTX(&s_obj));
                if (ret) {
                        /* handle return code and terminate state machine */
                        break;
                }
        }
}
```

State Machine Example With Initial Transitions And Transition To Self

tests/lib/smf/src/test_lib_self_transition_smf.c defines a state machine for testing the initial transitions and transitions to self in a parent state. The statechart for this test is below.

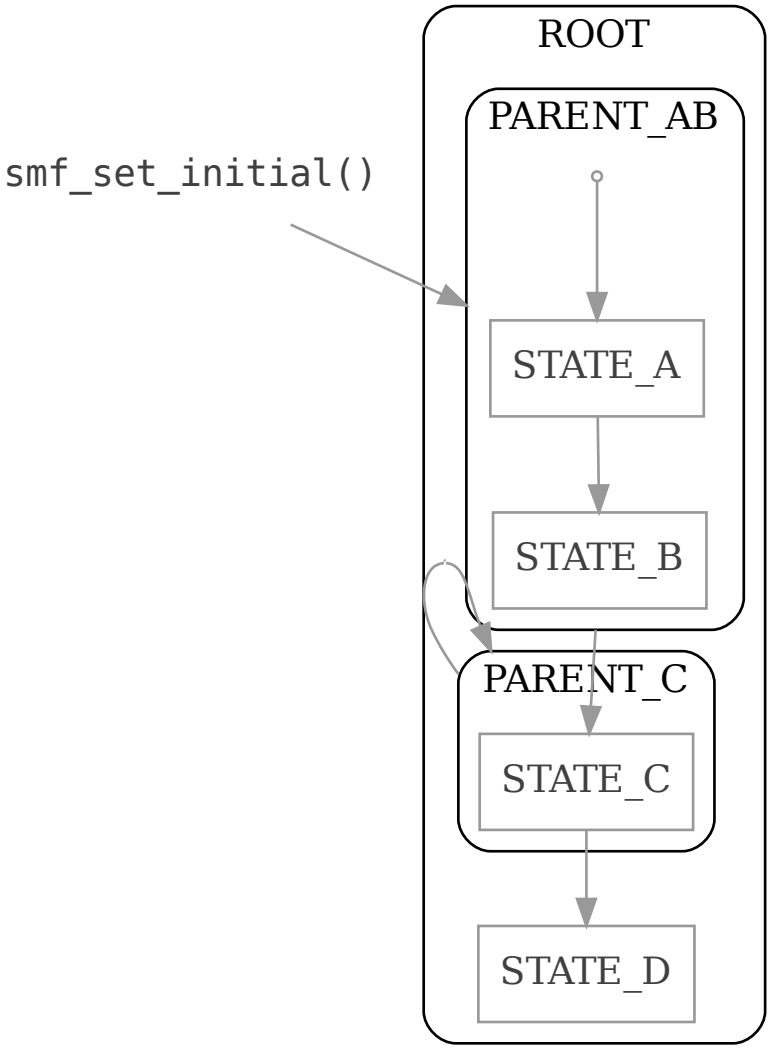


Fig. 24: Test state machine for UML State Transitions

4.25.9 API Reference

Related code samples

Hierarchical State Machine Demo based on example from PSiCC2
Implement an event-driven hierarchical state machine using State Machine Framework

(SMF).

group `smf`

State Machine Framework API.

Version
0.1.0

Defines

`SMF_CREATE_STATE`(_entry, _run, _exit, _parent, _initial)

Macro to create a hierarchical state with initial transitions.

Parameters

- `_entry` – State entry function or NULL
- `_run` – State run function or NULL
- `_exit` – State exit function or NULL
- `_parent` – State parent object or NULL
- `_initial` – State initial transition object or NULL

`SMF_CTX`(o)

Macro to cast user defined object to state machine context.

Parameters

- `o` – A pointer to the user defined object

Typedefs

typedef void (*`state_execution`)(void *obj)

Function pointer that implements a portion of a state.

Param obj
pointer user defined object

Functions

void `smf_set_initial`(struct *smf_ctx* *ctx, const struct *smf_state* *init_state)

Initializes the state machine and sets its initial state.

Parameters

- `ctx` – State machine context
- `init_state` – Initial state the state machine starts in.

void `smf_set_state`(struct *smf_ctx* *ctx, const struct *smf_state* *new_state)

Changes a state machines state.

This handles exiting the previous state and entering the target state. For HSMs the entry and exit actions of the Least Common Ancestor will not be run.

Parameters

- **ctx** – State machine context
- **new_state** – State to transition to (NULL is valid and exits all states)

void **smf_set_terminate**(struct *smf_ctx* *ctx, int32_t val)

Terminate a state machine.

Parameters

- **ctx** – State machine context
- **val** – Non-Zero termination value that's returned by the smf_run_state function.

void **smf_set_handled**(struct *smf_ctx* *ctx)

Tell the SMF to stop propagating the event to ancestors.

This allows HSMs to implement 'programming by difference' where substates can handle events on their own or propagate up to a common handler.

Parameters

- **ctx** – State machine context

int32_t **smf_run_state**(struct *smf_ctx* *ctx)

Runs one iteration of a state machine (including any parent states)

Parameters

- **ctx** – State machine context

Returns

A non-zero value should terminate the state machine. This non-zero value could represent a terminal state being reached or the detection of an error that should result in the termination of the state machine.

struct **smf_state**

#include <*smf.h*> General state that can be used in multiple state machines.

Public Members

const *state_execution* **entry**

Optional method that will be run when this state is entered.

const *state_execution* **run**

Optional method that will be run repeatedly during state machine loop.

const *state_execution* **exit**

Optional method that will be run when this state exists.

struct **smf_ctx**

#include <*smf.h*> Defines the current context of the state machine.

Public Members

const struct *smf_state* ***current**

Current state the state machine is executing.

const struct *smf_state* **previous**

Previous state the state machine executed.

int32_t **terminate_val**

This value is set by the set_terminate function and should terminate the state machine when its set to a value other than zero when it's returned by the run_state function.

uint32_t **internal**

The state machine casts this to a "struct internal_ctx" and it's used to track state machine context.

4.26 Storage

4.26.1 Non-Volatile Storage (NVS)

Elements, represented as id-data pairs, are stored in flash using a FIFO-managed circular buffer. The flash area is divided into sectors. Elements are appended to a sector until storage space in the sector is exhausted. Then a new sector in the flash area is prepared for use (erased). Before erasing the sector it is checked that identifier - data pairs exist in the sectors in use, if not the id-data pair is copied.

The id is a 16-bit unsigned number. NVS ensures that for each used id there is at least one id-data pair stored in flash at all time.

NVS allows storage of binary blobs, strings, integers, longs, and any combination of these.

Each element is stored in flash as metadata (8 byte) and data. The metadata is written in a table starting from the end of a nvs sector, the data is written one after the other from the start of the sector. The metadata consists of: id, data offset in sector, data length, part (unused), and a CRC. This CRC is only calculated over the metadata and only ensures that a write has been completed. The actual data of the element can be protected by a different (and optional) CRC-32. Use the `CONFIG_NVS_DATA_CRC` configuration item to enable the data part CRC.

Note

The data CRC is checked only when the whole data of the element is read. The data CRC is not checked for a partial read, as it is stored at the end of the element data area.

Note

Enabling the data CRC feature on a previously existing NVS content without data CRC will make all existing data invalid.

A write of data to nvs always starts with writing the data, followed by a write of the metadata. Data that is written in flash without metadata is ignored during initialization.

During initialization NVS will verify the data stored in flash, if it encounters an error it will ignore any data with missing/incorrect metadata.

NVS checks the id-data pair before writing data to flash. If the id-data pair is unchanged no write to flash is performed.

To protect the flash area against frequent erases it is important that there is sufficient free space. NVS has a protection mechanism to avoid getting in a endless loop of flash page erases when there is limited free space. When such a loop is detected NVS returns that there is no more space available.

For NVS the file system is declared as:

```
static struct nvs_fs fs = {
.flash_device = NVS_FLASH_DEVICE,
.sector_size = NVS_SECTOR_SIZE,
.sector_count = NVS_SECTOR_COUNT,
.offset = NVS_STORAGE_OFFSET,
};
```

where

- NVS_FLASH_DEVICE is a reference to the flash device that will be used. The device needs to be operational.
- NVS_SECTOR_SIZE is the sector size, it has to be a multiple of the flash erase page size and a power of 2.
- NVS_SECTOR_COUNT is the number of sectors, it is at least 2, one sector is always kept empty to allow copying of existing data.
- NVS_STORAGE_OFFSET is the offset of the storage area in flash.

Flash wear

When writing data to flash a study of the flash wear is important. Flash has a limited life which is determined by the number of times flash can be erased. Flash is erased one page at a time and the pagesize is determined by the hardware. As an example a nRF51822 device has a pagesize of 1024 bytes and each page can be erased about 20,000 times.

Calculating expected device lifetime Suppose we use a 4 bytes state variable that is changed every minute and needs to be restored after reboot. NVS has been defined with a sector_size equal to the pagesize (1024 bytes) and 2 sectors have been defined.

Each write of the state variable requires 12 bytes of flash storage: 8 bytes for the metadata and 4 bytes for the data. When storing the data the first sector will be full after 1024/12 = 85.33 minutes. After another 85.33 minutes, the second sector is full. When this happens, because we're using only two sectors, the first sector will be used for storage and will be erased after 171 minutes of system time. With the expected device life of 20,000 writes, with two sectors writing every 171 minutes, the device should last about 171 * 20,000 minutes, or about 6.5 years.

More generally then, with

- NS as the number of storage requests per minute,
- DS as the data size in bytes,
- SECTOR_SIZE in bytes, and
- PAGE_ERASES as the number of times the page can be erased,

the expected device life (in minutes) can be calculated as:

```
SECTOR_COUNT * SECTOR_SIZE * PAGE_ERASES / (NS * (DS+8)) minutes
```

From this formula it is also clear what to do in case the expected life is too short: increase SECTOR_COUNT or SECTOR_SIZE.

Flash write block size migration

It is possible that during a DFU process, the flash driver used by the NVS changes the supported minimal write block size. The NVS in-flash image will stay compatible unless the physical ATE size changes. Especially, migration between 1,2,4,8-bytes write block sizes is allowed.

Sample

A sample of how NVS can be used is supplied in `samples/subsys/nvs`.

Troubleshooting

MPU fault while using NVS, or `-ETIMEDOUT` error returned
NVS can use the internal flash of the SoC. While the MPU is enabled, the flash driver requires MPU RWX access to flash memory, configured using `CONFIG_MPU_ALLOW_FLASH_WRITE`. If this option is disabled, the NVS application will get an MPU fault if it references the internal SoC flash and it's the only thread running. In a multi-threaded application, another thread might intercept the fault and the NVS API will return an `-ETIMEDOUT` error.

API Reference

The NVS subsystem APIs are provided by `nvs.h`:

group **nvs_data_structures**
Non-volatile Storage Data Structures.

struct **nvs_fs**
#include <*nvs.h*> Non-volatile Storage File system structure.

Public Members

off_t **offset**
File system offset in flash.

uint32_t **ate_wra**
Allocation table entry write address.

Addresses are stored as uint32_t:
- high 2 bytes correspond to the sector
- low 2 bytes are the offset in the sector

uint32_t **data_wra**
Data write address.

uint16_t **sector_size**
File system is split into sectors, each sector must be multiple of erase-block-size.

uint16_t **sector_count**
Number of sectors in the file system.

bool `ready`
: Flag indicating if the file system is initialized.

struct *k_mutex* `nvs_lock`
: Mutex.

const struct *device* *`flash_device`
: Flash device runtime structure.

const struct *flash_parameters* *`flash_parameters`
: Flash memory parameters structure.

Related code samples

Non-Volatile Storage (NVS)
: Store and retrieve data from flash using the NVS API.

group `nvs_high_level_api`
: Non-volatile Storage APIs.

Functions

int `nvs_mount`(struct *nvs_fs* *fs)
: Mount an NVS file system onto the flash device specified in `fs`.

Parameters

- `fs` – Pointer to file system

Return values

- `0` – Success
- `-ERRNO` – errno code if error

int `nvs_clear`(struct *nvs_fs* *fs)
: Clear the NVS file system from flash.

Parameters

- `fs` – Pointer to file system

Return values

- `0` – Success
- `-ERRNO` – errno code if error

ssize_t `nvs_write`(struct *nvs_fs* *fs, uint16_t id, const void *data, size_t len)
: Write an entry to the file system.

Note

When `len` parameter is equal to `0` then entry is effectively removed (it is equivalent to calling of nvs_delete). Any calls to nvs_read for entries with data of length `0` will return error.

It is not possible to distinguish between deleted entry and entry with data of length 0.

Parameters

- `fs` – Pointer to file system
- `id` – Id of the entry to be written
- `data` – Pointer to the data to be written
- `len` – Number of bytes to be written

Returns

Number of bytes written. On success, it will be equal to the number of bytes requested to be written. When a rewrite of the same data already stored is attempted, nothing is written to flash, thus 0 is returned. On error, returns negative value of errno.h defined error codes.

int `nvs_delete`(struct *nvs_fs* *fs, uint16_t id)

Delete an entry from the file system.

Parameters

- `fs` – Pointer to file system
- `id` – Id of the entry to be deleted

Return values

- `0` – Success
- `-ERRNO` – errno code if error

ssize_t `nvs_read`(struct *nvs_fs* *fs, uint16_t id, void *data, size_t len)

Read an entry from the file system.

Parameters

- `fs` – Pointer to file system
- `id` – Id of the entry to be read
- `data` – Pointer to data buffer
- `len` – Number of bytes to be read

Returns

Number of bytes read. On success, it will be equal to the number of bytes requested to be read. When the return value is larger than the number of bytes requested to read this indicates not all bytes were read, and more data is available. On error, returns negative value of errno.h defined error codes.

ssize_t `nvs_read_hist`(struct *nvs_fs* *fs, uint16_t id, void *data, size_t len, uint16_t cnt)

Read a history entry from the file system.

Parameters

- `fs` – Pointer to file system
- `id` – Id of the entry to be read
- `data` – Pointer to data buffer
- `len` – Number of bytes to be read
- `cnt` – History counter: 0: latest entry, 1: one before latest ...

Returns
Number of bytes read. On success, it will be equal to the number of bytes requested to be read. When the return value is larger than the number of bytes requested to read this indicates not all bytes were read, and more data is available. On error, returns negative value of errno.h defined error codes.

ssize_t `nvs_calc_free_space`(struct *nvs_fs* *fs)
Calculate the available free space in the file system.

Parameters
- `fs` – Pointer to file system

Returns
Number of bytes free. On success, it will be equal to the number of bytes that can still be written to the file system. Calculating the free space is a time consuming operation, especially on spi flash. On error, returns negative value of errno.h defined error codes.

4.26.2 Disk Access

Overview

The disk access API provides access to storage devices.

Initializing Disks

Since many disk devices (such as SD cards) are hotpluggable, the disk access API provides IOCTLs to initialize and de-initialize the disk. They are as follows:

- *DISK_IOCTL_CTRL_INIT*: Initialize the disk. Must be called before additional I/O operations can be run on the disk device. Equivalent to calling the legacy function *disk_access_init()*.
- *DISK_IOCTL_CTRL_DEINIT*: De-initialize the disk. Once this IOCTL is issued, the *DISK_IOCTL_CTRL_INIT* must be issued before the disk can be used for addition I/O operations.

Init/deinit IOCTL calls are balanced, so a disk will not de-initialize until an equal number of deinit IOCTLs have been issued as init IOCTLs.

It is also possible to force a disk de-initialization by passing a pointer to a boolean set to `true` as a parameter to the *DISK_IOCTL_CTRL_DEINIT* IOCTL. This is an unsafe operation which each disk driver may handle differently, but it will always return a value indicating success.

Note that de-initializing a disk is a low level operation- typically the de-initialization and initialization calls should be left to the filesystem implementation, and the user application should not need to manually de-initialize the disk and can instead call *fs_unmount()*

SD Card support

Zephyr has support for some SD card controllers and support for interfacing SD cards via SPI. These drivers use disk driver interface and a file system can access the SD cards via disk access API. Both standard and high-capacity SD cards are supported.

> **Note**
>
> FAT filesystems are not power safe so the filesystem may become corrupted if power is lost or if the card is removed without unmounting the filesystem

SD Memory Card subsystem Zephyr supports SD memory cards via the disk driver API, or via the SDMMC subsystem. This subsystem can be used transparently via the disk driver API, but also supports direct block level access to cards. The SDMMC subsystem interacts with the *sd host controller api* to communicate with attached SD cards.

SD Card support via SPI Example devicetree fragment below shows how to add SD card node to `spi1` interface. Example uses pin `PA27` for chip select, and runs the SPI bus at 24 MHz once the SD card has been initialized:

```
&spi1 {
        status = "okay";
        cs-gpios = <&porta 27 GPIO_ACTIVE_LOW>;

        sdhc0: sdhc@0 {
                compatible = "zephyr,sdhc-spi-slot";
                reg = <0>;
                status = "okay";
                mmc {
                    compatible = "zephyr,sdmmc-disk";
                    status = "okay";
                };
                spi-max-frequency = <24000000>;
        };
};
```

The SD card will be automatically detected and initialized by the filesystem driver when the board boots.

To read and write files and directories, see the *File Systems* in include/zephyr/fs/fs.h such as *fs_open()*, *fs_read()*, and *fs_write()*.

eMMC Device Support

Zephyr also has support for eMMC devices using the Disk Access API. MMC in zephyr is implemented using the SD subsystem because the MMC bus shares a lot of similarity with the SD bus. MMC controllers also use the SDHC device driver API.

Emulated block device on flash partition support

Zephyr flashdisk driver makes it possible to use flash memory partition as a block device. The flashdisk instances are defined in devicetree:

```
/ {
    msc_disk0 {
        compatible = "zephyr,flash-disk";
        partition = <&storage_partition>;
        disk-name = "NAND";
        cache-size = <4096>;
    };
};
```

The cache size specified in `zephyr,flash-disk` node should be equal to backing partition minimum erasable block size.

NVMe disk support

NVMe disks are also supported

NVMe

NVMe is a standardized logical device interface on PCIe bus exposing storage devices.

NVMe controllers and disks are supported. Disks can be accessed via the *Disk Access API* they expose and thus be used through the *File System API*.

Driver design

The driver is sliced up in 3 main parts:

- NVMe controller: drivers/disk/nvme/nvme_controller.c
- NVMe commands: drivers/disk/nvme/nvme_cmd.c
- NVMe namespace: drivers/disk/nvme/nvme_namespace.c

Where the NVMe controller is the root of the device driver. This is the one that will get device driver instances. Note that this is only what DTS describes: the NVMe controller, and none of its namespaces (disks). The NVMe command is the generic logic used to communicate with the controller and the namespaces it exposes. Finally the NVMe namespace is the dedicated part to deal with an actual namespace which, in turn, enables applications accessing each ones through the Disk Access API drivers/disk/nvme/nvme_disk.c.

If a controller exposes more than 1 namespace (disk), it will be possible to raise the amount of built-in namespace support by tweaking the configuration option CONFIG_NVME_MAX_NAMESPACES (see below).

Each exposed disk, via it's related disk_info structure, will be distinguished by its name which is inherited from it's related namespace. As such, the disk name follows NVMe naming which is nvme<k>n<n> where k is the controller number and n the namespame number. Most of the time, if only one NVMe disk is plugged into the system, one will see 'nvme0n0' as an exposed disk.

NVMe configuration

DTS

Any board exposing an NVMe disk should provide a DTS overlay to enable its use within Zephyr

```
#include <zephyr/dt-bindings/pcie/pcie.h>
/ {
    pcie0 {
        nvme0: nvme0 {
            compatible = "nvme-controller";
            vendor-id = <VENDOR_ID>;
            device-id = <DEVICE_ID>;
            status = "okay";
        };
    };
};
```

Where VENDOR_ID and DEVICE_ID are the ones from the exposed NVMe controller.

Options

- CONFIG_NVME

Note that NVME requires the target to support PCIe multi-vector MSI-X in order to function.

- CONFIG_NVME_MAX_NAMESPACES

Important note for users NVMe specifications mandate the data buffer to be placed in a dword (4 bytes) aligned address. While this is not a problem for advanced OS managing virtual memory and dynamic allocations below the user processes, this can become an issue in Zephyr as soon as buffer addresses map directly to physical memory.

At this stage then, it is up to the user to make sure the buffer address being provided to *disk_access_read()* and *disk_access_write()* are dword aligned.

Disk Access API Configuration Options

Related configuration options:

- CONFIG_DISK_ACCESS

API Reference

Related code samples

File system manipulation
Use file system API with various filesystems and storage devices.

group `disk_access_interface`

Disk Access APIs.

Functions

int `disk_access_init`(const char *pdrv)

perform any initialization

This call is made by the consumer before doing any IO calls so that the disk or the backing device can do any initialization. Although still supported for legacy compatibility, users should instead call *disk_access_ioctl* with the IOCTL *DISK_IOCTL_CTRL_INIT*.

Disk initialization is reference counted, so only the first successful call to initialize a uninitialized (or previously de-initialized) disk will actually initialize the disk

Parameters

- `pdrv` – **[in]** Disk name

Returns
0 on success, negative errno code on fail

int `disk_access_status`(const char *pdrv)

Get the status of disk.

This call is used to get the status of the disk

Parameters

- `pdrv` – **[in]** Disk name

Returns
DISK_STATUS_OK or other DISK_STATUS_*s

int `disk_access_read`(const char *pdrv, uint8_t *data_buf, uint32_t start_sector, uint32_t num_sector)

read data from disk

Function to read data from disk to a memory buffer.

Note: if he disk is of NVMe type, user will need to ensure data_buf pointer is 4-bytes aligned.

Parameters

- `pdrv` – **[in]** Disk name
- `data_buf` – **[in]** Pointer to the memory buffer to put data.
- `start_sector` – **[in]** Start disk sector to read from
- `num_sector` – **[in]** Number of disk sectors to read

Returns
0 on success, negative errno code on fail

int `disk_access_write`(const char *pdrv, const uint8_t *data_buf, uint32_t start_sector, uint32_t num_sector)

write data to disk

Function write data from memory buffer to disk.

Note: if he disk is of NVMe type, user will need to ensure data_buf pointer is 4-bytes aligned.

Parameters

- `pdrv` – **[in]** Disk name
- `data_buf` – **[in]** Pointer to the memory buffer
- `start_sector` – **[in]** Start disk sector to write to
- `num_sector` – **[in]** Number of disk sectors to write

Returns
0 on success, negative errno code on fail

int `disk_access_ioctl`(const char *pdrv, uint8_t cmd, void *buff)

Get/Configure disk parameters.

Function to get disk parameters and make any special device requests.

Parameters

- `pdrv` – **[in]** Disk name
- `cmd` – **[in]** DISK_IOCTL_* code describing the request
- `buff` – **[in]** Command data buffer

Returns
0 on success, negative errno code on fail

Disk Driver Configuration Options

Related driver configuration options:

- `CONFIG_DISK_DRIVERS`

Disk Driver Interface

group `disk_driver_interface`

Disk Driver Interface.

Since
1.6

Version
1.0.0

Defines

`DISK_IOCTL_GET_SECTOR_COUNT`

Possible Cmd Codes for disk_ioctl()

Get the number of sectors in the disk

`DISK_IOCTL_GET_SECTOR_SIZE`

Get the size of a disk SECTOR in bytes.

`DISK_IOCTL_RESERVED`

reserved.

It used to be DISK_IOCTL_GET_DISK_SIZE

`DISK_IOCTL_GET_ERASE_BLOCK_SZ`

How many sectors constitute a FLASH Erase block.

`DISK_IOCTL_CTRL_SYNC`

Commit any cached read/writes to disk.

`DISK_IOCTL_CTRL_INIT`

Initialize the disk.

This IOCTL must be issued before the disk can be used for I/O. It is reference counted, so only the first successful invocation of this macro on an uninitialized disk will initialize the IO device

`DISK_IOCTL_CTRL_DEINIT`

Deinitialize the disk.

This IOCTL can be used to de-initialize the disk, enabling it to be removed from the system if the disk is hot-pluggable. Disk usage is reference counted, so for a given disk the `DISK_IOCTL_CTRL_DEINIT` IOCTL must be issued as many times as the `DISK_IOCTL_CTRL_INIT` IOCTL was issued in order to de-initialize it.

This macro optionally accepts a pointer to a boolean as the `buf` parameter, which if true indicates the disk should be forcibly stopped, ignoring all reference counts. The disk driver must report success if a forced stop is requested, but this operation is inherently unsafe.

DISK_STATUS_OK

Possible return bitmasks for disk_status()

Disk status okay

DISK_STATUS_UNINIT

Disk status uninitialized.

DISK_STATUS_NOMEDIA

Disk status no media.

DISK_STATUS_WR_PROTECT

Disk status write protected.

Functions

int **disk_access_register**(struct *disk_info* *disk)

Register disk.

Parameters

- **disk** – **[in]** Pointer to the disk info structure

Returns

0 on success, negative errno code on fail

int **disk_access_unregister**(struct *disk_info* *disk)

Unregister disk.

Parameters

- **disk** – **[in]** Pointer to the disk info structure

Returns

0 on success, negative errno code on fail

struct **disk_info**

#include <disk.h> Disk info.

Public Members

sys_dnode_t **node**

Internally used list node.

const char ***name**

Disk name.

const struct *disk_operations* ***ops**

Disk operations.

const struct *device* ***dev**

Device associated to this disk.

uint16_t **refcnt**
Internally used disk reference count.

struct **disk_operations**
#include <*disk.h*> Disk operations.

4.26.3 Flash map

The <zephyr/storage/flash_map.h> API allows accessing information about device flash partitions via *flash_area* structures.

Each *flash_area* describes a flash partition. The API provides access to a "flash map", which contains predefined flash areas accessible via globally unique ID numbers. The map is created from "fixed-partition" compatible entries in DTS file. Users may also create *flash_area* objects at runtime for application-specific purposes.

This documentation uses "flash area" when referencing single "fixed-partition" entities.

The *flash_area* contains a pointer to a *device*, which can be used to access the flash device an area is placed on directly with the *flash API*. Each flash area is characterized by a device it is placed on, offset from the beginning of the device and size on the device. An additional identifier parameter is used by the *flash_area_open()* function to find flash area in flash map.

The flash_map.h API provides functions for operating on a *flash_area*. The main examples are *flash_area_read()* and *flash_area_write()*. These functions are basically wrappers around the flash API with additional offset and size checks, to limit flash operations to a predefined area.

Most <zephyr/storage/flash_map.h> API functions require a *flash_area* object pointer characterizing the flash area they will be working on. There are two possible methods to obtain such a pointer:

- obtain it using *flash_area_open*;
- defining a *flash_area* type object, which requires providing a valid *device* object pointer with offset and size of the area within the flash device.

flash_area_open() uses numeric identifiers to search flash map for *flash_area* objects and returns, if found, a pointer to an object representing area with given ID. The ID number for a flash area can be obtained from a fixed-partition DTS node label using *FIXED_PARTITION_ID()*; these labels are obtained from the devicetree as described below.

Relationship with Devicetree

The flash_map.h API uses data generated from the *Devicetree API*, in particular its *Fixed flash partitions*. Zephyr additionally has some partitioning conventions used for *Device Firmware Upgrade* via the MCUboot bootloader, as well as defining partitions usable by *file systems* or other nonvolatile *storage*.

Here is an example devicetree fragment which uses fixed flash partitions for both MCUboot and a storage partition. Some details were left out for clarity.

```
/ {
        soc {
                flashctrl: flash-controller@deadbeef {
                        flash0: flash@0 {
                                compatible = "soc-nv-flash";
                                reg = <0x0 0x100000>;
```

(continues on next page)

(continued from previous page)

```
                                partitions {
                                        compatible = "fixed-partitions";
                                        #address-cells = <0x1>;
                                        #size-cells = <0x1>;

                                        boot_partition: partition@0 {
                                                reg = <0x0 0x10000>;
                                                read-only;
                                        };
                                        storage_partition: partition@1e000 {
                                                reg = <0x1e000 0x2000>;
                                        };
                                        slot0_partition: partition@20000 {
                                                reg = <0x20000 0x60000>;
                                        };
                                        slot1_partition: partition@80000 {
                                                reg = <0x80000 0x60000>;
                                        };
                                        scratch_partition: partition@e0000 {
                                                reg = <0xe0000 0x20000>;
                                        };
                                };
                        };
                };
        };
};
```

Partition offset shall be expressed in relation to the flash memory beginning address, to which the partition belongs to.

The `boot_partition`, `slot0_partition`, `slot1_partition`, and `scratch_partition` node labels are defined for MCUboot, though not all MCUboot configurations require all of them to be defined. See the MCUboot documentation for more details.

The `storage_partition` node is defined for use by a file system or other nonvolatile storage API.

Numeric flash area ID is obtained by passing DTS node label to *FIXED_PARTITION_ID()*; for example to obtain ID number for `slot0_partition`, user would invoke `FIXED_PARTITION_ID(slot0_partition)`.

All `FIXED_PARTITION_` macros take DTS node labels as partition identifiers.

Users do not have to obtain a *flash_area* object pointer using `flash_map_open()` to get information on flash area size, offset or device, if such area is defined in DTS file. Knowing the DTS node label of an area, users may use *FIXED_PARTITION_OFFSET()*, *FIXED_PARTITION_SIZE()* or *FIXED_PARTITION_DEVICE()* respectively to obtain such information directly from DTS node definition. For example to obtain offset of `storage_partition` it is enough to invoke `FIXED_PARTITION_OFFSET(storage_partition)`.

Below example shows how to obtain a *flash_area* object pointer using *flash_area_open()* and DTS node label:

```
const struct flash_area *my_area;
int err = flash_area_open(FIXED_PARTITION_ID(slot0_partition), &my_area);

if (err != 0) {
     handle_the_error(err);
} else {
     flash_area_read(my_area, ...);
}
```

API Reference

Related code samples

LittleFS filesystem
Use file system API over LittleFS.

nRF SoC Internal Storage
Use the flash API to interact with the SoC flash.

group `flash_area_api`

Abstraction over flash partitions/areas and their drivers.

Since
1.11

Version
1.0.0

Defines

`SOC_FLASH_0_ID`

Provided for compatibility with MCUboot.

`SPI_FLASH_0_ID`

Provided for compatibility with MCUboot.

`FIXED_PARTITION_EXISTS`(label)

Returns non-0 value if fixed-partition of given DTS node label exists.

Parameters

- `label` – DTS node label

Returns
non-0 if fixed-partition node exists and is enabled; 0 if node does not exist, is not enabled or is not fixed-partition.

`FIXED_PARTITION_ID`(label)

Get flash area ID from fixed-partition DTS node label.

Parameters

- `label` – DTS node label of a partition

Returns
flash area ID

`FIXED_PARTITION_OFFSET`(label)

Get fixed-partition offset from DTS node label.

Parameters

- `label` – DTS node label of a partition

Returns
fixed-partition offset, as defined for the partition in DTS.

FIXED_PARTITION_NODE_OFFSET(node)

Get fixed-partition offset from DTS node.

Parameters

- node – DTS node of a partition

Returns

fixed-partition offset, as defined for the partition in DTS.

FIXED_PARTITION_SIZE(label)

Get fixed-partition size for DTS node label.

Parameters

- label – DTS node label

Returns

fixed-partition offset, as defined for the partition in DTS.

FIXED_PARTITION_NODE_SIZE(node)

Get fixed-partition size for DTS node.

Parameters

- node – DTS node of a partition

Returns

fixed-partition size, as defined for the partition in DTS.

FLASH_AREA_DEVICE(label)

Get device pointer for device the area/partition resides on.

Parameters

- label – DTS node label of a partition

Returns

const struct device type pointer

FIXED_PARTITION_DEVICE(label)

Get device pointer for device the area/partition resides on.

Parameters

- label – DTS node label of a partition

Returns

Pointer to a device.

FIXED_PARTITION_NODE_DEVICE(node)

Get device pointer for device the area/partition resides on.

Parameters

- node – DTS node of a partition

Returns

Pointer to a device.

Typedefs

typedef void (*flash_area_cb_t)(const struct *flash_area* *fa, void *user_data)

Flash map iteration callback.

Param fa

flash area

Param user_data
User supplied data

Functions

int `flash_area_open`(uint8_t id, const struct *flash_area* **fa)

Retrieve partitions flash area from the flash_map.

Function Retrieves *flash_area* from flash_map for given partition.

Parameters

- `id` – **[in]** ID of the flash partition.
- `fa` – **[out]** Pointer which has to reference *flash_area*. If `ID` is unknown, it will be NULL on output.

Returns
0 on success, -EACCES if the flash_map is not available , -ENOENT if `ID` is unknown, -ENODEV if there is no driver attached to the area.

void `flash_area_close`(const struct *flash_area* *fa)

Close *flash_area*.

Reserved for future usage and external projects compatibility reason. Currently is NOP.

Parameters

- `fa` – **[in]** Flash area to be closed.

int `flash_area_read`(const struct *flash_area* *fa, off_t off, void *dst, size_t len)

Read flash area data.

Read data from flash area. Area readout boundaries are asserted before read request. API has the same limitation regard read-block alignment and size as wrapped flash driver.

Parameters

- `fa` – **[in]** Flash area
- `off` – **[in]** Offset relative from beginning of flash area to read
- `dst` – **[out]** Buffer to store read data
- `len` – **[in]** Number of bytes to read

Returns
0 on success, negative errno code on fail.

int `flash_area_write`(const struct *flash_area* *fa, off_t off, const void *src, size_t len)

Write data to flash area.

Write data to flash area. Area write boundaries are asserted before write request. API has the same limitation regard write-block alignment and size as wrapped flash driver.

Parameters

- `fa` – **[in]** Flash area
- `off` – **[in]** Offset relative from beginning of flash area to write
- `src` – **[in]** Buffer with data to be written
- `len` – **[in]** Number of bytes to write

Returns
0 on success, negative errno code on fail.

int `flash_area_erase`(const struct *flash_area* *fa, off_t off, size_t len)

Erase flash area.

Erase given flash area range. Area boundaries are asserted before erase request. API has the same limitation regard erase-block alignment and size as wrapped flash driver.

Parameters

- `fa` – **[in]** Flash area
- `off` – **[in]** Offset relative from beginning of flash area.
- `len` – **[in]** Number of bytes to be erase

Returns

0 on success, negative errno code on fail.

int `flash_area_flatten`(const struct *flash_area* *fa, off_t off, size_t len)

Erase flash area or fill with erase-value.

On program-erase devices this function behaves exactly like flash_area_erase. On RAM non-volatile device it will call erase, if driver provides such callback, or will fill given range with erase-value defined by driver. This function should be only used by code that has not been written to directly support devices that do not require erase and rely on device being erased prior to some operations. Note that emulated erase, on devices that do not require, is done via write, which affects endurance of device.

See also

flash_area_erase()

See also

flash_flatten()

Parameters

- `fa` – **[in]** Flash area
- `off` – **[in]** Offset relative from beginning of flash area.
- `len` – **[in]** Number of bytes to be erase

Returns

0 on success, negative errno code on fail.

uint32_t `flash_area_align`(const struct *flash_area* *fa)

Get write block size of the flash area.

Currently write block size might be treated as read block size, although most of drivers supports unaligned readout.

Parameters

- `fa` – **[in]** Flash area

Returns

Alignment restriction for flash writes in [B].

int `flash_area_get_sectors`(int fa_id, uint32_t *count, struct *flash_sector* *sectors)

Retrieve info about sectors within the area.

Parameters

- `fa_id` – **[in]** Given flash area ID
- `sectors` – **[out]** buffer for sectors data
- `count` – **[inout]** On input Capacity of `sectors`, on output number of sectors Retrieved.

Returns

0 on success, negative errno code on fail. Especially returns -ENOMEM if There are too many flash pages on the *flash_area* to fit in the array.

void `flash_area_foreach`(*flash_area_cb_t* user_cb, void *user_data)

Iterate over flash map.

Parameters

- `user_cb` – User callback
- `user_data` – User supplied data

int `flash_area_has_driver`(const struct *flash_area* *fa)

Check whether given flash area has supporting flash driver in the system.

Parameters

- `fa` – **[in]** Flash area.

Returns

1 On success. -ENODEV if no driver match.

const struct *device* *`flash_area_get_device`(const struct *flash_area* *fa)

Get driver for given flash area.

Parameters

- `fa` – **[in]** Flash area.

Returns

device driver.

uint8_t `flash_area_erased_val`(const struct *flash_area* *fa)

Get the value expected to be read when accessing any erased flash byte.

This API is compatible with the MCUBoot's porting layer.

Parameters

- `fa` – Flash area.

Returns

Byte value of erase memory.

struct `flash_area`

#include <flash_map.h> Flash partition.

This structure represents a fixed-size partition on a flash device. Each partition contains one or more flash sectors.

Public Members

uint8_t **fa_id**
: ID number.

off_t **fa_off**
: Start offset from the beginning of the flash device.

size_t **fa_size**
: Total size.

const struct *device* ***fa_dev**
: Backing flash device.

struct **flash_sector**
: *#include <flash_map.h>* Structure for transfer flash sector boundaries.

This template is used for presentation of flash memory structure. It consumes much less RAM than *flash_area*

Public Members

off_t **fs_off**
: Sector offset from the beginning of the flash device.

size_t **fs_size**
: Sector size in bytes.

4.26.4 Flash Circular Buffer (FCB)

Flash circular buffer provides an abstraction through which you can treat flash like a FIFO. You append entries to the end, and read data from the beginning.

Note

As of Zephyr release 2.1 the *NVS* storage API is recommended over FCB for use as a back-end for the *settings API*.

Description

Entries in the flash contain the length of the entry, the data within the entry, and checksum over the entry contents.

Storage of entries in flash is done in a FIFO fashion. When you request space for the next entry, space is located at the end of the used area. When you start reading, the first entry served is the oldest entry in flash.

Entries can be appended to the end of the area until storage space is exhausted. You have control over what happens next; either erase oldest block of data, thereby freeing up some space, or stop writing new data until existing data has been collected. FCB treats underlying storage as an array of flash sectors; when it erases old data, it does this a sector at a time.

Entries in the flash are checksummed. That is how FCB detects whether writing entry to flash completed ok. It will skip over entries which don't have a valid checksum.

Usage

To add an entry to circular buffer:

- Call *fcb_append()* to get the location where data can be written. If this fails due to lack of space, you can call *fcb_rotate()* to erase the oldest sector which will make the space. And then call *fcb_append()* again.
- Use *flash_area_write()* to write entry contents.
- Call *fcb_append_finish()* when done. This completes the writing of the entry by calculating the checksum.

To read contents of the circular buffer:

- Call *fcb_walk()* with a pointer to your callback function.
- Within callback function copy in data from the entry using *flash_area_read()*. You can tell when all data from within a sector has been read by monitoring the returned entry's area pointer. Then you can call *fcb_rotate()*, if you're done with that data.

Alternatively:

- Call *fcb_getnext()* with 0 in entry offset to get the pointer to the oldest entry.
- Use *flash_area_read()* to read entry contents.
- Call *fcb_getnext()* with pointer to current entry to get the next one. And so on.

API Reference

The FCB subsystem APIs are provided by `fcb.h`:

Data structures

group `fcb_data_structures`

Defines

`FCB_MAX_LEN`
Max length of element (16,383)

`FCB_ENTRY_FA_DATA_OFF`(entry)
Helper macro for calculating the data offset related to the fcb *flash_area* start offset.

Parameters

- `entry` – fcb entry structure

`FCB_FLAGS_CRC_DISABLED`
Flag to disable CRC for the fcb_entries in flash.

struct `fcb_entry`

#include <fcb.h> FCB entry info structure.

This data structure describes the element location in the flash.

You would use it to figure out what parameters to pass to *flash_area_read()* to read element contents. Or to *flash_area_write()* when adding a new element. Entry location is pointer to area (within fcb->f_sectors), and offset within that area.

Public Members

struct *flash_sector* *`fe_sector`

Pointer to info about sector where data are placed.

uint32_t `fe_elem_off`

Offset from the start of the sector to beginning of element.

uint32_t `fe_data_off`

Offset from the start of the sector to the start of element.

uint16_t `fe_data_len`

Size of data area in fcb entry.

struct `fcb_entry_ctx`

#include <fcb.h> Structure for transferring complete information about FCB entry location within flash memory.

Public Members

struct *fcb_entry* `loc`

FCB entry info.

const struct *flash_area* *`fap`

Flash area where the entry is placed.

struct `fcb`

#include <fcb.h> FCB instance structure.

The following data structure describes the FCB itself. First part should be filled in by the user before calling *fcb_init*. The second part is used by FCB for its internal bookkeeping.

Public Members

uint32_t `f_magic`

Magic value, should not be 0xFFFFFFFF.

It is xored with inversion of f_erase_value and placed in the beginning of FCB flash sector. FCB uses this when determining whether sector contains valid data or not. Giving it value of 0xFFFFFFFF means leaving bytes of the filed in “erased” state.

uint8_t **f_version**

Current version number of the data.

uint8_t **f_sector_cnt**

Number of elements in sector array.

uint8_t **f_scratch_cnt**

Number of sectors to keep empty.

This can be used if you need to have scratch space for garbage collecting when FCB fills up.

struct *flash_sector* ***f_sectors**

Array of sectors, must be contiguous.

struct *k_mutex* **f_mtx**

Locking for accessing the FCB data, internal state.

struct *flash_sector* ***f_oldest**

Pointer to flash sector containing the oldest data, internal state.

struct *fcb_entry* **f_active**

internal state

uint16_t **f_active_id**

Flash location where the newest data is, internal state.

uint8_t **f_align**

writes to flash have to aligned to this, internal state

const struct *flash_area* ***fap**

Flash area used by the fcb instance, internal state.

This can be transfer to FCB user

uint8_t **f_erase_value**

The value flash takes when it is erased.

This is read from flash parameters and initialized upon call to fcb_init.

API functions

group **fcb_api**

Flash Circular Buffer APIs.

Typedefs

typedef int (*`fcb_walk_cb`)(struct *fcb_entry_ctx* *loc_ctx, void *arg)

FCB Walk callback function type.

Type of function which is expected to be called while walking over fcb entries thanks to a *fcb_walk* call.

Entry data can be read using *flash_area_read()*, using loc_ctx fields as arguments. If cb wants to stop the walk, it should return non-zero value.

Param loc_ctx
[in] entry location information (full context)

Param arg
[inout] callback context, transferred from *fcb_walk*.

Return
0 continue walking, non-zero stop walking.

Functions

int `fcb_init`(int f_area_id, struct *fcb* *fcbp)

Initialize FCB instance.

Parameters

- `f_area_id` – **[in]** ID of flash area where fcb storage resides.
- `fcbp` – **[inout]** FCB instance structure.

Returns
0 on success, non-zero on failure.

int `fcb_append`(struct *fcb* *fcbp, uint16_t len, struct *fcb_entry* *loc)

Appends an entry to circular buffer.

When writing the contents for the entry, use loc->fe_sector and loc->fe_data_off with *flash_area_write()* to fcb *flash_area*. When you're finished, call *fcb_append_finish()* with loc as argument.

Parameters

- `fcbp` – **[in]** FCB instance structure.
- `len` – **[in]** Length of data which are expected to be written as the entry payload.
- `loc` – **[out]** entry location information

Returns
0 on success, non-zero on failure.

int `fcb_append_finish`(struct *fcb* *fcbp, struct *fcb_entry* *append_loc)

Finishes entry append operation.

Parameters

- `fcbp` – **[in]** FCB instance structure.
- `append_loc` – **[in]** entry location information

Returns
0 on success, non-zero on failure.

int `fcb_walk`(struct *fcb* *fcbp, struct *flash_sector* *sector, *fcb_walk_cb* cb, void *cb_arg)

Walk over all entries in the FCB sector.

Parameters

- `sector` – **[in]** fcb sector to be walked. If null, traverse entire storage.
- `fcbp` – **[in]** FCB instance structure.
- `cb` – **[in]** pointer to the function which gets called for every entry. If cb wants to stop the walk, it should return non-zero value.
- `cb_arg` – **[inout]** callback context, transferred to the callback implementation.

Returns

0 on success, negative on failure (or transferred form callback return-value), positive transferred form callback return-value

int `fcb_getnext`(struct *fcb* *fcbp, struct *fcb_entry* *loc)

Get next fcb entry location.

Function to obtain fcb entry location in relation to entry pointed by

loc. If loc->fe_sector is set and loc->fe_elem_off is not 0 function fetches next fcb entry location. If loc->fe_sector is NULL function fetches the oldest entry location within FCB storage. loc->fe_sector is set and loc->fe_elem_off is 0 function fetches the first entry location in the fcb sector.

Parameters

- `fcbp` – **[in]** FCB instance structure.
- `loc` – **[inout]** entry location information

Returns

0 on success, non-zero on failure.

int `fcb_rotate`(struct *fcb* *fcbp)

Rotate fcb sectors.

Function erases the data from oldest sector. Upon that the next sector becomes the oldest. Active sector is also switched if needed.

Parameters

- `fcbp` – **[in]** FCB instance structure.

int `fcb_append_to_scratch`(struct *fcb* *fcbp)

Start using the scratch block.

Take one of the scratch blocks into use. So a scratch sector becomes active sector to which entries can be appended.

Parameters

- `fcbp` – **[in]** FCB instance structure.

Returns

0 on success, non-zero on failure.

int `fcb_free_sector_cnt`(struct *fcb* *fcbp)

Get free sector count.

Parameters

- `fcbp` – **[in]** FCB instance structure.

Returns

Number of free sectors.

int `fcb_is_empty`(struct *fcb* *fcbp)

Check whether FCB has any data.

Parameters

- `fcbp` – **[in]** FCB instance structure.

Returns

Positive value if fcb is empty, otherwise 0.

int `fcb_offset_last_n`(struct *fcb* *fcbp, uint8_t entries, struct *fcb_entry* *last_n_entry)

Finds the fcb entry that gives back up to n entries at the end.

Parameters

- `fcbp` – **[in]** FCB instance structure.
- `entries` – **[in]** number of fcb entries the user wants to get
- `last_n_entry` – **[out]** last_n_entry the *fcb_entry* to be returned

Returns

0 on there are any fcbs available; -ENOENT otherwise

int `fcb_clear`(struct *fcb* *fcbp)

Clear fcb instance storage.

Parameters

- `fcbp` – **[in]** FCB instance structure.

Returns

0 on success; non-zero on failure

4.26.5 Stream Flash

The Stream Flash module takes contiguous fragments of a stream of data (e.g. from radio packets), aggregates them into a user-provided buffer, then when the buffer fills (or stream ends) writes it to a raw flash partition. It supports providing the read-back buffer to the client to use in validating the persisted stream content.

One typical use of a stream write operation is when receiving a new firmware image to be used in a DFU operation.

There are several reasons why one might want to use buffered writes instead of writing the data directly as it is made available. Some devices have hardware limitations which does not allow flash writes to be performed in parallel with other operations, such as radio RX and TX. Also, fewer write operations result in faster response times seen from the application.

Persistent stream write progress

Some stream write operations, such as DFU operations, may run for a long time. When performing such long running operations it can be useful to be able to save the stream write progress to persistent storage so that the operation can resume at the same point after an unexpected interruption.

The Stream Flash module offers an API for loading, saving and clearing stream write progress to persistent storage using the *Settings* module. The API can be enabled using `CONFIG_STREAM_FLASH_PROGRESS`.

API Reference

group `stream_flash`

Abstraction over stream writes to flash.

Since
2.3

Version
0.1.0

Typedefs

typedef int (***stream_flash_callback_t**)(uint8_t *buf, size_t len, size_t offset)

Signature for callback invoked after flash write completes.

Functions of this type are invoked with a buffer containing data read back from the flash after a flash write has completed. This enables verifying that the data has been correctly stored (for instance by using a SHA function). The write buffer 'buf' provided in stream_flash_init is used as a read buffer for this purpose.

Param buf
Pointer to the data read.

Param len
The length of the data read.

Param offset
The offset the data was read from.

Functions

int **stream_flash_init**(struct *stream_flash_ctx* *ctx, const struct *device* *fdev, uint8_t *buf, size_t buf_len, size_t offset, size_t size, *stream_flash_callback_t* cb)

Initialize context needed for stream writes to flash.

Parameters

- `ctx` – context to be initialized
- `fdev` – Flash device to operate on
- `buf` – Write buffer
- `buf_len` – Length of write buffer. Can not be larger than the page size. Must be multiple of the flash device write-block-size.
- `offset` – Offset within flash device to start writing to
- `size` – Number of bytes available for performing buffered write. If this is '0', the size will be set to the total size of the flash device minus the offset.
- `cb` – Callback to be invoked on completed flash write operations.

Returns
non-negative on success, negative errno code on fail

size_t **stream_flash_bytes_written**(struct *stream_flash_ctx* *ctx)

Read number of bytes written to the flash.

> **Note**
>
> api-tags: pre-kernel-ok isr-ok

Parameters

- **ctx** – context

Returns

Number of payload bytes written to flash.

int **stream_flash_buffered_write**(struct *stream_flash_ctx* *ctx, const uint8_t *data, size_t len, bool flush)

Process input buffers to be written to flash device in single blocks.

Will store remainder between calls.

A write with the `flush` set to true has to be issued as the last write request for a given context, as it concludes write of a stream, and flushes buffers to storage device.

> **Warning**
>
> There must not be any additional write requests issued for a flushed context, unless it is re-initialized, as such write attempts may result in the function failing and returning error. Once context has been flushed, it can be re-initialized and re-used for new stream flash session.

Parameters

- **ctx** – context
- **data** – data to write
- **len** – Number of bytes to write
- **flush** – when true this forces any buffered data to be written to flash

Returns

non-negative on success, negative errno code on fail

int **stream_flash_erase_page**(struct *stream_flash_ctx* *ctx, off_t off)

Erase the flash page to which a given offset belongs.

This function erases a flash page to which an offset belongs if this page is not the page previously erased by the provided ctx (ctx->last_erased_page_start_offset).

Parameters

- **ctx** – context
- **off** – offset from the base address of the flash device

Returns

non-negative on success, negative errno code on fail

int **stream_flash_progress_load**(struct *stream_flash_ctx* *ctx, const char *settings_key)

Load persistent stream write progress stored with key `settings_key` .

This function should be called directly after *stream_flash_init* to load previous stream write progress before writing any data. If the loaded progress has fewer bytes written than `ctx` then it will be ignored.

Parameters

- **ctx** – context
- **settings_key** – key to use with the settings module for loading the stream write progress

Returns
non-negative on success, negative errno code on fail

int **stream_flash_progress_save**(struct *stream_flash_ctx* *ctx, const char *settings_key)

Save persistent stream write progress using key `settings_key` .

Parameters

- **ctx** – context
- **settings_key** – key to use with the settings module for storing the stream write progress

Returns
non-negative on success, negative errno code on fail

int **stream_flash_progress_clear**(struct *stream_flash_ctx* *ctx, const char *settings_key)

Clear persistent stream write progress stored with key `settings_key` .

Parameters

- **ctx** – context
- **settings_key** – key previously used for storing the stream write progress

Returns
non-negative on success, negative errno code on fail

struct **stream_flash_ctx**

#include <stream_flash.h> Structure for stream flash context.

Users should treat these structures as opaque values and only interact with them through the below API.

4.27 Sensing Subsystem

- *Overview*
- *Configurability*
- *Main Features*
- *Major Flows*
- *Sensor Types And Instance*
- *Sensor Instance Handler*
- *Sensor Sample Value*
- *Device Tree Configuration*
- *API Reference*

4.27.1 Overview

Sensing Subsystem is a high level sensor framework inside the OS user space service layer. It is a framework focused on sensor fusion, client arbitration, sampling, timing, scheduling and sensor based power management.

Key concepts in Sensing Subsystem include physical sensor and virtual sensor objects, and a scheduling framework over sensor object relationships. Physical sensors do not depend on any other sensor objects for input, and will directly interact with existing zephyr sensor device drivers. Virtual sensors rely on other sensor objects (physical or virtual) as report inputs.

The sensing subsystem relies on Zephyr sensor device APIs (existing version or update in future) to leverage Zephyr's large library of sensor device drivers (100+).

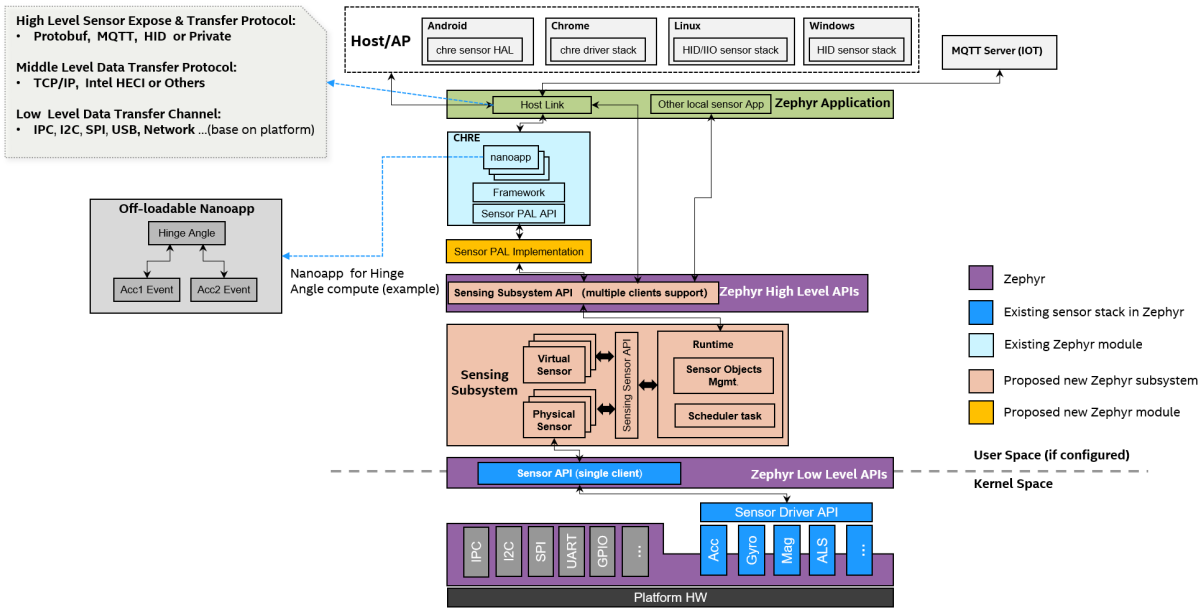
Use of the sensing subsystem is optional. Applications that only need to access simple sensors devices can use the Zephyr *Sensors* API directly.

Since the sensing subsystem is separated from device driver layer or kernel space and could support various customizations and sensor algorithms in user space with virtual sensor concepts. The existing sensor device driver can focus on low layer device side works, can keep simple as much as possible, just provide device HW abstraction and operations etc. This is very good for system stability.

The sensing subsystem is decoupled with any sensor expose/transfer protocols, the target is to support various up-layer frameworks and Applications with different sensor expose/transfer protocols, such as CHRE, HID sensors Applications, MQTT sensor Applications according different products requirements. Or even support multiple Applications with different up-layer sensor protocols at the same time with it's multiple clients support design.

Sensing subsystem can help build a unified Zephyr sensing architecture for cross host OSes support and as well as IoT sensor solutions.

The diagram below illustrates how the Sensing Subsystem integrates with up-layer frameworks.



4.27.2 Configurability

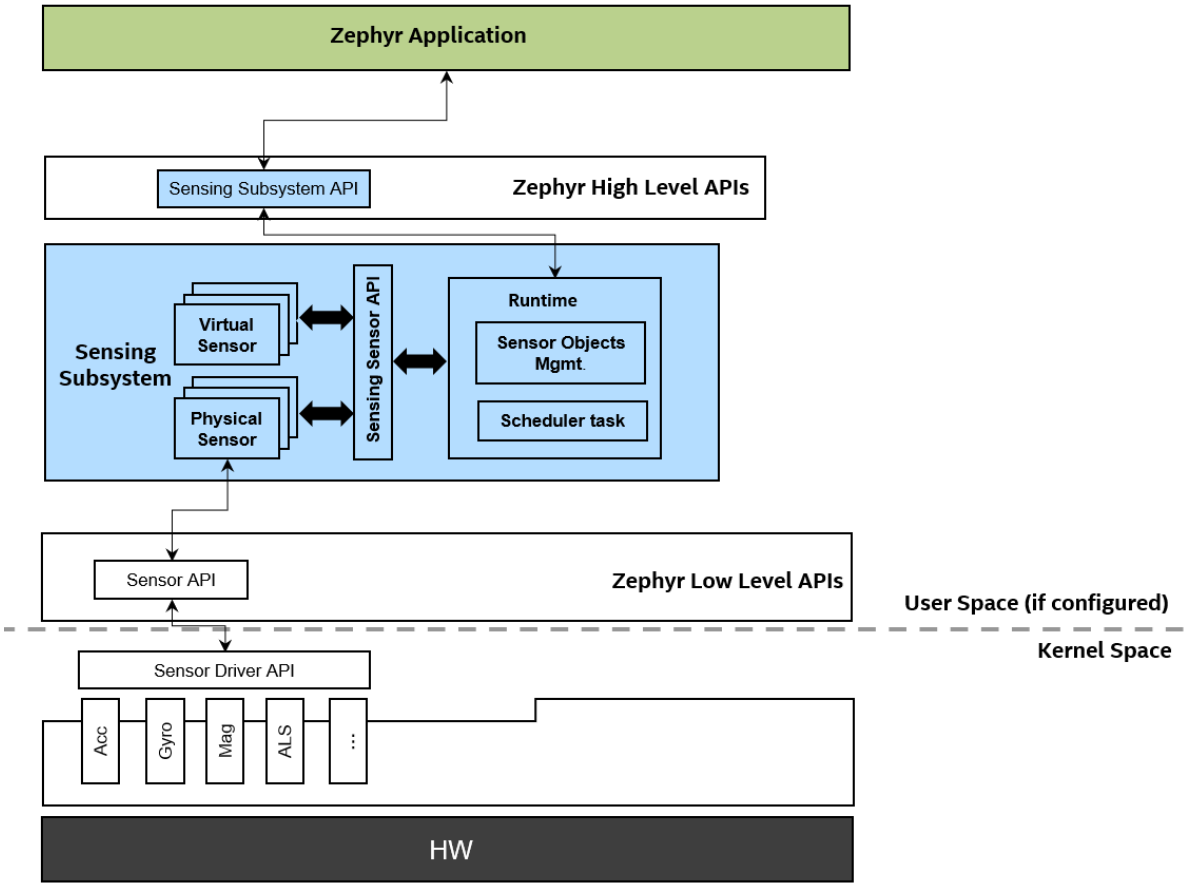
- Reusable and configurable standalone subsystem.
- Based on Zephyr existing low-level Sensor API (reuse 100+ existing sensor device drivers)
- Provide Zephyr high-level Sensing Subsystem API for Applications.

- Separate option CHRE Sensor PAL Implementation module to support CHRE.
- Decoupled with any host link protocols, it's Zephyr Application's role to handle different protocols (MQTT, HID or Private, all configurable)

4.27.3 Main Features

- **Scope**
  - Focus on framework for sensor fusion, multiple clients, arbitration, data sampling, timing management and scheduling.
- **Sensor Abstraction**
  - `Physical sensor`: interacts with Zephyr sensor device drivers, focus on data collecting.
  - `Virtual sensor`: relies on other sensor(s), `physical` or `virtual`, focus on data fusion.
- **Data Driven Model**
  - `Polling mode`: periodical sampling rate
  - `Interrupt mode`: data ready, threshold interrupt etc.
- **Scheduling**
  - single thread main loop for all sensor objects sampling and process.
- Buffer Mode for Batching
- Configurable Via Device Tree

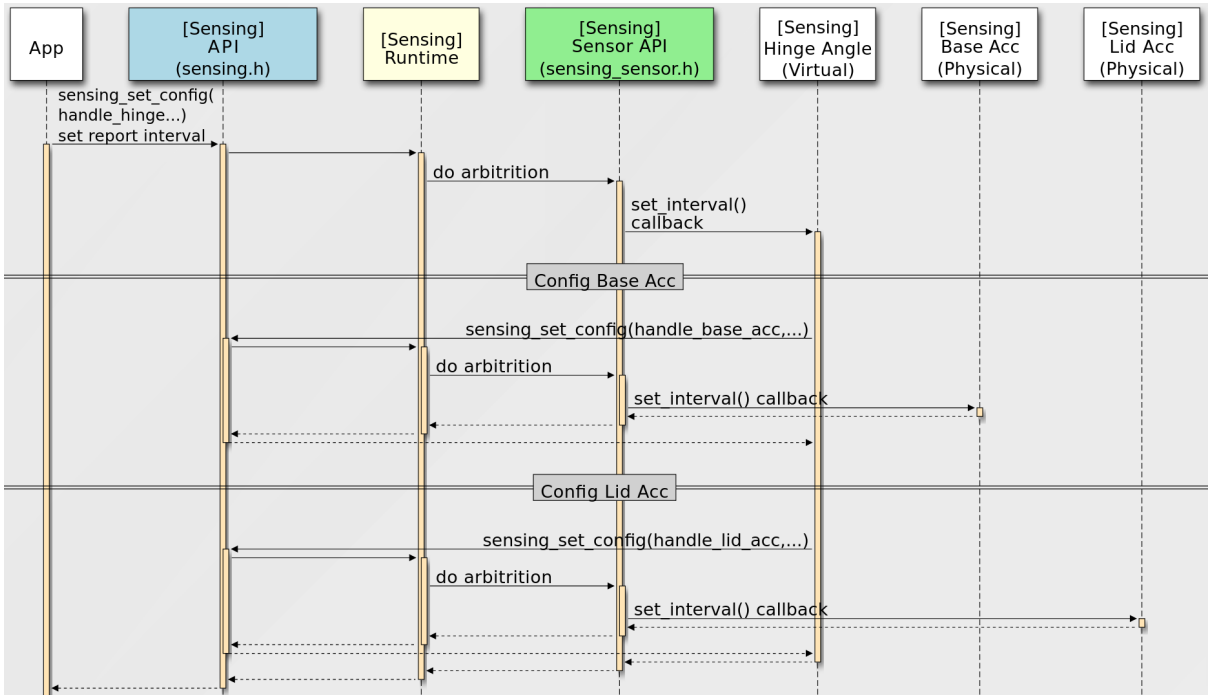
Below diagram shows the API position and scope:



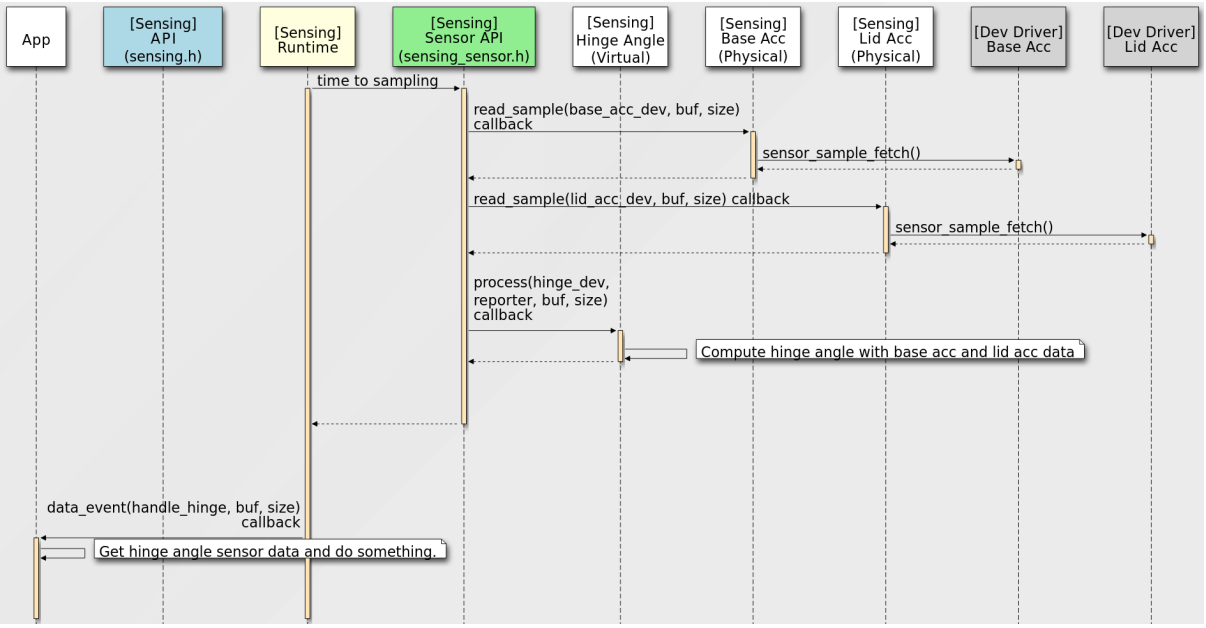
`Sensing Subsystem API` is for Applications. `Sensing Sensor API` is for development sensors.

4.27.4 Major Flows

- Sensor Configuration Flow



- Sensor Data Flow



4.27.5 Sensor Types And Instance

The `Sensing Subsystem` supports multiple instances of the same sensor type, there're two methods for Applications to identify and open an unique sensor instance:

- Enumerate all sensor instances

  `sensing_get_sensors()` returns all current board configuration supported sensor instances' information in a *`sensing_sensor_info`* pointer array .

  Then Applications can use *`sensing_open_sensor()`* to open specific sensor instance for future accessing, configuration and receive sensor data etc.

  This method is suitable for supporting some up-layer frameworks like `CHRE`, `HID` which need to dynamically enumerate the underlying platform's sensor instances.

- Open the sensor instance by devicetree node directly

  Applications can use *`sensing_open_sensor_by_dt()`* to open a sensor instance directly with sensor devicetree node identifier.

  For example:

```
sensing_open_sensor_by_dt(DEVICE_DT_GET(DT_NODELABEL(base_accel)), cb_list, handle);
sensing_open_sensor_by_dt(DEVICE_DT_GET(DT_CHOSEN(zephyr_sensing_base_accel)), cb_list,␣
↪handle);
```

This method is useful and easy use for some simple Application which just want to access specific sensor(s).

`Sensor type` follows the HID standard sensor types definition.

See include/zephyr/sensing/sensing_sensor_types.h

4.27.6 Sensor Instance Handler

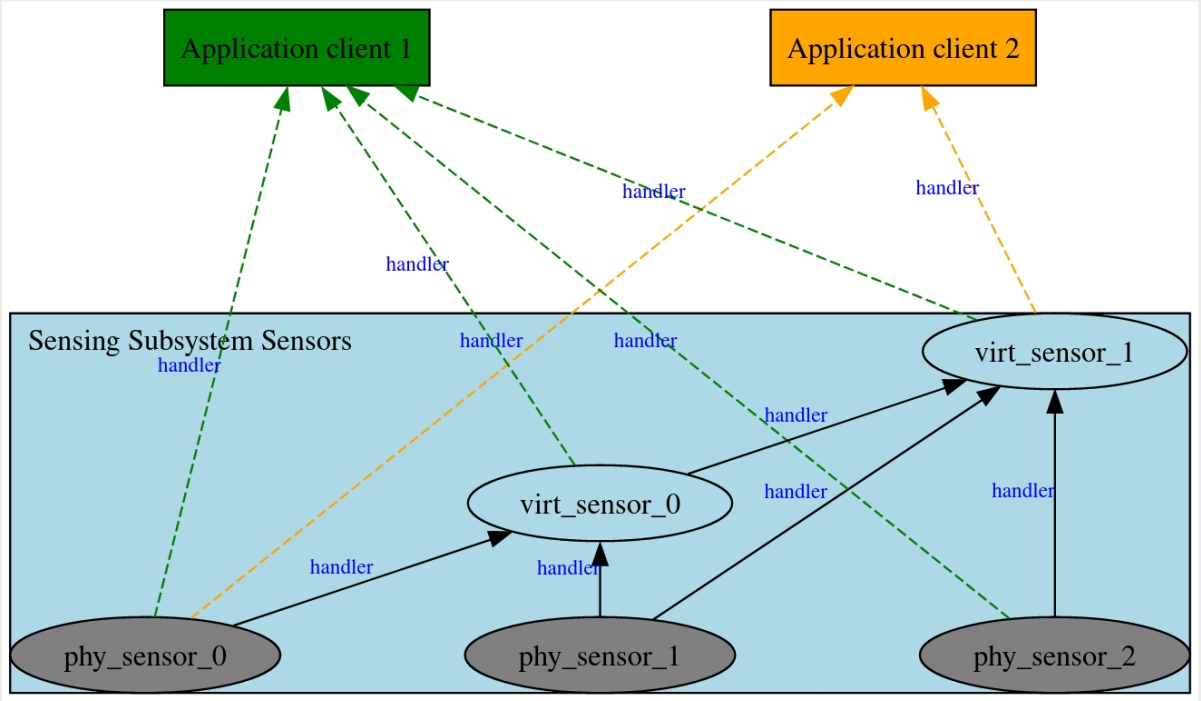
Clients using a *`sensing_sensor_handle_t`* type handler to handle a opened sensor instance, and all subsequent operations on this sensor instance need use this handler, such as set configurations, read sensor sample data, etc.

For a sensor instance, could have two kinds of clients: `Application clients` and `Sensor clients`.

`Application clients` can use *sensing_open_sensor()* to open a sensor instance and get it's handler.

For `Sensor clients`, there is no open API for opening a reporter, because the client-report relationship is built at the sensor's registration stage with devicetree.

The `Sensing Subsystem` will auto open and create `handlers` for client sensor to it's reporter sensors. `Sensor clients` can get it's reporters' handlers via *sensing_sensor_get_reporters()*.



Note

Sensors inside the Sensing Subsystem, the reporting relationship between them are all auto generated by Sensing Subsystem according devicetree definitions, handlers between client sensor and reporter sensors are auto created. Application(s) need to call *sensing_open_sensor()* to explicitly open the sensor instance.

4.27.7 Sensor Sample Value

- Data Structure

  Each sensor sample value defines as a common `header + readings[]` data structure, like *sensing_sensor_value_3d_q31*, *sensing_sensor_value_q31*, and *sensing_sensor_value_uint32*.

  The `header` definition *sensing_sensor_value_header()*.

- Time Stamp

  Time stamp unit in sensing subsystem is `micro seconds`.

  The `header` defines a **base_timestamp**, and each element in the **readings[]** array defines **timestamp_delta**.

  The **timestamp_delta** is in relation to the previous **readings** (or the **base_timestamp**)

  For example:

  - timestamp of `readings[0]` is `header.base_timestamp + readings[0].timestamp_delta`.
  - timestamp of `readings[1]` is timestamp of `readings[0] + readings[1].timestamp_delta`.

  Since timestamp unit is micro seconds, the max **timestamp_delta** (`uint32_t`) is `4295` seconds.

  If a sensor has batched data where two consecutive readings differ by more than `4295` seconds, the sensing subsystem runtime will split them across multiple instances of the readings structure, and send multiple events.

  This concept is referred from CHRE Sensor API.

- Data Format

  `Sensing Subsystem` uses per sensor type defined data format structure, and support `Q Format` defined in include/zephyr/dsp/types.h for zdsp lib support.

  For example *sensing_sensor_value_3d_q31* can be used by 3D IMU sensors like *SENSING_SENSOR_TYPE_MOTION_ACCELEROMETER_3D*, *SENSING_SENSOR_TYPE_MOTION_UNCALIB_ACCELEROMETER_3D*, and *SENSING_SENSOR_TYPE_MOTION_GYROMETER_3D*.

  sensing_sensor_value_uint32 can be used by *SENSING_SENSOR_TYPE_LIGHT_AMBIENTLIGHT* sensor,

  and *sensing_sensor_value_q31* can be used by *SENSING_SENSOR_TYPE_MOTION_HINGE_ANGLE* sensor

  See include/zephyr/sensing/sensing_datatypes.h

4.27.8 Device Tree Configuration

Sensing subsystem using device tree to configuration all sensor instances and their properties, reporting relationships.

See the example samples/subsys/sensing/simple/boards/native_sim.overlay

4.27.9 API Reference

group `sensing_sensor_types`

Sensor Types Definition.

Sensor types definition followed HID standard. https://usb.org/sites/default/files/hutrr39b_0.pdf

TODO: will add more types

Defines

`SENSING_SENSOR_TYPE_LIGHT_AMBIENTLIGHT`

sensor category light

`SENSING_SENSOR_TYPE_MOTION_ACCELEROMETER_3D`

sensor category motion

SENSING_SENSOR_TYPE_MOTION_GYROMETER_3D

SENSING_SENSOR_TYPE_MOTION_MOTION_DETECTOR

SENSING_SENSOR_TYPE_OTHER_CUSTOM

sensor category other

SENSING_SENSOR_TYPE_MOTION_UNCALIB_ACCELEROMETER_3D

SENSING_SENSOR_TYPE_MOTION_HINGE_ANGLE

SENSING_SENSOR_TYPE_ALL

Sensor type for all sensors.

This macro defines the sensor type for all sensors.

group sensing_datatypes

Data Types.

struct sensing_sensor_value_header

#include <*sensing_datatypes.h*> sensor value header

Each sensor value data structure should have this header

Here use 'base_timestamp' (uint64_t) and 'timestamp_delta' (uint32_t) to save memory usage in batching mode.

The 'base_timestamp' is for readings[0], the 'timestamp_delta' is relation to the previous 'readings'. So, timestamp of readings[0] is header.base_timestamp + readings[0].timestamp_delta. timestamp of readings[1] is timestamp of readings[0] + readings[1].timestamp_delta.

Since timestamp unit is micro seconds, the max 'timestamp_delta' (uint32_t) is 4295 seconds.

If a sensor has batched data where two consecutive readings differ by more than 4295 seconds, the sensor subsystem core will split them across multiple instances of the readings structure, and send multiple events.

This concept is borrowed from CHRE: https://cs.android.com/android/platform/superproject/+/master:\ system/chre/chre_api/include/chre_api/chre/sensor_types.h

Public Members

uint64_t base_timestamp

Base timestamp of this data readings, unit is micro seconds.

uint16_t reading_count

Count of this data readings.

struct sensing_sensor_value_3d_q31

#include <*sensing_datatypes.h*> Sensor value data structure types based on common data types.

Suitable for common sensors, such as IMU, Light sensors and orientation sensors.

Sensor value data structure for 3-axis sensors. struct *sensing_sensor_value_3d_q31* can be used by 3D IMU sensors like: SENSING_SENSOR_TYPE_MOTION_ACCELEROMETER_3D, SENSING_SENSOR_TYPE_MOTION_UNCALIB_ACCELEROMETER_3D, SENSING_SENSOR_TYPE_MOTION_GYROMETER_3D, q31 version

Public Members

struct *sensing_sensor_value_header* **header**

Header of the sensor value data structure.

int8_t **shift**

The shift value for the q31_t v[3] reading.

uint32_t **timestamp_delta**

Timestamp delta of the reading.

Unit is micro seconds.

q31_t v[3]

3D vector of the reading represented as an array.

For SENSING_SENSOR_TYPE_MOTION_ACCELEROMETER_3D and SENSING_SENSOR_TYPE_MOTION_UNCALIB_ACCELEROMETER_3D, the unit is Gs (gravitational force). For SENSING_SENSOR_TYPE_MOTION_GYROMETER_3D, the unit is degrees.

q31_t **x**

X value of the 3D vector.

q31_t **y**

Y value of the 3D vector.

q31_t **z**

Z value of the 3D vector.

struct *sensing_sensor_value_3d_q31* **readings**[1]

Array of readings.

struct **sensing_sensor_value_uint32**

#include <sensing_datatypes.h> Sensor value data structure for single 1-axis value.

struct *sensing_sensor_value_uint32* can be used by SENSING_SENSOR_TYPE_LIGHT_AMBIENTLIGHT sensor uint32_t version

Public Members

struct *sensing_sensor_value_header* **header**

Header of the sensor value data structure.

uint32_t **timestamp_delta**

Timestamp delta of the reading.

Unit is micro seconds.

uint32_t **v**

Value of the reading.

For SENSING_SENSOR_TYPE_LIGHT_AMBIENTLIGHT, the unit is luxs.

struct *sensing_sensor_value_uint32* **readings**[1]

Array of readings.

struct **sensing_sensor_value_q31**

#include <*sensing_datatypes.h*> Sensor value data structure for single 1-axis value.

struct *sensing_sensor_value_q31* can be used by SENSING_SENSOR_TYPE_MOTION_HINGE_ANGLE sensor q31 version

Public Members

struct *sensing_sensor_value_header* **header**

Header of the sensor value data structure.

int8_t **shift**

The shift value for the q31_t v reading.

uint32_t **timestamp_delta**

Timestamp delta of the reading.

Unit is micro seconds.

q31_t **v**

Value of the reading.

For SENSING_SENSOR_TYPE_MOTION_HINGE_ANGLE, the unit is degrees.

struct *sensing_sensor_value_q31* **readings**[1]

Array of readings.

Related code samples

Sensing subsystem

Get high-level sensor data in defined intervals.

group **sensing_api**

Sensing Subsystem API.

Defines

SENSING_SENSOR_VERSION(_major, _minor, _hotfix, _build)

Macro to create a sensor version value.

SENSING_SENSOR_FLAG_REPORT_ON_EVENT

Sensor flag indicating if this sensor is on event reporting data.

Reporting sensor data when the sensor event occurs, such as a motion detect sensor reporting a motion or motionless detected event.

SENSING_SENSOR_FLAG_REPORT_ON_CHANGE

Sensor flag indicating if this sensor is on change reporting data.

Reporting sensor data when the sensor data changes.

Exclusive with *SENSING_SENSOR_FLAG_REPORT_ON_EVENT*

SENSING_SENSITIVITY_INDEX_ALL

SENSING_SENSITIVITY_INDEX_ALL indicating sensitivity of each data field should be set.

Typedefs

typedef void *sensing_sensor_handle_t

Define Sensing subsystem sensor handle.

typedef void (*sensing_data_event_t)(*sensing_sensor_handle_t* handle, const void *buf, void *context)

Sensor data event receive callback.

Param handle
The sensor instance handle.

Param buf
The data buffer with sensor data.

Param context
User provided context pointer.

Enums

enum sensing_sensor_state

Sensing subsystem sensor state.

Values:

enumerator SENSING_SENSOR_STATE_READY = 0

The sensor is ready.

enumerator SENSING_SENSOR_STATE_OFFLINE = 1

The sensor is offline.

enum **sensing_sensor_attribute**

Sensing subsystem sensor config attribute.

Values:

enumerator **SENSING_SENSOR_ATTRIBUTE_INTERVAL** = 0

The interval attribute of a sensor configuration.

enumerator **SENSING_SENSOR_ATTRIBUTE_SENSITIVITY** = 1

The sensitivity attribute of a sensor configuration.

enumerator **SENSING_SENSOR_ATTRIBUTE_LATENCY** = 2

The latency attribute of a sensor configuration.

enumerator **SENSING_SENSOR_ATTRIBUTE_MAX**

The maximum number of attributes that a sensor configuration can have.

Functions

int **sensing_get_sensors**(int *num_sensors, const struct *sensing_sensor_info* **info)

Get all supported sensor instances' information.

This API just returns read only information of sensor instances, pointer info will directly point to internal buffer, no need for caller to allocate buffer, no side effect to sensor instances.

Parameters

- **num_sensors** – Get number of sensor instances.
- **info** – For receiving sensor instances' information array pointer.

Returns

0 on success or negative error value on failure.

int **sensing_open_sensor**(const struct *sensing_sensor_info* *info, struct *sensing_callback_list* *cb_list, *sensing_sensor_handle_t* *handle)

Open sensor instance by sensing sensor info.

Application clients use it to open a sensor instance and get its handle. Support multiple Application clients for open same sensor instance, in this case, the returned handle will different for different clients. meanwhile, also register sensing callback list

Parameters

- **info** – The sensor info got from *sensing_get_sensors*
- **cb_list** – callback list to be registered to sensing, must have a static lifetime.
- **handle** – The opened instance handle, if failed will be set to NULL.

Returns

0 on success or negative error value on failure.

int **sensing_open_sensor_by_dt**(const struct *device* *dev, struct *sensing_callback_list* *cb_list, *sensing_sensor_handle_t* *handle)

Open sensor instance by device.

Application clients use it to open a sensor instance and get its handle. Support multiple Application clients for open same sensor instance, in this case, the returned handle will different for different clients. meanwhile, also register sensing callback list.

Parameters

- `dev` – pointer device get from device tree.
- `cb_list` – callback list to be registered to sensing, must have a static lifetime.
- `handle` – The opened instance handle, if failed will be set to NULL.

Returns
0 on success or negative error value on failure.

int `sensing_close_sensor`(*sensing_sensor_handle_t* *handle)

Close sensor instance.

Parameters

- `handle` – The sensor instance handle need to close.

Returns
0 on success or negative error value on failure.

int `sensing_set_config`(*sensing_sensor_handle_t* handle, struct *sensing_sensor_config* *configs, int count)

Set current config items to Sensing subsystem.

Parameters

- `handle` – The sensor instance handle.
- `configs` – The configs to be set according to config attribute.
- `count` – count of configs.

Returns
0 on success or negative error value on failure, not support etc.

int `sensing_get_config`(*sensing_sensor_handle_t* handle, struct *sensing_sensor_config* *configs, int count)

Get current config items from Sensing subsystem.

Parameters

- `handle` – The sensor instance handle.
- `configs` – The configs to be get according to config attribute.
- `count` – count of configs.

Returns
0 on success or negative error value on failure, not support etc.

const struct *sensing_sensor_info* *`sensing_get_sensor_info`(*sensing_sensor_handle_t* handle)

Get sensor information from sensor instance handle.

Parameters

- `handle` – The sensor instance handle.

Returns
a const pointer to *sensing_sensor_info* on success or NULL on failure.

struct `sensing_sensor_version`

#include <*sensing.h*> Sensor Version.

Public Members

uint32_t **value**
The version represented as a 32-bit value.

uint8_t **major**
The major version number.

uint8_t **minor**
The minor version number.

uint8_t **hotfix**
The hotfix version number.

uint8_t **build**
The build version number.

struct **sensing_sensor_info**
#include <*sensing.h*> Sensor basic constant information.

Public Members

const char ***name**
Name of the sensor instance.

const char ***friendly_name**
Friendly name of the sensor instance.

const char ***vendor**
Vendor name of the sensor instance.

const char ***model**
Model name of the sensor instance.

const int32_t **type**
Sensor type.

const uint32_t **minimal_interval**
Minimal report interval in micro seconds.

struct **sensing_callback_list**
#include <*sensing.h*> Sensing subsystem event callback list.

Public Members

sensing_data_event_t **on_data_event**
Callback function for a sensor data event.

void *context

Associated context with on_data_event.

struct sensing_sensor_config

#include <sensing.h> Sensing subsystem sensor configure, including interval, sensitivity, latency.

Public Members

enum *sensing_sensor_attribute* attri

Attribute of the sensor configuration.

int8_t data_field

SENSING_SENSITIVITY_INDEX_ALL

Data field of the sensor configuration.

uint32_t interval

Interval between two sensor samples in microseconds (us).

uint32_t sensitivity

Sensitivity threshold for reporting new data.

A new sensor sample is reported only if the difference between it and the previous sample exceeds this sensitivity value.

uint64_t latency

Maximum duration for batching sensor samples before reporting in microseconds (us).

This defines how long sensor samples can be accumulated before they must be reported.

group sensing_sensor

Sensing Sensor API.

Defines

SENSING_SENSORS_DT_DEFINE(node, reg_ptr, cb_list_ptr, init_fn, pm_device, data_ptr, cfg_ptr, level, prio, api_ptr, ...)

Like *SENSOR_DEVICE_DT_DEFINE()* with sensing specifics.

Defines a sensor which implements the sensor API. May define an element in the sensing sensor iterable section used to enumerate all sensing sensors.

Parameters

- node – The devicetree node identifier.
- reg_ptr – Pointer to the device's *sensing_sensor_register_info*.
- cb_list_ptr – Pointer to sensing callback list.
- init_fn – Name of the init function of the driver.

- **pm_device** – PM device resources reference (NULL if device does not use PM).
- **data_ptr** – Pointer to the device's private data.
- **cfg_ptr** – The address to the structure containing the configuration information for this instance of the driver.
- **level** – The initialization level. See SYS_INIT() for details.
- **prio** – Priority within the selected initialization level. See SYS_INIT() for details.
- **api_ptr** – Provides an initial pointer to the API function struct used by the driver. Can be NULL.

SENSING_SENSORS_DT_INST_DEFINE(inst, ...)

Like *SENSING_SENSORS_DT_DEFINE()* for an instance of a DT_DRV_COMPAT compatible.

Parameters

- **inst** – instance number. This is replaced by `DT_DRV_COMPAT(inst)` in the call to *SENSING_SENSORS_DT_DEFINE()*.
- **...** – other parameters as expected by *SENSING_SENSORS_DT_DEFINE()*.

Functions

int **sensing_sensor_get_reporters**(const struct *device* *dev, int type, *sensing_sensor_handle_t* *reporter_handles, int max_handles)

Get reporter handles of a given sensor instance by sensor type.

Parameters

- **dev** – The sensor instance device structure.
- **type** – The given type, *SENSING_SENSOR_TYPE_ALL* to get reporters with all types.
- **max_handles** – The max count of the `reporter_handles` array input. Can get real count number via *sensing_sensor_get_reporters_count*
- **reporter_handles** – Input handles array for receiving found reporter sensor instances

Returns

number of reporters found, 0 returned if not found.

int **sensing_sensor_get_reporters_count**(const struct *device* *dev, int type)

Get reporters count of a given sensor instance by sensor type.

Parameters

- **dev** – The sensor instance device structure.
- **type** – The sensor type for checking, *SENSING_SENSOR_TYPE_ALL*

Returns

Count of reporters by `type`, 0 returned if no reporters by `type`.

int **sensing_sensor_get_state**(const struct *device* *dev, enum *sensing_sensor_state* *state)

Get this sensor's state.

Parameters

- **dev** – The sensor instance device structure.
- **state** – Returned sensor state value

Returns
0 on success or negative error value on failure.

struct **sensing_sensor_register_info**

#include <*sensing_sensor.h*> Sensor registration information.

Public Members

uint16_t **flags**

Sensor flags.

uint16_t **sample_size**

Sample size in bytes for a single sample of the registered sensor.

sensing runtime need this information for internal buffer allocation.

uint8_t **sensitivity_count**

The number of sensor sensitivities.

struct *sensing_sensor_version* **version**

Sensor version.

Version can be used to identify different versions of sensor implementation.

4.28 Task Watchdog

4.28.1 Overview

Many microcontrollers feature a hardware watchdog timer peripheral. Its purpose is to trigger an action (usually a system reset) in case of severe software malfunctions. Once initialized, the watchdog timer has to be restarted ("fed") in regular intervals to prevent it from timing out. If the software got stuck and does not manage to feed the watchdog anymore, the corrective action is triggered to bring the system back to normal operation.

In real-time operating systems with multiple tasks running in parallel, a single watchdog instance may not be sufficient anymore, as it can be used for only one task. This software watchdog based on kernel timers provides a method to supervise multiple threads or tasks (called watchdog channels).

An existing hardware watchdog can be used as an optional fallback if the task watchdog itself or the scheduler has a malfunction.

The task watchdog uses a kernel timer as its backend. If configured properly, the timer ISR is never actually called during normal operation, as the timer is continuously updated in the feed calls.

It's currently not possible to have multiple instances of task watchdogs. Instead, the task watchdog API can be accessed globally to add or delete new channels without passing around a context or device pointer in the firmware.

The maximum number of channels is predefined via Kconfig and should be adjusted to match exactly the number of channels required by the application.

4.28.2 Configuration Options

Related configuration options can be found under subsys/task_wdt/Kconfig.

- CONFIG_TASK_WDT
- CONFIG_TASK_WDT_CHANNELS
- CONFIG_TASK_WDT_HW_FALLBACK
- CONFIG_TASK_WDT_MIN_TIMEOUT
- CONFIG_TASK_WDT_HW_FALLBACK_DELAY

4.28.3 API Reference

Related code samples

Task watchdog
Monitor a thread using a task watchdog.

group `task_wdt_api`

Task Watchdog APIs.

Since
2.5

Version
0.8.0

Typedefs

typedef void (***task_wdt_callback_t**)(int channel_id, void *user_data)

Task watchdog callback.

Functions

int **task_wdt_init**(const struct *device* *hw_wdt)

Initialize task watchdog.

This function sets up necessary kernel timers and the hardware watchdog (if desired as fallback). It has to be called before *task_wdt_add()* and *task_wdt_feed()*.

Parameters

- **hw_wdt** – Pointer to the hardware watchdog device used as fallback. Pass NULL if no hardware watchdog fallback is desired.

Return values

- **0** – If successful.
- **-ENOTSUP** – If assigning a hardware watchdog is not supported.
- **-Errno** – Negative errno if the fallback hw_wdt is used and the install timeout API fails. See *wdt_install_timeout()* API for possible return values.

int **task_wdt_add**(uint32_t reload_period, *task_wdt_callback_t* callback, void *user_data)

Install new timeout.

Adds a new timeout to the list of task watchdog channels.

Parameters

- **reload_period** – Period in milliseconds used to reset the timeout
- **callback** – Function to be called when watchdog timer expired. Pass NULL to use system reset handler.
- **user_data** – User data to associate with the watchdog channel.

Return values

- **channel_id** – If successful, a non-negative value indicating the index of the channel to which the timeout was assigned. This ID is supposed to be used as the parameter in calls to *task_wdt_feed()*.
- **-EINVAL** – If the reload_period is invalid.
- **-ENOMEM** – If no more timeouts can be installed.

int **task_wdt_delete**(int channel_id)

Delete task watchdog channel.

Deletes the specified channel from the list of task watchdog channels. The channel is now available again for other tasks via *task_wdt_add()* function.

Parameters

- **channel_id** – Index of the channel as returned by *task_wdt_add()*.

Return values

- **0** – If successful.
- **-EINVAL** – If there is no installed timeout for supplied channel.

int **task_wdt_feed**(int channel_id)

Feed specified watchdog channel.

This function loops through all installed task watchdogs and updates the internal kernel timer used as for the software watchdog with the next due timeout.

Parameters

- **channel_id** – Index of the fed channel as returned by *task_wdt_add()*.

Return values

- **0** – If successful.
- **-EINVAL** – If there is no installed timeout for supplied channel.

4.29 Trusted Firmware-M

4.29.1 Trusted Firmware-M Overview

Trusted Firmware-M (TF-M) is a reference implementation of the Platform Security Architecture (PSA) IoT Security Framework. It defines and implements an architecture and a set of software components that aim to address some of the main security concerns in IoT products.

Zephyr RTOS has been PSA Certified since Zephyr 2.0.0 with TF-M 1.0, and is currently integrated with TF-M 2.1.0.

What Does TF-M Offer?

Through a set of secure services and by design, TF-M provides:

- Isolation of secure and non-secure resources
- Embedded-appropriate crypto
- Management of device secrets (keys, etc.)
- Firmware verification (and encryption)
- Protected off-chip data storage and retrieval
- Proof of device identity (device attestation)
- Audit logging

Build System Integration

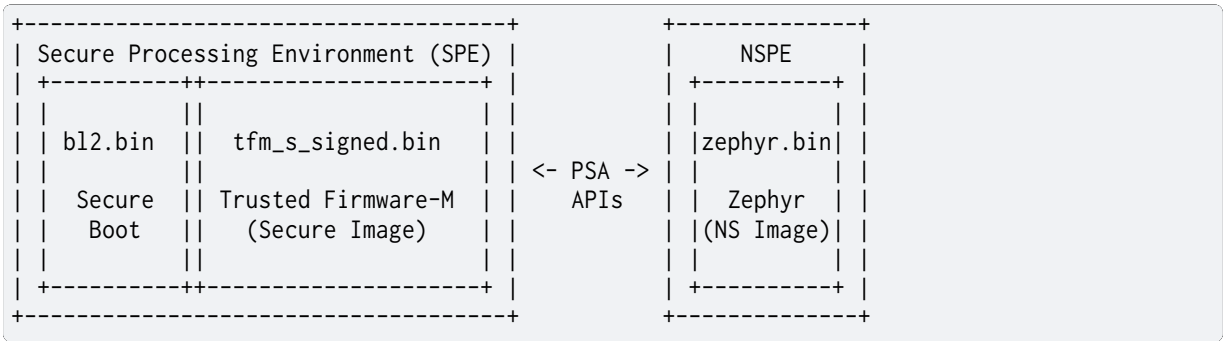
When using TF-M with a supported platform, TF-M will be automatically built and link in the background as part of the standard Zephyr build process. This build process makes a number of assumptions about how TF-M is being used, and has certain implications about what the Zephyr application image can and can not do:

- The secure processing environment (secure boot and TF-M) starts first
- Resource allocation for Zephyr relies on choices made in the secure image.

Architecture Overview

A TF-M application will, generally, have the following three parts, from most to least trusted, left-to-right, with code execution happening in the same order (secure boot > secure image > ns image).

While the secure bootloader is optional, it is enabled by default, and secure boot is an important part of providing a secure solution:



Communication between the (Zephyr) Non-Secure Processing Environment (NSPE) and the (TF-M) Secure Processing Environment image happens based on a set of PSA APIs, and normally makes use of an IPC mechanism that is included as part of the TF-M build, and implemented in Zephyr (see modules/trusted-firmware-m/interface).

Root of Trust (RoT) Architecture TF-M is based upon a **Root of Trust (RoT)** architecture. This allows for hierarchies of trust from most, to less, to least trusted, providing a sound foundation upon which to build or access trusted services and resources.

The benefit of this approach is that less trusted components are prevented from accessing or compromising more critical parts of the system, and error conditions in less trusted environments won't corrupt more trusted, isolated resources.

The following RoT hierarchy is defined for TF-M, from most to least trusted:

- PSA Root of Trust (**PRoT**), which consists of:
  - PSA Immutable Root of Trust: secure boot
  - PSA Updateable Root of Trust: most trusted secure services
- Application Root of Trust (**ARoT**): isolated secure services

The **PSA Immutable Root of Trust** is the most trusted piece of code in the system, to which subsequent Roots of Trust are anchored. In TF-M, this is the secure boot image, which verifies that the secure and non-secure images are valid, have not been tampered with, and come from a reliable source. The secure bootloader also verifies new images during the firmware update process, thanks to the public signing key(s) built into it. As the name implies, this image is **immutable**.

The **PSA Updateable Root of Trust** implements the most trusted secure services and components in TF-M, such as the Secure Partition Manager (SPM), and shared secure services like PSA Crypto, Internal Trusted Storage (ITS), etc. Services in the PSA Updateable Root of Trust have access to other resources in the same Root of Trust.

The **Application Root of Trust** is a reduced-privilege area in the secure processing environment which, depending on the isolation level chosen when building TF-M, has limited access to the PRoT, or even other ARoT services at the highest isolation levels. Some standard services exist in the ARoT, such as Protected Storage (PS), and generally custom secure services that you implement should be placed in the ARoT, unless a compelling reason is present to place them in the PRoT.

These divisions are distinct from the **untrusted code**, which runs in the non-secure environment, and has the least privilege in the system. This is the Zephyr application image in this case.

Isolation Levels At present, there are three distinct **isolation levels** defined in TF-M, with increasingly rigid boundaries between regions. The isolation level used will depend on your security requirements, and the system resources available to you.

- **Isolation Level 1** is the lowest isolation level, and the only major boundary is between the secure and non-secure processing environment, usually by means of Arm TrustZone on Armv8-M processors. There is no distinction here between the PSA Updateable Root of Trust (PRoT) and the Application Root of Trust (ARoT). They execute at the same privilege level. This isolation level will lead to the smallest combined application images.
- **Isolation Level 2** builds upon level one by introducing a distinction between the PSA Updateable Root of Trust and the Application Root of Trust, where ARoT services have limited access to PRoT services, and can only communicate with them through public APIs exposed by the PRoT services. ARoT services, however, are not strictly isolated from one another.
- **Isolation Level 3** is the highest isolation level, and builds upon level 2 by isolating ARoT services from each other, so that each ARoT is essentially silo'ed from other services. This provides the highest level of isolation, but also comes at the cost of additional overhead and code duplication between services.

The current isolation level can be checked via `CONFIG_TFM_ISOLATION_LEVEL`.

Secure Boot The default secure bootloader in TF-M is based on MCUBoot, and is referred to as `BL2` in TF-M (for the second-stage bootloader, potentially after a HW-based bootloader on the secure MCU, etc.).

All images in TF-M are hashed and signed, with the hash and signature verified by MCUBoot during the firmware update process.

Some key features of MCUBoot as used in TF-M are:

- Public signing key(s) are baked into the bootloader

- S and NS images can be signed using different keys
- Firmware images can optionally be encrypted
- Client software is responsible for writing a new image to the secondary slot
- By default, uses static flash layout of two identically-sized memory regions
- Optional security counter for rollback protection

When dealing with (optionally) encrypted images:

- Only the payload is encrypted (header, TLVs are plain text)
- Hashing and signing are applied over the un-encrypted data
- Uses `AES-CTR-128` or `AES-CTR-256` for encryption
- Encryption key randomized every encryption cycle (via `imgtool`)
- The `AES-CTR` key is included in the image and can be encrypted using:
  - `RSA-OAEP`
  - `AES-KW` (128 or 256 bits depending on the `AES-CTR` key length)
  - `ECIES-P256`
  - `ECIES-X25519`

Key config properties to control secure boot in Zephyr are:

- `CONFIG_TFM_BL2` toggles the bootloader (default = `y`).
- `CONFIG_TFM_KEY_FILE_S` overrides the secure signing key.
- `CONFIG_TFM_KEY_FILE_NS` overrides the non-secure signing key.

Secure Processing Environment Once the secure bootloader has finished executing, a TF-M based secure image will begin execution in the **secure processing environment**. This is where our device will be initially configured, and any secure services will be initialised.

Note that the starting state of our device is controlled by the secure firmware, meaning that when the non-secure Zephyr application starts, peripherals may not be in the HW-default reset state. In case of doubts, be sure to consult the board support packages in TF-M, available in the `platform/ext/target/` folder of the TF-M module (which is in `modules/tee/tf-m/trusted-firmware-m/` within a default Zephyr west workspace.)

Secure Services As of TF-M 1.8.0, the following secure services are generally available (although vendor support may vary):

- Crypto
- Firmware Update (FWU)
- Initial Attestation
- Platform
- Secure Storage, which has two parts:
  - Internal Trusted Storage (ITS)
  - Protected Storage (PS)

A template also exists for creating your own custom services.

For full details on these services, and their exposed APIs, please consult the TF-M Documentation.

Key Management and Derivation Key and secret management is a critical part of any secure device. You need to ensure that key material is available to regions that require it, but not to anything else, and that it is stored securely in a way that makes it difficult to tamper with or maliciously access.

The **Internal Trusted Storage** service in TF-M is used by the **PSA Crypto** service (which itself makes use of mbedtls) to store keys, and ensure that private keys are only ever accessible to the secure processing environment. Crypto operations that make use of key material, such as when signing payloads or when decrypting sensitive data, all take place via key handles. At no point should the key material ever be exposed to the NS environment.

One exception is that private keys can be provisioned into the secure processing environment as a one-way operation, such as during a factory provisioning process, but even this should be avoided where possible, and a request should be made to the SPE (via the PSA Crypto service) to generate a new private key itself, and the public key for that can be requested during provisioning and logged in the factory. This ensures the private key material is never exposed, or even known during the provisioning phase.

TF-M also makes extensive use of the **Hardware Unique Key (HUK)**, which every TF-M device must provide. This device-unique key is used by the **Protected Storage** service, for example, to encrypt information stored in external memory. For example, this ensures that the contents of flash memory can't be decrypted if they are removed and placed on a new device, since each device has its own unique HUK used while encrypting the memory contents the first time.

HUKs provide an additional advantage for developers, in that they can be used to derive new keys, and the **derived keys** don't need to be stored since they can be regenerated from the HUK at startup, using an additional salt/seed value (depending on the key derivation algorithm used). This removes the storage issue and a frequent attack vector. The HUK itself it usually highly protected in secure devices, and inaccessible directly by users.

`TFM_CRYPTO_ALG_HUK_DERIVATION` identifies the default key derivation algorithm used if a software implementation is used. The current default algorithm is `HKDF` (RFC 5869) with a SHA-256 hash. Other hardware implementations may be available on some platforms.

Non-Secure Processing Environment Zephyr is used for the NSPE, using a board that is supported by TF-M where the `CONFIG_BUILD_WITH_TFM` flag has been enabled.

Generally, you simply need to select the `*_ns` variant of a valid target (for example `mps2_an521_ns`), which will configure your Zephyr application to run in the NSPE, correctly build and link it with the TF-M secure images, sign the secure and non-secure images, and merge the three binaries into a single `tfm_merged.hex` file. The *west flash* command will flash `tfm_merged.hex` by default in this configuration.

At present, Zephyr can not be configured to be used as the secure processing environment.

4.29.2 TF-M Requirements

The following are some of the boards that can be used with TF-M:

Board	NSPE board name
mps2_an521_board	mps2_an521_ns (qemu supported)
mps3_an547_board	mps3_an547_ns (qemu supported)
bl5340_dvk	bl5340_dvk/nrf5340/cpuapp/ns
lpcxpresso55s69	lpcxpresso55s69_ns
nrf9160dk_nrf9160	nrf9160dk/nrf9160/ns
nrf5340dk_nrf5340	nrf5340dk/nrf5340/cpuapp/ns
b_u585i_iot02a_board	b_u585i_iot02a/stm32u585xx/ns
nucleo_l552ze_q_board	nucleo_l552ze_q/stm32l552xx/ns
stm32l562e_dk_board	stm32l562e_dk/stm32l562xx/ns
v2m_musca_b1_board	v2m_musca_b1_ns
v2m_musca_s1_board	v2m_musca_s1_ns

You can run `west boards -n _ns$` to search for non-secure variants of different board targets. To make sure TF-M is supported for a board in its output, check that `CONFIG_TRUSTED_EXECUTION_NONSECURE` is set to y in that board's default configuration.

Software Requirements

The following Python modules are required when building TF-M binaries:

- cryptography
- pyasn1
- pyyaml
- cbor>=1.0.0
- imgtool>=1.9.0
- jinja2
- click

You can install them via:

```
$ pip3 install --user cryptography pyasn1 pyyaml cbor>=1.0.0 imgtool>=1.9.0␣
↪jinja2 click
```

They are used by TF-M's signing utility to prepare firmware images for validation by the bootloader.

Part of the process of generating binaries for QEMU and merging signed secure and non-secure binaries on certain platforms also requires the use of the `srec_cat` utility.

This can be installed on Linux via:

```
$ sudo apt-get install srecord
```

And on OS X via:

```
$ brew install srecord
```

For Windows-based systems, please make sure you have a copy of the utility available on your system path. See, for example: SRecord for Windows

4.29.3 TF-M Build System

When building a valid _ns board target, TF-M will be built in the background, and linked with the Zephyr non-secure application. No knowledge of TF-M's build system is required in most cases, and the following will build a TF-M and Zephyr image pair, and run it in qemu with no additional steps required:

```
$ west build -p auto -b mps2_an521_ns samples/tfm_integration/psa_protected_
↪storage/ -t run
```

The outputs and certain key steps in this build process are described here, however, since you will need to understand and interact with the outputs, and deal with signing the secure and non-secure images before deploying them.

Images Created by the TF-M Build

The TF-M build system creates the following executable files:

- tfm_s - TF-M secure firmware
- tfm_ns - TF-M non-secure app (only used by regression tests).
- bl2 - TF-M MCUboot, if enabled

For each of these, it creates .bin, .hex, .elf, and .axf files.

The TF-M build system also creates signed variants of tfm_s and tfm_ns, and a file which combines them:

- tfm_s_signed
- tfm_ns_signed
- tfm_s_ns_signed

For each of these, only .bin files are created.

The TF-M non-secure app is discarded in favor of Zephyr non-secure app except when running the TF-M regression test suite.

The Zephyr build system usually signs both tfm_s and the Zephyr non-secure app itself. See below for details.

The 'tfm' target contains properties for all these paths. For example, the following will resolve to <path>/tfm_s.hex:

```
$<TARGET_PROPERTY:tfm,TFM_S_HEX_FILE>
```

See the top level CMakeLists.txt file in the tfm module for an overview of all the properties.

Signing Images

When CONFIG_TFM_BL2 is set to y, TF-M uses a secure bootloader (BL2) and firmware images must be signed with a private key. The firmware image is validated by the bootloader during updates using the corresponding public key, which is stored inside the secure bootloader firmware image.

By default, <tfm-dir>/bl2/ext/mcuboot/root-rsa-3072.pem is used to sign secure images, and <tfm-dir>/bl2/ext/mcuboot/root-rsa-3072_1.pem is used to sign non-secure images. These default .pem keys can (and **should**) be overridden using the CONFIG_TFM_KEY_FILE_S and CONFIG_TFM_KEY_FILE_NS config flags.

To satisfy PSA Certified Level 1 requirements, **You MUST replace the default .pem file with a new key pair!**

To generate a new public/private key pair, run the following commands:

```
$ imgtool keygen -k root-rsa-3072_s.pem -t rsa-3072
$ imgtool keygen -k root-rsa-3072_ns.pem -t rsa-3072
```

You can then place the new .pem files in an alternate location, such as your Zephyr application folder, and reference them in the `prj.conf` file via the `CONFIG_TFM_KEY_FILE_S` and `CONFIG_TFM_KEY_FILE_NS` config flags.

> **Warning**
>
> Be sure to keep your private key file in a safe, reliable location! If you lose this key file, you will be unable to sign any future firmware images, and it will no longer be possible to update your devices in the field!

After the built-in signing script has run, it creates a `tfm_merged.hex` file that contains all three binaries: bl2, tfm_s, and the zephyr app. This hex file can then be flashed to your development board or run in QEMU.

Custom CMake arguments When building a Zephyr application with TF-M it might be necessary to control the CMake arguments passed to the TF-M build.

Zephyr TF-M build offers several Kconfig options for controlling the build, but doesn't cover every CMake argument supported by the TF-M build system.

The `TFM_CMAKE_OPTIONS` property on the `zephyr_property_target` can be used to pass custom CMake arguments to the TF-M build system.

To pass the CMake argument `-DFOO=bar` to the TF-M build system, place the following CMake snippet in your CMakeLists.txt file.

```
set_property(TARGET zephyr_property_target
             APPEND PROPERTY TFM_CMAKE_OPTIONS
             -DFOO=bar
)
```

> **Note**
>
> The `TFM_CMAKE_OPTIONS` is a list so it is possible to append multiple options. Also CMake generator expressions are supported, such as `$<1:-DFOO=bar>`

Since `TFM_CMAKE_OPTIONS` is a list argument it will be expanded before it is passed to the TF-M build system. Options that have list arguments must therefore be properly escaped to avoid being expanded as a list.

```
set_property(TARGET zephyr_property_target
             APPEND PROPERTY TFM_CMAKE_OPTIONS
             -DFOO="bar\\\;baz"
)
```

Footprint and Memory Usage

The build system offers targets to view and analyse RAM and ROM usage in generated images. The tools run on the final images and give information about size of symbols and code being used in both RAM and ROM. For more information on these tools look here: *Footprint and Memory Usage*

Use the `tfm_ram_report` to get the RAM report for TF-M secure firmware (tfm_s).

Using west:

```
west build -b mps2_an521_ns samples/hello_world
west build -t tfm_ram_report
```

Using CMake and ninja:

```
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=mps2_an521_ns samples/hello_world

# Now run the build tool on the generated build system:
ninja -Cbuild tfm_ram_report
```

Use the `tfm_rom_report` to get the ROM report for TF-M secure firmware (tfm_s).

Using west:

```
west build -b mps2_an521_ns samples/hello_world
west build -t tfm_rom_report
```

Using CMake and ninja:

```
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=mps2_an521_ns samples/hello_world

# Now run the build tool on the generated build system:
ninja -Cbuild tfm_rom_report
```

Use the `bl2_ram_report` to get the RAM report for TF-M MCUboot, if enabled.

Using west:

```
west build -b mps2_an521_ns samples/hello_world
west build -t bl2_ram_report
```

Using CMake and ninja:

```
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=mps2_an521_ns samples/hello_world

# Now run the build tool on the generated build system:
ninja -Cbuild bl2_ram_report
```

Use the `bl2_rom_report` to get the ROM report for TF-M MCUboot, if enabled.

Using west:

```
west build -b mps2_an521_ns samples/hello_world
west build -t bl2_rom_report
```

Using CMake and ninja:

```
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=mps2_an521_ns samples/hello_world

# Now run the build tool on the generated build system:
ninja -Cbuild bl2_rom_report
```

4.29.4 Trusted Firmware-M Integration

The Trusted Firmware-M (TF-M) section contains information about the integration between TF-M and Zephyr RTOS. Use this information to help understand how to integrate TF-M with Zephyr for Cortex-M platforms and make use of its secure run-time services in Zephyr applications.

Board Definitions

TF-M will be built for the secure processing environment along with Zephyr if the `CONFIG_BUILD_WITH_TFM` flag is set to `y`.

Generally, this value should never be set at the application level, however, and all config flags required for TF-M should be set in a board variant with the `_ns` suffix.

This board variant must define an appropriate flash, SRAM and peripheral configuration that takes into account the initialisation process in the secure processing environment. `CONFIG_TFM_BOARD` must also be set via modules/trusted-firmware-m/Kconfig.tfm to the board name that TF-M expects for this target, so that it knows which target to build for the secure processing environment.

Example: `mps2_an521_ns` The `mps2_an521` target is a dual-core Arm Cortex-M33 evaluation board that, when using the default board variant, would generate a secure Zephyr binary.

The optional `mps2_an521_ns` target, however, sets these additional kconfig flags that indicate that Zephyr should be built as a non-secure image, linked with TF-M as an external project, and optionally the secure bootloader:

- `CONFIG_TRUSTED_EXECUTION_NONSECURE y`
- `CONFIG_ARM_TRUSTZONE_M y`

Comparing the `mps2_an521.dts` and `mps2_an521_ns.dts` files, we can see that the `_ns` version defines offsets in flash and SRAM memory, which leave the required space for TF-M and the secure bootloader:

```
reserved-memory {
            #address-cells = <1>;
            #size-cells = <1>;
            ranges;

            /* The memory regions defined below must match what the TF-M
             * project has defined for that board - a single image boot is
             * assumed. Please see the memory layout in:
             * https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/platform/
↪ext/target/mps2/an521/partition/flash_layout.h
             */

            code: memory@100000 {
                    reg = <0x00100000 DT_SIZE_K(512)>;
            };

            ram: memory@28100000 {
                    reg = <0x28100000 DT_SIZE_M(1)>;
            };
    };
```

This reserves 1 MB of code memory and 1 MB of RAM for secure boot and TF-M, such that our non-secure Zephyr application code will start at 0x10000, with RAM at 0x28100000. 512 KB code memory is available for the NS zephyr image, along with 1 MB of RAM.

This matches the flash memory layout we see in `flash_layout.h` in TF-M:

```
* 0x0000_0000 BL2 - MCUBoot (0.5 MB)
* 0x0008_0000 Secure image     primary slot (0.5 MB)
* 0x0010_0000 Non-secure image primary slot (0.5 MB)
* 0x0018_0000 Secure image     secondary slot (0.5 MB)
* 0x0020_0000 Non-secure image secondary slot (0.5 MB)
* 0x0028_0000 Scratch area (0.5 MB)
* 0x0030_0000 Protected Storage Area (20 KB)
* 0x0030_5000 Internal Trusted Storage Area (16 KB)
* 0x0030_9000 NV counters area (4 KB)
* 0x0030_A000 Unused (984 KB)
```

`mps2/an521` will be passed in to Tf-M as the board target, specified via `CONFIG_TFM_BOARD`.

4.29.5 Test Suites

TF-M includes two sets of test suites:

- tf-m-tests - Standard TF-M specific regression tests
- psa-arch-tests - Test suites for specific PSA APIs (secure storage, etc.)

These test suites can be run from Zephyr via an appropriate sample application in the samples/tfm_integration folder.

TF-M Regression Tests

The regression test suite can be run via the tfm_regression_test sample.

This sample tests various services and communication mechanisms across the NS/S boundary via the PSA APIs. They provide a useful sanity check for proper integration between the NS RTOS (Zephyr in this case) and the secure application (TF-M).

PSA Arch Tests

The PSA Arch Test suite, available via tfm_psa_test, contains a number of test suites that can be used to validate that PSA API specifications are being followed by the secure application, TF-M being an implementation of the Platform Security Architecture (PSA).

Only one of these suites can be run at a time, with the available test suites described via `CONFIG_TFM_PSA_TEST_*` KConfig flags:

Purpose

The output of these test suites is required to obtain PSA Certification for your specific board, RTOS (Zephyr here), and PSA implementation (TF-M in this case).

They also provide a useful test case to validate any PRs that make meaningful changes to TF-M, such as enabling a new TF-M board target, or making changes to the core TF-M module(s). They should generally be run as a coherence check before publishing a new PR for new board support, etc.

4.30 Virtualization

4.30.1 Inter-VM Shared Memory

- *Overview*
- *Support*
- *ivshmem-v2*
- *API Reference*

Overview

As Zephyr is enabled to run as a guest OS on Qemu and ACRN it might be necessary to make VMs aware of each other, or aware of the host. This is made possible by exposing a shared memory among parties via a feature called ivshmem, which stands for inter-VM Shared Memory.

The two types are supported: a plain shared memory (ivshmem-plain) or a shared memory with the ability for a VM to generate an interruption on another, and thus to be interrupted as well itself (ivshmem-doorbell).

Please refer to the official Qemu ivshmem documentation for more information.

Support

Zephyr supports both versions: plain and doorbell. Ivshmem driver can be built by enabling `CONFIG_IVSHMEM`. By default, this will expose the plain version. `CONFIG_IVSHMEM_DOORBELL` needs to be enabled to get the doorbell version.

Because the doorbell version uses MSI-X vectors to support notification vectors, the `CONFIG_IVSHMEM_MSI_X_VECTORS` has to be tweaked to the number of vectors that will be needed.

Note that a tiny shell module can be exposed to test the ivshmem feature by enabling `CONFIG_IVSHMEM_SHELL`.

ivshmem-v2

Zephyr also supports ivshmem-v2:

https://github.com/siemens/jailhouse/blob/master/Documentation/ivshmem-v2-specification.md

This is primarily used for IPC in the Jailhouse hypervisor (e.g. eth-ivshmem). It is also possible to use ivshmem-v2 without Jailhouse by building the Siemens fork of QEMU, and modifying the QEMU launch flags:

https://github.com/siemens/qemu/tree/wip/ivshmem2

API Reference

Related code samples

IVSHMEM doorbell
Use Inter-VM Shared Memory to exchange messages between two processes running on different operating systems.

Inter-VM Shared Memory (ivshmem) Ethernet
Communicate with another ”cell” in the Jailhouse hypervisor using IVSHMEM Ethernet.

group `ivshmem`

Inter-VM Shared Memory (ivshmem) reference API.

Defines

`IVSHMEM_V2_PROTO_UNDEFINED`

`IVSHMEM_V2_PROTO_NET`

Typedefs

typedef size_t (***ivshmem_get_mem_f**)(const struct *device* *dev, uintptr_t *memmap)

typedef uint32_t (***ivshmem_get_id_f**)(const struct *device* *dev)

typedef uint16_t (***ivshmem_get_vectors_f**)(const struct *device* *dev)

typedef int (***ivshmem_int_peer_f**)(const struct *device* *dev, uint32_t peer_id, uint16_t vector)

typedef int (***ivshmem_register_handler_f**)(const struct *device* *dev, struct *k_poll_signal* *signal, uint16_t vector)

Functions

size_t **ivshmem_get_mem**(const struct *device* *dev, uintptr_t *memmap)

Get the inter-VM shared memory.

Note: This API is not supported for ivshmem-v2, as the R/W and R/O areas may not be mapped contiguously. For ivshmem-v2, use the ivshmem_get_rw_mem_section, ivshmem_get_output_mem_section and ivshmem_get_state APIs to access the shared memory.

Parameters

- `dev` – Pointer to the device structure for the driver instance
- `memmap` – A pointer to fill in with the memory address

Returns
the size of the memory mapped, or 0

uint32_t **ivshmem_get_id**(const struct *device* *dev)

Get our VM ID.

Parameters

- `dev` – Pointer to the device structure for the driver instance

Returns
our VM ID or 0 if we are not running on doorbell version

uint16_t **ivshmem_get_vectors**(const struct *device* *dev)

Get the number of interrupt vectors we can use.

Parameters

- **dev** – Pointer to the device structure for the driver instance

Returns
the number of available interrupt vectors

int **ivshmem_int_peer**(const struct *device* *dev, uint32_t peer_id, uint16_t vector)

Interrupt another VM.

Parameters

- **dev** – Pointer to the device structure for the driver instance
- **peer_id** – The VM ID to interrupt
- **vector** – The interrupt vector to use

Returns
0 on success, a negative errno otherwise

int **ivshmem_register_handler**(const struct *device* *dev, struct *k_poll_signal* *signal, uint16_t vector)

Register a vector notification (interrupt) handler.

Note: The returned status, if positive, to a raised signal is the vector that generated the signal. This lets the possibility to the user to have one signal for all vectors, or one per-vector.

Parameters

- **dev** – Pointer to the device structure for the driver instance
- **signal** – A pointer to a valid and ready to be signaled struct *k_poll_signal*. Or NULL to unregister any handler registered for the given vector.
- **vector** – The interrupt vector to get notification from

Returns
0 on success, a negative errno otherwise

struct **ivshmem_driver_api**

#include <ivshmem.h>

4.31 Retention System

The retention system provides an API which allows applications to read and write data from and to memory areas or devices that retain the data while the device is powered. This allows for sharing information between different applications or within a single application without losing state information when a device reboots. The stored data should not persist in the event of a power failure (or during some low-power modes on some devices) nor should it be stored to a non-volatile storage like *Flash*, *EEPROM API*, or battery-backed RAM.

The retention system builds on top of the retained data driver, and adds additional software-level features to it for ensuring the validity of data. Optionally, a magic header can be used to check if the front of the retained data memory section contains this specific value, and an

optional checksum (1, 2, or 4-bytes in size) of the stored data can be appended to the end of the data. Additionally, the retention system API allows partitioning of the retained data sections into multiple distinct areas. For example, a 64-byte retained data area could be split up into 4 bytes for a boot mode, 16 bytes for a timestamp, 44 bytes for a last log message. All of these sections can be accessed or updated independently. The prefix and checksum can be set per-instance using devicetree.

4.31.1 Devicetree setup

To use the retention system, a retained data driver must be setup for the board you are using, there is a zephyr driver which can be used which will use some RAM as non-init for this purpose. The retention system is then initialised as a child node of this device 1 or more times - note that the memory region will need to be decremented to account for this reserved portion of RAM. See the following example (examples in this guide are based on the nrf52840dk_nrf52840 board and memory layout):

```
/ {
        sram@2003FC00 {
                compatible = "zephyr,memory-region", "mmio-sram";
                reg = <0x2003FC00 DT_SIZE_K(1)>;
                zephyr,memory-region = "RetainedMem";
                status = "okay";

                retainedmem {
                        compatible = "zephyr,retained-ram";
                        status = "okay";
                        #address-cells = <1>;
                        #size-cells = <1>;

                        /* This creates a 256-byte partition */
                        retention0: retention@0 {
                                compatible = "zephyr,retention";
                                status = "okay";

                                /* The total size of this area is 256
                                 * bytes which includes the prefix and
                                 * checksum, this means that the usable
                                 * data storage area is 256 - 3 = 253
                                 * bytes
                                 */
                                reg = <0x0 0x100>;

                                /* This is the prefix which must appear
                                 * at the front of the data
                                 */
                                prefix = [08 04];

                                /* This uses a 1-byte checksum */
                                checksum = <1>;
                        };

                        /* This creates a 768-byte partition */
                        retention1: retention@100 {
                                compatible = "zephyr,retention";
                                status = "okay";

                                /* Start position must be after the end
                                 * of the previous partition. The total
                                 * size of this area is 768 bytes which
                                 * includes the prefix and checksum,
```

(continues on next page)

(continued from previous page)

```
                                * this means that the usable data
                                * storage area is 768 - 6 = 762 bytes
                                */
                               reg = <0x100 0x300>;

                               /* This is the prefix which must appear
                                * at the front of the data
                                */
                               prefix = [00 11 55 88 fa bc];

                               /* If omitted, there will be no
                                * checksum
                                */
                        };
                };
        };
};

/* Reduce SRAM0 usage by 1KB to account for non-init area */
&sram0 {
        reg = <0x20000000 DT_SIZE_K(255)>;
};
```

The retention areas can then be accessed using the data retention API (once enabled with CONFIG_RETENTION, which requires that CONFIG_RETAINED_MEM be enabled) by getting the device by using:

```
#include <zephyr/device.h>
#include <zephyr/retention/retention.h>

const struct device *retention1 = DEVICE_DT_GET(DT_NODELABEL(retention1));
const struct device *retention2 = DEVICE_DT_GET(DT_NODELABEL(retention2));
```

When the write function is called, the magic header and checksum (if enabled) will be set on the area, and it will be marked as valid from that point onwards.

4.31.2 Mutex protection

Mutex protection of retention areas is enabled by default when applications are compiled with multithreading support. This means that different threads can safely call the retention functions without clashing with other concurrent thread function usage, but means that retention functions cannot be used from ISRs. It is possible to disable mutex protection globally on all retention areas by enabling CONFIG_RETENTION_MUTEX_FORCE_DISABLE - users are then responsible for ensuring that the function calls do not conflict with each other. Note that to use this, retention driver mutex support must also be disabled by enabling CONFIG_RETAINED_MEM_MUTEX_FORCE_DISABLE.

4.31.3 Boot mode

An addition to the retention subsystem is a boot mode interface, this can be used to dynamically change the state of an application or run a different application with a minimal set of functions when a device is rebooted (an example is to have a buttonless way of entering mcuboot's serial recovery feature from the main application).

To use the boot mode feature, a data retention entry must exist in the device tree, which is dedicated for use as the boot mode selection (the user area data size only needs to be a single byte), and this area be assigned to the chosen node of zephyr,boot-mode. See the following example:

```
/ {
        sram@2003FFFF {
                compatible = "zephyr,memory-region", "mmio-sram";
                reg = <0x2003FFFF 0x1>;
                zephyr,memory-region = "RetainedMem";
                status = "okay";

                retainedmem {
                        compatible = "zephyr,retained-ram";
                        status = "okay";
                        #address-cells = <1>;
                        #size-cells = <1>;

                        retention0: retention@0 {
                                compatible = "zephyr,retention";
                                status = "okay";
                                reg = <0x0 0x1>;
                        };
                };
        };

        chosen {
                zephyr,boot-mode = &retention0;
        };
};

/* Reduce SRAM0 usage by 1 byte to account for non-init area */
&sram0 {
        reg = <0x20000000 0x3FFFF>;
};
```

The boot mode interface can be enabled with `CONFIG_RETENTION_BOOT_MODE` and then accessed by using the boot mode functions. If using mcuboot with serial recovery, it can be built with `CONFIG_MCUBOOT_SERIAL` and `CONFIG_BOOT_SERIAL_BOOT_MODE` enabled which will allow rebooting directly into the serial recovery mode by using:

```
#include <zephyr/retention/bootmode.h>
#include <zephyr/sys/reboot.h>

bootmode_set(BOOT_MODE_TYPE_BOOTLOADER);
sys_reboot(0);
```

4.31.4 Retention system modules

Modules can expand the functionality of the retention system by using it as a transport (e.g. between a bootloader and application).

Bootloader Information

The bootloader information (abbreviated to blinfo) subsystem is an extension of the *Retention System* which allows for reading shared data from a bootloader and allowing applications to query it. It has an optional feature of organising the information retrieved from the bootloader and storing it in the *Settings* with the `blinfo/` prefix.

Devicetree setup To use the bootloader information subsystem, a retention area needs to be created which has a retained data section as its parent, generally non-init RAM is

used for this purpose. See the following example (examples in this guide are based on the nrf52840dk_nrf52840 board and memory layout):

```
/ {
        sram@2003F000 {
                compatible = "zephyr,memory-region", "mmio-sram";
                reg = <0x2003F000 DT_SIZE_K(1)>;
                zephyr,memory-region = "RetainedMem";
                status = "okay";

                retainedmem {
                        compatible = "zephyr,retained-ram";
                        status = "okay";
                        #address-cells = <1>;
                        #size-cells = <1>;

                        boot_info0: boot_info@0 {
                                compatible = "zephyr,retention";
                                status = "okay";
                                reg = <0x0 0x100>;
                        };
                };
        };

        chosen {
                zephyr,bootloader-info = &boot_info0;
        };
};


/* Reduce SRAM0 usage by 1KB to account for non-init area */
&sram0 {
        reg = <0x20000000 DT_SIZE_K(255)>;
};
```

Note that this configuration needs to be applied on both the bootloader (MCUboot) and application to be usable. It can be combined with other retention system APIs such as the *Boot mode*

MCUboot setup Once the above devicetree configuration is applied, MCUboot needs to be configured to store the shared data in this area, the following Kconfigs need to be set for this:

- `CONFIG_RETAINED_MEM` - Enables retained memory driver
- `CONFIG_RETENTION` - Enables retention system
- `CONFIG_BOOT_SHARE_DATA` - Enables shared data
- `CONFIG_BOOT_SHARE_DATA_BOOTINFO` - Enables boot information shared data type
- `CONFIG_BOOT_SHARE_BACKEND_RETENTION` - Stores shared data using retention/blinfo subsystem

Application setup The application must enable the following base Kconfig options for the bootloader information subsystem to function:

- `CONFIG_RETAINED_MEM`
- `CONFIG_RETENTION`
- `CONFIG_RETENTION_BOOTLOADER_INFO`
- `CONFIG_RETENTION_BOOTLOADER_INFO_TYPE_MCUBOOT`

The following include is needed to use the bootloader information subsystem:

```
#include <zephyr/retention/blinfo.h>
```

By default, only the lookup function is provided: *blinfo_lookup()*, the application can call this to query the information from the bootloader. This function is enabled by default with CONFIG_RETENTION_BOOTLOADER_INFO_OUTPUT_FUNCTION, however, applications can optionally choose to use the settings storage feature instead. In this mode, the bootloader information can be queries by using settings keys, the following Kconfig options need to be enabled for this mode:

- CONFIG_SETTINGS
- CONFIG_SETTINGS_RUNTIME
- CONFIG_RETENTION_BOOTLOADER_INFO_OUTPUT_SETTINGS

This allows the information to be queried via the *settings_runtime_get()* function with the following keys:

- blinfo/mode The mode that MCUboot is configured for (enum mcuboot_mode value)
- blinfo/signature_type The signature type MCUboot is configured for (enum mcuboot_signature_type value)
- blinfo/recovery The recovery type enabled in MCUboot (enum mcuboot_recovery_mode value)
- blinfo/running_slot The running slot, useful for direct-XIP mode to know which slot to use for an update
- blinfo/bootloader_version Version of the bootloader (struct image_version object)
- blinfo/max_application_size Maximum size of an application (in bytes) that can be loaded

In addition to the previous include, the following includes are required for this mode:

```
#include <bootutil/boot_status.h>
#include <bootutil/image.h>
#include <zephyr/mcuboot_version.h>
#include <zephyr/settings/settings.h>
```

API Reference

Bootloader information API

group **bootloader_info_interface**

Bootloader info interface.

Since
3.5

Version
0.1.0

Functions

int **blinfo_lookup**(uint16_t key, char *val, int val_len_max)

Returns bootinfo information.

Parameters

- **key** – The information to return (for MCUboot: minor TLV).

- `val` – Where the return information will be placed.
- `val_len_max` – The maximum size of the provided buffer.

Return values

- `>=` – 0 If successful (contains length of read value)
- `-EOVERFLOW` – If the data is too large to fit the supplied buffer.
- `-EIO` – If the requested key was not found.
- `-errno` – Error code.

4.31.5 API Reference

Retention system API

group `retention_api`

Retention API.

Since

3.4

Version

0.1.0

Typedefs

typedef ssize_t (*`retention_size_api`)(const struct *device* *dev)

typedef int (*`retention_is_valid_api`)(const struct *device* *dev)

typedef int (*`retention_read_api`)(const struct *device* *dev, off_t offset, uint8_t *buffer, size_t size)

typedef int (*`retention_write_api`)(const struct *device* *dev, off_t offset, const uint8_t *buffer, size_t size)

typedef int (*`retention_clear_api`)(const struct *device* *dev)

Functions

ssize_t `retention_size`(const struct *device* *dev)

Returns the size of the retention area.

Parameters

- `dev` – Retention device to use.

Return values

`Positive` – value indicating size in bytes on success, else negative errno code.

int `retention_is_valid`(const struct *device* *dev)

Checks if the underlying data in the retention area is valid or not.

Parameters

- `dev` – Retention device to use.

Return values

- `1` – If successful and data is valid.
- `0` – If data is not valid.
- `-ENOTSUP` – If there is no header/checksum configured for the retention area.
- `-errno` – Error code code.

int `retention_read`(const struct *device* *dev, off_t offset, uint8_t *buffer, size_t size)

Reads data from the retention area.

Parameters

- `dev` – Retention device to use.
- `offset` – Offset to read data from.
- `buffer` – Buffer to store read data in.
- `size` – Size of data to read.

Return values

- `0` – If successful.
- `-errno` – Error code code.

int `retention_write`(const struct *device* *dev, off_t offset, const uint8_t *buffer, size_t size)

Writes data to the retention area (underlying data does not need to be cleared prior to writing), once function returns with a success code, the data will be classed as valid if queried using *retention_is_valid()*.

Parameters

- `dev` – Retention device to use.
- `offset` – Offset to write data to.
- `buffer` – Data to write.
- `size` – Size of data to be written.

Return values

`0` – on success else negative errno code.

int `retention_clear`(const struct *device* *dev)

Clears all data in the retention area (sets it to 0)

Parameters

- `dev` – Retention device to use.

Return values

`0` – on success else negative errno code.

struct `retention_api`

#include <retention.h>

Boot mode interface

group `boot_mode_interface`

Boot mode interface.

Enums

enum `BOOT_MODE_TYPES`

Values:

enumerator `BOOT_MODE_TYPE_NORMAL` = 0x00

Default (normal) boot, to user application.

enumerator `BOOT_MODE_TYPE_BOOTLOADER`

Bootloader boot mode (e.g.

serial recovery for MCUboot)

Functions

int `bootmode_check`(uint8_t boot_mode)

Checks if the boot mode of the device is set to a specific value.

Parameters

- `boot_mode` – Expected boot mode to check.

Return values

- `1` – If successful and boot mode matches.
- `0` – If boot mode does not match.
- `-errno` – Error code code.

int `bootmode_set`(uint8_t boot_mode)

Sets boot mode of device.

Parameters

- `boot_mode` – Boot mode value to set.

Return values

- `0` – If successful.
- `-errno` – Error code code.

int `bootmode_clear`(void)

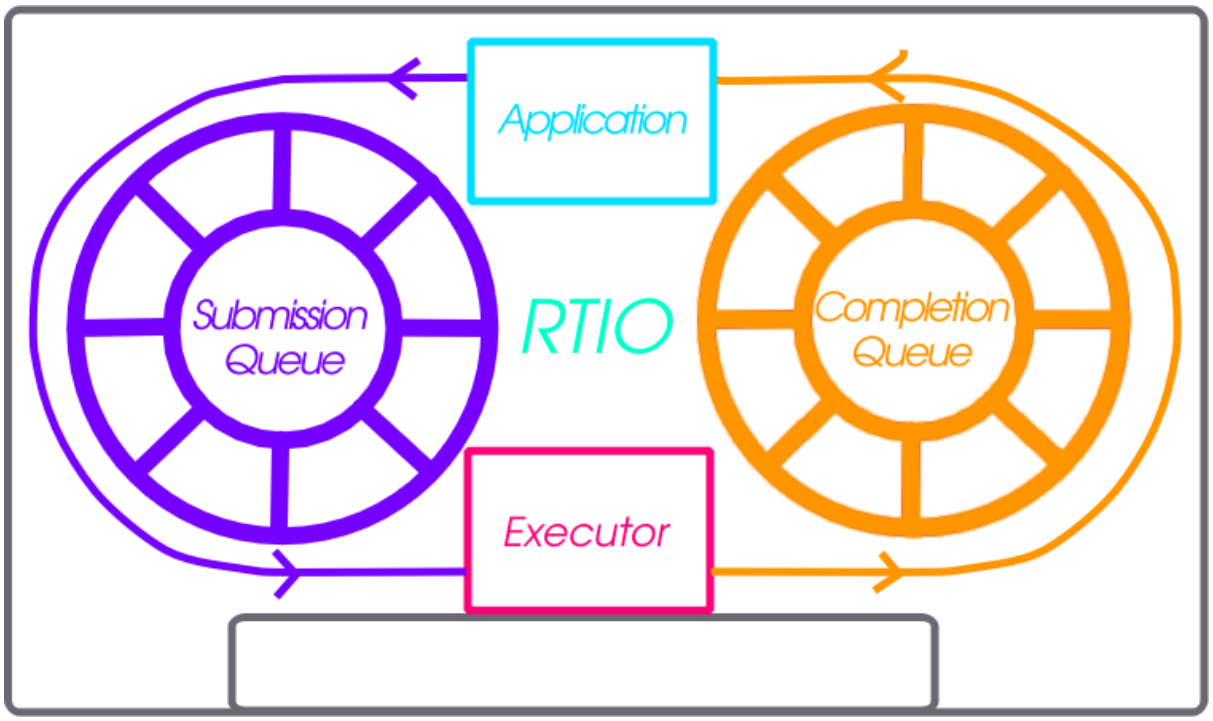
Clear boot mode value (sets to 0) - which corresponds to *BOOT_MODE_TYPE_NORMAL*.

Return values

- `0` – If successful.
- `-errno` – Error code code.

4.32 Real Time I/O (RTIO)

- *Problem*
- *Inspiration, introducing io_uring*
- *Submission Queue*
- *Completion Queue*
- *Executor*
- *IO Device*
- *Cancellation*
- *Memory pools*
- *When to Use*
- *API Reference*



RTIO provides a framework for doing asynchronous operation chains with event driven I/O. This section covers the RTIO API, queues, executor, iodev, and common usage patterns with peripheral devices.

RTIO takes a lot of inspiration from Linux's io_uring in its operations and API as that API matches up well with hardware transfer queues and descriptions such as DMA transfer lists.

4.32.1 Problem

An application wishing to do complex DMA or interrupt driven operations today in Zephyr requires direct knowledge of the hardware and how it works. There is no understanding in the DMA API of other Zephyr devices and how they relate.

This means doing complex audio, video, or sensor streaming requires direct hardware knowledge or leaky abstractions over DMA controllers. Neither is ideal.

To enable asynchronous operations, especially with DMA, a description of what to do rather than direct operations through C and callbacks is needed. Enabling DMA features such as channels with priority, and sequences of transfers requires more than a simple list of descriptions.

Using DMA and/or interrupt driven I/O shouldn't dictate whether or not the call is blocking or not.

4.32.2 Inspiration, introducing io_uring

It's better not to reinvent the wheel (or ring in this case) and io_uring as an API from the Linux kernel provides a winning model. In io_uring there are two lock-free ring buffers acting as queues shared between the kernel and a userland application. One queue for submission entries which may be chained and flushed to create concurrent sequential requests. A second queue for completion queue events. Only a single syscall is actually required to execute many operations, the io_uring_submit call. This call may block the caller when a number of operations to wait on is given.

This model maps well to DMA and interrupt driven transfers. A request to do a sequence of operations in an asynchronous way directly relates to the way hardware typically works with interrupt driven state machines potentially involving multiple peripheral IPs like bus and DMA controllers.

4.32.3 Submission Queue

The submission queue (sq), is the description of the operations to perform in concurrent chains.

For example imagine a typical SPI transfer where you wish to write a register address to then read from. So the sequence of operations might be...

1. Chip Select
2. Clock Enable
3. Write register address into SPI transmit register
4. Read from the SPI receive register into a buffer
5. Disable clock
6. Disable Chip Select

If anything in this chain of operations fails give up. Some of those operations can be embodied in a device abstraction that understands a read or write implicitly means setup the clock and chip select. The transactional nature of the request also needs to be embodied in some manner. Of the operations above perhaps the read could be done using DMA as its large enough make sense. That requires an understanding of how to setup the device's particular DMA to do so.

The above sequence of operations is embodied in RTIO as chain of submission queue entries (sqe). Chaining is done by setting a bitflag in an sqe to signify the next sqe must wait on the current one.

Because the chip select and clocking is common to a particular SPI controller and device on the bus it is embodied in what RTIO calls an iodev.

Multiple operations against the same iodev are done in the order provided as soon as possible. If two operation chains have varying points using the same device its possible one chain will have to wait for another to complete.

4.32.4 Completion Queue

In order to know when a sqe has completed there is a completion queue (cq) with completion queue events (cqe). A sqe once completed results in a cqe being pushed into the cq. The ordering of cqe may not be the same order of sqe. A chain of sqe will however ensure ordering and failure cascading.

Other potential schemes are possible but a completion queue is a well trod idea with io_uring and other similar operating system APIs.

4.32.5 Executor

The RTIO executor is a low overhead concurrent I/O task scheduler. It ensures certain request flags provide the expected behavior. It takes a list of submissions working through them in order. Various flags allow for changing the behavior of how submissions are worked through. Flags to form in order chains of submissions, transactional sets of submissions, or create multi-shot (continuously producing) requests are all possible!

4.32.6 IO Device

Turning submission queue entries (sqe) into completion queue events (cqe) is the job of objects implementing the iodev (IO device) API. This API accepts requests in the form of the iodev submit API call. It is the io devices job to work through its internal queue of submissions and convert them into completions. In effect every io device can be viewed as an independent, event driven actor like object, that accepts a never ending queue of I/O like requests. How the iodev does this work is up to the author of the iodev, perhaps the entire queue of operations can be converted to a set of DMA transfer descriptors, meaning the hardware does almost all of the real work.

4.32.7 Cancellation

Canceling an already queued operation is possible but not guaranteed. If the SQE has not yet started, it's likely that a call to *rtio_sqe_cancel()* will remove the SQE and never run it. If, however, the SQE already started running, the cancel request will be ignored.

4.32.8 Memory pools

In some cases requests to read may not know how much data will be produced. Alternatively, a reader might be handling data from multiple io devices where the frequency of the data is unpredictable. In these cases it may be wasteful to bind memory to in flight read requests. Instead with memory pools the memory to read into is left to the iodev to allocate from a memory pool associated with the RTIO context that the read was associated with. To create such an RTIO context the *RTIO_DEFINE_WITH_MEMPOOL* can be used. It allows creating an RTIO context with a dedicated pool of "memory blocks" which can be consumed by the iodev. Below is a snippet setting up the RTIO context with a memory pool. The memory pool has 128 blocks, each block has the size of 16 bytes, and the data is 4 byte aligned.

```
#include <zephyr/rtio/rtio.h>

#define SQ_SIZE       4
#define CQ_SIZE       4
#define MEM_BLK_COUNT 128
#define MEM_BLK_SIZE  16
#define MEM_BLK_ALIGN 4
```

(continues on next page)

(continued from previous page)

```
RTIO_DEFINE_WITH_MEMPOOL(rtio_context,
    SQ_SIZE, CQ_SIZE, MEM_BLK_COUNT, MEM_BLK_SIZE, MEM_BLK_ALIGN);
```

When a read is needed, the caller simply needs to replace the call *rtio_sqe_prep_read()* (which takes a pointer to a buffer and a length) with a call to *rtio_sqe_prep_read_with_pool()*. The iodev requires only a small change which works with both pre-allocated data buffers as well as the mempool. When the read is ready, instead of getting the buffers directly from the *rtio_iodev_sqe*, the iodev should get the buffer and count by calling *rtio_sqe_rx_buf()* like so:

```
uint8_t *buf;
uint32_t buf_len;
int rc = rtio_sqe_rx_buff(iodev_sqe, MIN_BUF_LEN, DESIRED_BUF_LEN, &buf, &buf_len);

if (rc != 0) {
  LOG_ERR("Failed to get buffer of at least %u bytes", MIN_BUF_LEN);
  return;
}
```

Finally, the consumer will be able to access the allocated buffer via *rtio_cqe_get_mempool_buffer()*.

```
uint8_t *buf;
uint32_t buf_len;
int rc = rtio_cqe_get_mempool_buffer(&rtio_context, &cqe, &buf, &buf_len);

if (rc != 0) {
  LOG_ERR("Failed to get mempool buffer");
  return rc;
}

/* Release the cqe events (note that the buffer is not released yet */
rtio_cqe_release_all(&rtio_context);

/* Do something with the memory */

/* Release the mempool buffer */
rtio_release_buffer(&rtio_context, buf);
```

4.32.9 When to Use

RTIO is useful in cases where concurrent or batch like I/O flows are useful.

From the driver/hardware perspective the API enables batching of I/O requests, potentially in an optimal way. Many requests to the same SPI peripheral for example might be translated to hardware command queues or DMA transfer descriptors entirely. Meaning the hardware can potentially do more than ever.

There is a small cost to each RTIO context and iodev. This cost could be weighed against using a thread for each concurrent I/O operation or custom queues and threads per peripheral. RTIO is much lower cost than that.

4.32.10 API Reference

group `rtio`

RTIO.

Since
3.2

Version
0.1.0

Defines

`RTIO_IODEV_I2C_STOP`
Equivalent to the I2C_MSG_STOP flag.

`RTIO_IODEV_I2C_RESTART`
Equivalent to the I2C_MSG_RESTART flag.

`RTIO_IODEV_I2C_10_BITS`
Equivalent to the I2C_MSG_ADDR_10_BITS.

`RTIO_OP_NOP`
An operation that does nothing and will complete immediately.

`RTIO_OP_RX`
An operation that receives (reads)

`RTIO_OP_TX`
An operation that transmits (writes)

`RTIO_OP_TINY_TX`
An operation that transmits tiny writes by copying the data to write.

`RTIO_OP_CALLBACK`
An operation that calls a given function (callback)

`RTIO_OP_TXRX`
An operation that transceives (reads and writes simultaneously)

`RTIO_OP_I2C_RECOVER`
An operation to recover I2C buses.

`RTIO_OP_I2C_CONFIGURE`
An operation to configure I2C buses.

`RTIO_IODEV_DEFINE`(name, iodev_api, iodev_data)
Statically define and initialize an RTIO IODev.

Parameters

- `name` – Name of the iodev
- `iodev_api` – Pointer to struct *rtio_iodev_api*

- **iodev_data** – Data pointer

RTIO_BMEM

Allocate to bss if available.

If CONFIG_USERSPACE is selected, allocate to the rtio_partition bss. Maps to: K_APP_BMEM(rtio_partition) static

If CONFIG_USERSPACE is disabled, allocate as plain static: static

RTIO_DMEM

Allocate as initialized memory if available.

If CONFIG_USERSPACE is selected, allocate to the rtio_partition init. Maps to: K_APP_DMEM(rtio_partition) static

If CONFIG_USERSPACE is disabled, allocate as plain static: static

RTIO_DEFINE(name, sq_sz, cq_sz)

Statically define and initialize an RTIO context.

Parameters

- **name** – Name of the RTIO
- **sq_sz** – Size of the submission queue entry pool
- **cq_sz** – Size of the completion queue entry pool

RTIO_DEFINE_WITH_MEMPOOL(name, sq_sz, cq_sz, num_blks, blk_size, balign)

Statically define and initialize an RTIO context.

Parameters

- **name** – Name of the RTIO
- **sq_sz** – Size of the submission queue, must be power of 2
- **cq_sz** – Size of the completion queue, must be power of 2
- **num_blks** – Number of blocks in the memory pool
- **blk_size** – The number of bytes in each block
- **balign** – The block alignment

Typedefs

typedef void (***rtio_callback_t**)(struct *rtio* *r, const struct *rtio_sqe* *sqe, void *arg0)

Callback signature for RTIO_OP_CALLBACK.

Param r
RTIO context being used with the callback

Param sqe
Submission for the callback op

Param arg0
Argument option as part of the sqe

Functions

static inline size_t **rtio_mempool_block_size**(const struct *rtio* *r)

Get the mempool block size of the RTIO context.

Parameters

- **r** – **[in]** The RTIO context

Returns
The size of each block in the context's mempool

Returns
0 if the context doesn't have a mempool

static inline void **rtio_sqe_prep_nop**(struct *rtio_sqe* *sqe, const struct *rtio_iodev* *iodev, void *userdata)

Prepare a nop (no op) submission.

static inline void **rtio_sqe_prep_read**(struct *rtio_sqe* *sqe, const struct *rtio_iodev* *iodev, int8_t prio, uint8_t *buf, uint32_t len, void *userdata)

Prepare a read op submission.

static inline void **rtio_sqe_prep_read_with_pool**(struct *rtio_sqe* *sqe, const struct *rtio_iodev* *iodev, int8_t prio, void *userdata)

Prepare a read op submission with context's mempool.

See also

rtio_sqe_prep_read()

static inline void **rtio_sqe_prep_read_multishot**(struct *rtio_sqe* *sqe, const struct *rtio_iodev* *iodev, int8_t prio, void *userdata)

static inline void **rtio_sqe_prep_write**(struct *rtio_sqe* *sqe, const struct *rtio_iodev* *iodev, int8_t prio, uint8_t *buf, uint32_t len, void *userdata)

Prepare a write op submission.

static inline void **rtio_sqe_prep_tiny_write**(struct *rtio_sqe* *sqe, const struct *rtio_iodev* *iodev, int8_t prio, const uint8_t *tiny_write_data, uint8_t tiny_write_len, void *userdata)

Prepare a tiny write op submission.

Unlike the normal write operation where the source buffer must outlive the call the tiny write data in this case is copied to the sqe. It must be tiny to fit within the specified size of a *rtio_sqe*.

This is useful in many scenarios with RTL logic where a write of the register to subsequently read must be done.

static inline void **rtio_sqe_prep_callback**(struct *rtio_sqe* *sqe, *rtio_callback_t* callback, void *arg0, void *userdata)

Prepare a callback op submission.

A somewhat special operation in that it may only be done in kernel mode.

Used where general purpose logic is required in a queue of io operations to do transforms or logic.

static inline void **rtio_sqe_prep_transceive**(struct *rtio_sqe* *sqe, const struct *rtio_iodev* *iodev, int8_t prio, uint8_t *tx_buf, uint8_t *rx_buf, uint32_t buf_len, void *userdata)

Prepare a transceive op submission.

static inline struct *rtio_iodev_sqe* ***rtio_sqe_pool_alloc**(struct *rtio_sqe_pool* *pool)

static inline void **rtio_sqe_pool_free**(struct *rtio_sqe_pool* *pool, struct *rtio_iodev_sqe* *iodev_sqe)

static inline struct *rtio_cqe* ***rtio_cqe_pool_alloc**(struct *rtio_cqe_pool* *pool)

static inline void **rtio_cqe_pool_free**(struct *rtio_cqe_pool* *pool, struct *rtio_cqe* *cqe)

static inline int **rtio_block_pool_alloc**(struct *rtio* *r, size_t min_sz, size_t max_sz, uint8_t **buf, uint32_t *buf_len)

static inline void **rtio_block_pool_free**(struct *rtio* *r, void *buf, uint32_t buf_len)

static inline uint32_t **rtio_sqe_acquirable**(struct *rtio* *r)

Count of acquirable submission queue events.

Parameters

- **r** – RTIO context

Returns

Count of acquirable submission queue events

static inline struct *rtio_iodev_sqe* ***rtio_txn_next**(const struct *rtio_iodev_sqe* *iodev_sqe)

Get the next sqe in the transaction.

Parameters

- **iodev_sqe** – Submission queue entry

Return values

- **NULL** – if current sqe is last in transaction
- **struct** – *rtio_sqe* * if available

static inline struct *rtio_iodev_sqe* ***rtio_chain_next**(const struct *rtio_iodev_sqe* *iodev_sqe)

Get the next sqe in the chain.

Parameters

- **iodev_sqe** – Submission queue entry

Return values

- **NULL** – if current sqe is last in chain
- **struct** – *rtio_sqe* * if available

static inline struct *rtio_iodev_sqe* ***rtio_iodev_sqe_next**(const struct *rtio_iodev_sqe* *iodev_sqe)

Get the next sqe in the chain or transaction.

Parameters

- **iodev_sqe** – Submission queue entry

Return values

- `NULL` – if current sqe is last in chain
- `struct` – *rtio_iodev_sqe* * if available

static inline struct *rtio_sqe* *`rtio_sqe_acquire`(struct *rtio* *r)

Acquire a single submission queue event if available.

Parameters

- `r` – RTIO context

Return values

- `sqe` – A valid submission queue event acquired from the submission queue
- `NULL` – No subsmission queue event available

static inline void `rtio_sqe_drop_all`(struct *rtio* *r)

Drop all previously acquired sqe.

Parameters

- `r` – RTIO context

static inline struct *rtio_cqe* *`rtio_cqe_acquire`(struct *rtio* *r)

Acquire a complete queue event if available.

static inline void `rtio_cqe_produce`(struct *rtio* *r, struct *rtio_cqe* *cqe)

Produce a complete queue event if available.

static inline struct *rtio_cqe* *`rtio_cqe_consume`(struct *rtio* *r)

Consume a single completion queue event if available.

If a completion queue event is returned rtio_cq_release(r) must be called at some point to release the cqe spot for the cqe producer.

Parameters

- `r` – RTIO context

Return values

- `cqe` – A valid completion queue event consumed from the completion queue
- `NULL` – No completion queue event available

static inline struct *rtio_cqe* *`rtio_cqe_consume_block`(struct *rtio* *r)

Wait for and consume a single completion queue event.

If a completion queue event is returned rtio_cq_release(r) must be called at some point to release the cqe spot for the cqe producer.

Parameters

- `r` – RTIO context

Return values

`cqe` – A valid completion queue event consumed from the completion queue

static inline void `rtio_cqe_release`(struct *rtio* *r, struct *rtio_cqe* *cqe)

Release consumed completion queue event.

Parameters

- `r` – RTIO context

- **cqe** – Completion queue entry

static inline uint32_t **rtio_cqe_compute_flags**(struct *rtio_iodev_sqe* *iodev_sqe)

Compute the CQE flags from the *rtio_iodev_sqe* entry.

Parameters

- **iodev_sqe** – The SQE entry in question.

Returns
The value that should be set for the CQE's flags field.

int **rtio_cqe_get_mempool_buffer**(const struct *rtio* *r, struct *rtio_cqe* *cqe, uint8_t **buff, uint32_t *buff_len)

Retrieve the mempool buffer that was allocated for the CQE.

If the RTIO context contains a memory pool, and the SQE was created by calling rtio_sqe_read_with_pool(), this function can be used to retrieve the memory associated with the read. Once processing is done, it should be released by calling *rtio_release_buffer()*.

Parameters

- **r** – **[in]** RTIO context
- **cqe** – **[in]** The CQE handling the event.
- **buff** – **[out]** Pointer to the mempool buffer
- **buff_len** – **[out]** Length of the allocated buffer

Returns
0 on success

Returns
-EINVAL if the buffer wasn't allocated for this cqe

Returns
-ENOTSUP if memory blocks are disabled

void **rtio_executor_submit**(struct *rtio* *r)

void **rtio_executor_ok**(struct *rtio_iodev_sqe* *iodev_sqe, int result)

void **rtio_executor_err**(struct *rtio_iodev_sqe* *iodev_sqe, int result)

static inline void **rtio_iodev_sqe_ok**(struct *rtio_iodev_sqe* *iodev_sqe, int result)

Inform the executor of a submission completion with success.

This may start the next asynchronous request if one is available.

Parameters

- **iodev_sqe** – IODev Submission that has succeeded
- **result** – Result of the request

static inline void **rtio_iodev_sqe_err**(struct *rtio_iodev_sqe* *iodev_sqe, int result)

Inform the executor of a submissions completion with error.

This SHALL fail the remaining submissions in the chain.

Parameters

- **iodev_sqe** – Submission that has failed
- **result** – Result of the request

static inline void **rtio_cqe_submit**(struct *rtio* *r, int result, void *userdata, uint32_t flags)

Submit a completion queue event with a given result and userdata.

Called by the executor to produce a completion queue event, no inherent locking is performed and this is not safe to do from multiple callers.

Parameters

- **r** – RTIO context
- **result** – Integer result code (could be -errno)
- **userdata** – Userdata to pass along to completion
- **flags** – Flags to use for the CEQ see RTIO_CQE_FLAG_*

static inline int **rtio_sqe_rx_buf**(const struct *rtio_iodev_sqe* *iodev_sqe, uint32_t min_buf_len, uint32_t max_buf_len, uint8_t **buf, uint32_t *buf_len)

Get the buffer associate with the RX submission.

Parameters

- **iodev_sqe** – **[in]** The submission to probe
- **min_buf_len** – **[in]** The minimum number of bytes needed for the operation
- **max_buf_len** – **[in]** The maximum number of bytes needed for the operation
- **buf** – **[out]** Where to store the pointer to the buffer
- **buf_len** – **[out]** Where to store the size of the buffer

Returns
0 if buf and buf_len were successfully filled

Returns
-ENOMEM Not enough memory for min_buf_len

void **rtio_release_buffer**(struct *rtio* *r, void *buff, uint32_t buff_len)

Release memory that was allocated by the RTIO's memory pool.

If the RTIO context was created by a call to *RTIO_DEFINE_WITH_MEMPOOL()*, then the cqe data might contain a buffer that's owned by the RTIO context. In those cases (if the read request was configured via rtio_sqe_read_with_pool()) the buffer must be returned back to the pool.

Call this function when processing is complete. This function will validate that the memory actually belongs to the RTIO context and will ignore invalid arguments.

Parameters

- **r** – RTIO context
- **buff** – Pointer to the buffer to be released.
- **buff_len** – Number of bytes to free (will be rounded up to nearest memory block).

static inline void **rtio_access_grant**(struct *rtio* *r, struct *k_thread* *t)

Grant access to an RTIO context to a user thread.

int **rtio_sqe_cancel**(struct *rtio_sqe* *sqe)

Attempt to cancel an SQE.

If possible (not currently executing), cancel an SQE and generate a failure with -ECANCELED result.

Parameters

- `sqe` – **[in]** The SQE to cancel

Returns
0 if the SQE was flagged for cancellation

Returns
<0 on error

int `rtio_sqe_copy_in_get_handles`(struct *rtio* *r, const struct *rtio_sqe* *sqes, struct *rtio_sqe* **handle, size_t sqe_count)

Copy an array of SQEs into the queue and get resulting handles back.

Copies one or more SQEs into the RTIO context and optionally returns their generated SQE handles. Handles can be used to cancel events via the *rtio_sqe_cancel()* call.

Parameters

- `r` – **[in]** RTIO context
- `sqes` – **[in]** Pointer to an array of SQEs
- `handle` – **[out]** Optional pointer to *rtio_sqe* pointer to store the handle of the first generated SQE. Use NULL to ignore.
- `sqe_count` – **[in]** Count of sqes in array

Return values

- `0` – success
- `-ENOMEM` – not enough room in the queue

static inline int `rtio_sqe_copy_in`(struct *rtio* *r, const struct *rtio_sqe* *sqes, size_t sqe_count)

Copy an array of SQEs into the queue.

Useful if a batch of submissions is stored in ROM or RTIO is used from user mode where a copy must be made.

Partial copying is not done as chained SQEs need to be submitted as a whole set.

Parameters

- `r` – RTIO context
- `sqes` – Pointer to an array of SQEs
- `sqe_count` – Count of sqes in array

Return values

- `0` – success
- `-ENOMEM` – not enough room in the queue

int `rtio_cqe_copy_out`(struct *rtio* *r, struct *rtio_cqe* *cqes, size_t cqe_count, *k_timeout_t* timeout)

Copy an array of CQEs from the queue.

Copies from the RTIO context and its queue completion queue events, waiting for the given time period to gather the number of completions requested.

Parameters

- `r` – RTIO context
- `cqes` – Pointer to an array of SQEs
- `cqe_count` – Count of sqes in array

- **timeout** – Timeout to wait for each completion event. Total wait time is potentially timeout*cqe_count at maximum.

Return values

copy_count – Count of copied CQEs (0 to cqe_count)

int **rtio_submit**(struct *rtio* *r, uint32_t wait_count)

Submit I/O requests to the underlying executor.

Submits the queue of submission queue events to the executor. The executor will do the work of managing tasks representing each submission chain, freeing submission queue events when done, and producing completion queue events as submissions are completed.

Parameters

- **r** – RTIO context
- **wait_count** – Number of submissions to wait for completion of.

Return values

0 – On success

Variables

struct *k_mem_partition* **rtio_partition**

The memory partition associated with all RTIO context information.

struct **rtio_sqe**

#include <rtio.h> A submission queue event.

Public Members

uint8_t **op**

Op code.

uint8_t **prio**

Op priority.

uint16_t **flags**

Op Flags.

uint16_t **iodev_flags**

Op iodev flags.

const struct *rtio_iodev* ***iodev**

Device to operation on.

void ***userdata**

User provided data which is returned upon operation completion.

Could be a pointer or integer.

If unique identification of completions is desired this should be unique as well.

uint32_t **buf_len**
: Length of buffer.

uint8_t ***buf**
: Buffer to use.

uint8_t **tiny_buf_len**
: Length of tiny buffer.

uint8_t **tiny_buf**[7]
: Tiny buffer.

void ***arg0**
: Last argument given to callback.

uint32_t **i2c_config**
: OP_I2C_CONFIGURE.

struct **rtio_cqe**
: *#include <rtio.h>* A completion queue event.

Public Members

int32_t **result**
: Result from operation.

void ***userdata**
: Associated userdata with operation.

uint32_t **flags**
: Flags associated with the operation.

struct **rtio_sqe_pool**
: *#include <rtio.h>*

struct **rtio_cqe_pool**
: *#include <rtio.h>*

struct **rtio**
: *#include <rtio.h>* An RTIO context containing what can be viewed as a pair of queues.

  A queue for submissions (available and in queue to be produced) as well as a queue of completions (available and ready to be consumed).

  The rtio executor along with any objects implementing the *rtio_iodev* interface are the consumers of submissions and producers of completions.

  No work is started until *rtio_submit()* is called.

struct `rtio_iodev_sqe`

#include <*rtio.h*> Compute the mempool block index for a given pointer.

IO device submission queue entry

May be cast safely to and from a *rtio_sqe* as they occupy the same memory provided by the pool

Param r
[in] RTIO context

Param ptr
[in] Memory pointer in the mempool

Return
Index of the mempool block associated with the pointer. Or UINT16_MAX if invalid.

struct `rtio_iodev_api`

#include <*rtio.h*> API that an RTIO IO device should implement.

Public Members

void (*`submit`)(struct *rtio_iodev_sqe* *iodev_sqe)

Submit to the iodev an entry to work on.

This call should be short in duration and most likely either enqueue or kick off an entry with the hardware.

Param iodev_sqe
Submission queue entry

struct `rtio_iodev`

#include <*rtio.h*> An IO device with a function table for submitting requests.

4.33 Zephyr bus (zbus)

The *Zephyr bus - zbus* is a lightweight and flexible software bus enabling a simple way for threads to talk to one another in a many-to-many way.

- *Concepts*
  - *Virtual Distributed Event Dispatcher*
  - *Limitations*
- *Usage*
  - *Publishing to a channel*
  - *Reading from a channel*
  - *Notifying a channel*
  - *Declaring channels and observers*

  - *Iterating over channels and observers*
  - *Advanced channel control*
- *Samples*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

4.33.1 Concepts

Threads can send messages to one or more observers using zbus. It makes the many-to-many communication possible. The bus implements message-passing and publish/subscribe communication paradigms that enable threads to communicate synchronously or asynchronously through shared memory.

The communication through zbus is channel-based. Threads (or callbacks) use channels to exchange messages. Additionally, besides other actions, threads can publish and observe channels. When a thread publishes a message on a channel, the bus will make the message available to all the published channel's observers. Based on the observer's type, it can access the message directly, receive a copy of it, or even receive only a reference of the published channel.

The figure below shows an example of a typical application using zbus in which the application logic (hardware independent) talks to other threads via software bus. Note that the threads are decoupled from each other because they only use zbus channels and do not need to know each other to talk.

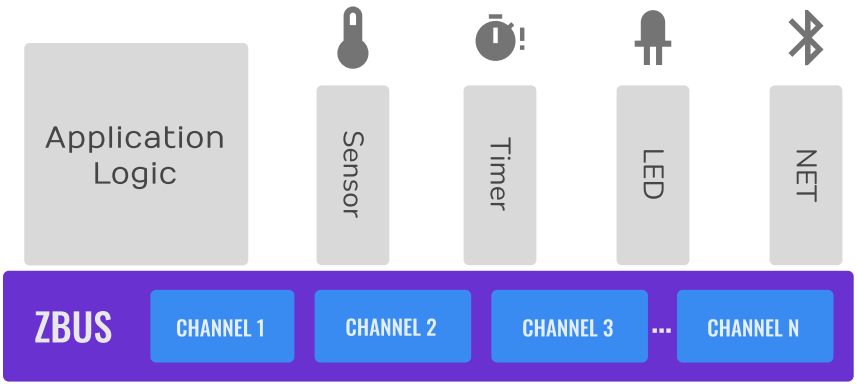


Fig. 25: A typical zbus application architecture.

The bus comprises:

- Set of channels that consists of the control metadata information, and the message itself;
- *Virtual Distributed Event Dispatcher* (VDED), the bus logic responsible for sending notifications/messages to the observers. The VDED logic runs inside the publishing action in the same thread context, giving the bus an idea of a distributed execution. When a thread publishes to a channel, it also propagates the notifications to the observers;
- Threads (subscribers and message subscribers) and callbacks (listeners) publishing, reading, and receiving notifications from the bus.

The bus makes the publish, read, claim, finish, notify, and subscribe actions available over channels. Publishing, reading, claiming, and finishing are available in all RTOS thread contexts, including ISRs. The publish and read operations are simple and fast; the procedure is channel

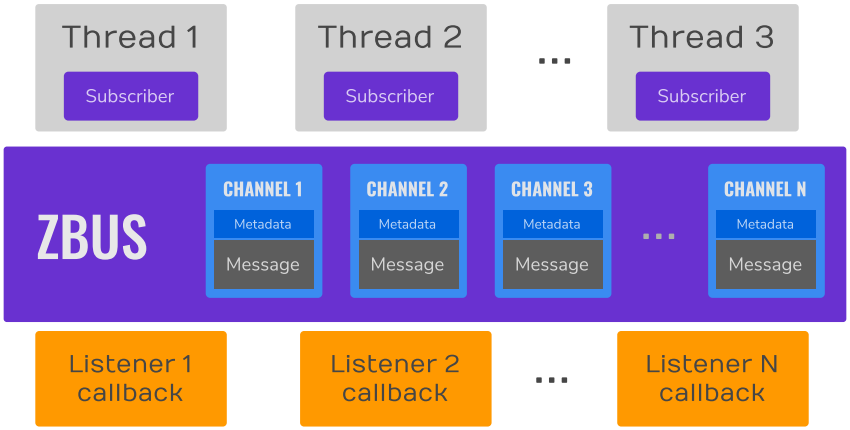


Fig. 26: ZBus anatomy.

locking followed by a memory copy to and from a shared memory region and then a channel unlocking. Another essential aspect of zbus is the observers. There are three types of observers:



Fig. 27: ZBus observers.

- Listeners, a callback that the event dispatcher executes every time an observed channel is published or notified;
- Subscriber, a thread-based observer that relies internally on a message queue where the event dispatcher puts a changed channel's reference every time an observed channel is published or notified. Note this kind of observer does not receive the message itself. It should read the message from the channel after receiving the notification;
- Message subscribers, a thread-based observer that relies internally on a FIFO where the event dispatcher puts a copy of the message every time an observed channel is published or notified.

Channel observation structures define the relationship between a channel and its observers. For every observation, a pair channel/observer. Developers can statically allocate observation using the *ZBUS_CHAN_DEFINE* or *ZBUS_CHAN_ADD_OBS*. There are also runtime observers, enabling developers to create runtime observations. It is possible to disable an observer entirely or observations individually. The event dispatcher will ignore disabled observers and observations.

The above figure illustrates some states, from (a) to (d), for channels from `C1` to `C5`, `Subscriber 1`, and the observations. The last two are in orange to indicate they are dynamically allocated (runtime observation). (a) shows that the observer and all observations are enabled. (b) shows the observer is disabled, so the event dispatcher will ignore it. (c) shows the observer enabled. However, there is one static observervation disabled. The event dispatcher will only stop sending notifications from channel `C3`. In (d), the event dispatcher will stop sending notifications from channels `C3` and `C5` to `Subscriber 1`.

Suppose a usual sensor-based solution is in the figure below for illustration purposes. When triggered, the timer publishes to the `Trigger` channel. As the sensor thread subscribed to the `Trigger` channel, it receives the sensor data. Notice the VDED executes the `Blink` because it also listens to the `Trigger` channel. When the sensor data is ready, the sensor thread publishes it to the `Sensor data` channel. The core thread receives the message as a `Sensor data` channel message subscriber, processes the sensor data, and stores it in an internal sample buffer. It repeats until the sample buffer is full; when it happens, the core thread aggregates the sample buffer information, prepares a package, and publishes that to the `Payload` channel. The Lora thread receives that because it is a `Payload` channel message subscriber and sends the payload to the

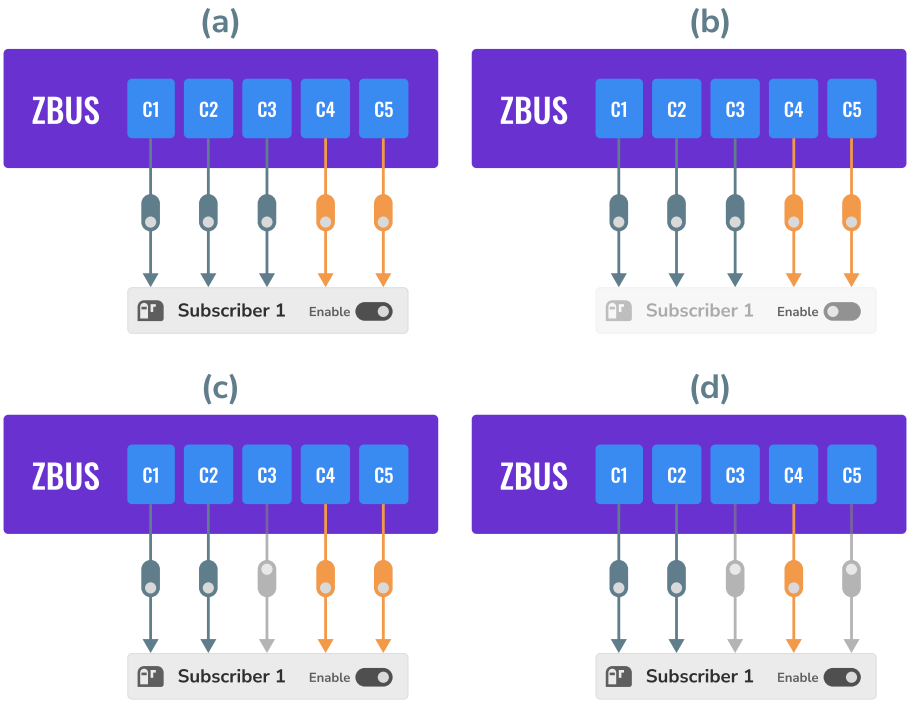


Fig. 28: ZBus observation mask.

cloud. When it completes the transmission, the Lora thread publishes to the `Transmission done` channel. The VDED executes the `Blink` again since it listens to the `Transmission done` channel.

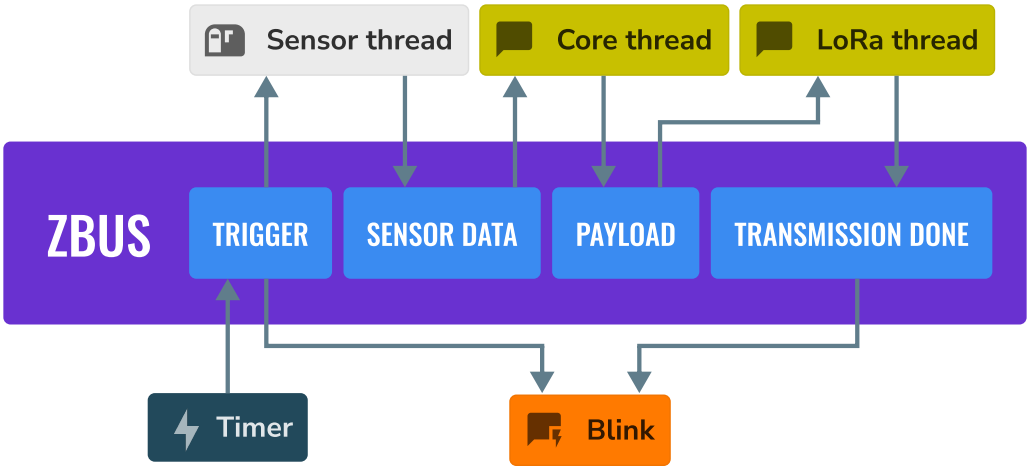


Fig. 29: ZBus sensor-based application.

This way of implementing the solution makes the application more flexible, enabling us to change things independently. For example, we want to change the trigger from a timer to a button press. We can do that, and the change does not affect other parts of the system. Likewise, we would like to change the communication interface from LoRa to Bluetooth; we only need to change the LoRa thread. No other change is required in order to make that work. Thus, the developer would do that for every block of the image. Based on that, there is a sign zbus promotes decoupling in the system architecture.

Another important aspect of using zbus is the reuse of system modules. If a code portion with well-defined behaviors (we call that module) only uses zbus channels and not hardware interfaces, it can easily be reused in other solutions. The new solution must implement the interfaces (set of channels) the module needs to work. That indicates zbus could improve the module reuse.

The last important note is the zbus solution reach. We can count on many ways of using zbus to enable the developer to be as free as possible to create what they need. For example, messages can be dynamic or static allocated; notifications can be synchronous or asynchronous; the developer can control the channel in so many different ways claiming the channel, developers can add their metadata information to a channel by using the user-data field, the discretionary use of a validator enables the systems to be accurate over message format, and so on. Those characteristics increase the solutions that can be done with zbus and make it a good fit as an open-source community tool.

Virtual Distributed Event Dispatcher

The VDED execution always happens in the publisher's context. It can be a thread or an ISR. Be careful with publications inside ISR because the scheduler won't preempt the VDED. Use that wisely. The basic description of the execution is as follows:

- The channel lock is acquired;
- The channel receives the new message via direct copy (by a raw `memcpy()`);
- The event dispatcher logic executes the listeners, sends a copy of the message to the message subscribers, and pushes the channel's reference to the subscribers' notification message queue in the same sequence they appear on the channel observers' list. The listeners can perform non-copy quick access to the constant message reference directly (via the *zbus_chan_const_msg()* function) since the channel is still locked;
- At last, the publishing function unlocks the channel.

To illustrate the VDED execution, consider the example illustrated below. We have four threads in ascending priority `S1`, `MS2`, `MS1`, and `T1` (the highest priority); two listeners, `L1` and `L2`; and channel A. Supposing `L1`, `L2`, `MS1`, `MS2`, and `S1` observer channel A.

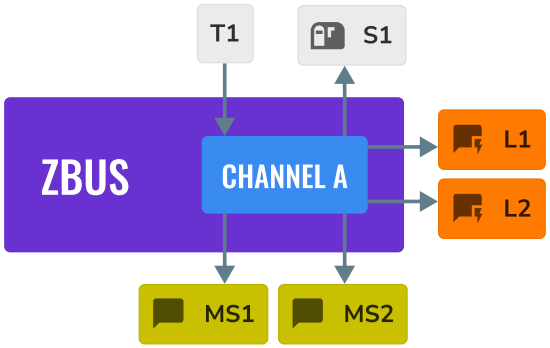


Fig. 30: ZBus VDED execution example scenario.

The following code implements channel A. Note the `struct a_msg` is illustrative only.

```
ZBUS_CHAN_DEFINE(a_chan,                        /* Name */
         struct a_msg,                          /* Message type */

         NULL,                                  /* Validator */
         NULL,                                  /* User Data */
         ZBUS_OBSERVERS(L1, L2, MS1, MS2, S1), /* observers */
         ZBUS_MSG_INIT(0)                       /* Initial value {0} */
);
```

In the figure below, the letters indicate some action related to the VDED execution. The X-axis represents the time, and the Y-axis represents the priority of threads. Channel A's message, represented by a voice balloon, is only one memory portion (shared memory). It appears several times only as an illustration of the message at that point in time.

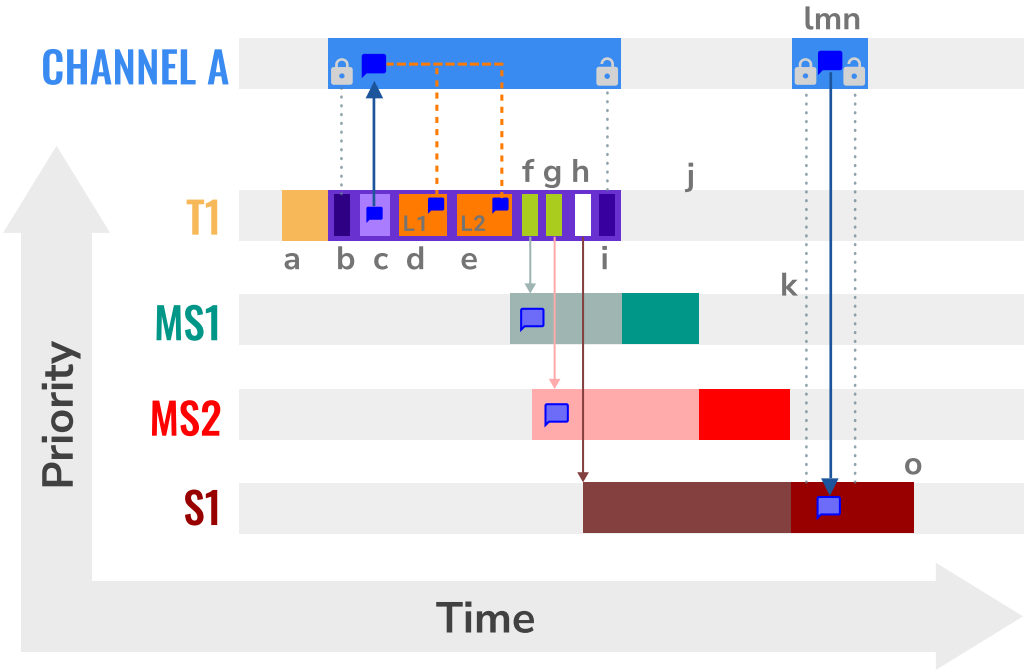


Fig. 31: ZBus VDED execution detail for priority T1 > MS1 > MS2 > S1.

The figure above illustrates the actions performed during the VDED execution when T1 publishes to channel A. Thus, the table below describes the activities (represented by a letter) of the VDED execution. The scenario considers the following priorities: T1 > MS1 > MS2 > S1. T1 has the highest priority.

Table 65: VDED execution steps in detail for priority T1 > MS1 > MS2 > S1.

Actions	Description
a	T1 starts and, at some point, publishes to channel A.
b	The publishing (VDED) process starts. The VDED locks the channel A.
c	The VDED copies the T1 message to the channel A message.
d, e	The VDED executes L1 and L2 in the respective sequence. Inside the listeners, usually, there is a call to the *zbus_chan_const_msg()* function, which provides a direct constant reference to channel A's message. It is quick, and no copy is needed here.
f, g	The VDED copies the message and sends that to MS1 and MS2 sequentially. Notice the threads get ready to execute right after receiving the notification. However, they go to a pending state because they have less priority than T1.
h	The VDED pushes the notification message to the queue of S1. Notice the thread gets ready to execute right after receiving the notification. However, it goes to a pending state because it cannot access the channel since it is still locked.
i	VDED finishes the publishing by unlocking channel A. The MS1 leaves the pending state and starts executing.
j	MS1 finishes execution. The MS2 leaves the pending state and starts executing.
k	MS2 finishes execution. The S1 leaves the pending state and starts executing.
l, m, n	The S1 leaves the pending state since channel A is not locked. It gets in the CPU again and starts executing. As it did receive a notification from channel A, it performed a channel read (as simple as lock, memory copy, unlock), continues its execution and goes out of the CPU.
o	S1 finishes its workload.

The figure below illustrates the actions performed during the VDED execution when T1 publishes

to channel A. The scenario considers the following priorities: T1 < MS1 < MS2 < S1.

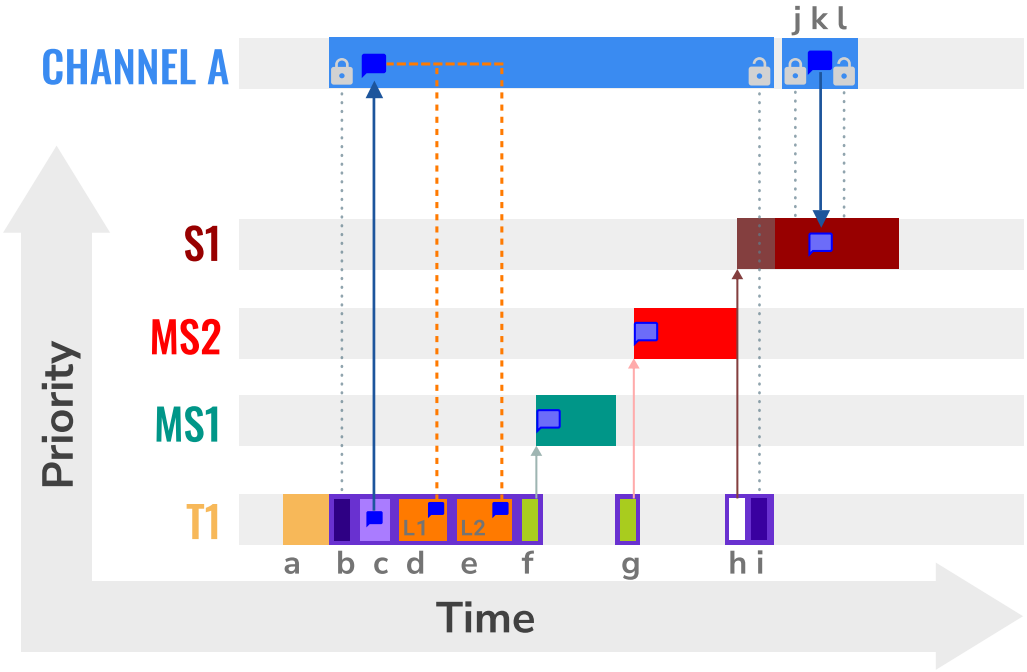


Fig. 32: ZBus VDED execution detail for priority T1 < MS1 < MS2 < S1.

Thus, the table below describes the activities (represented by a letter) of the VDED execution.

Table 66: VDED execution steps in detail for priority T1 < MS1 < MS2 < S1.

Actions	Description
a	T1 starts and, at some point, publishes to channel A.
b	The publishing (VDED) process starts. The VDED locks the channel A.
c	The VDED copies the T1 message to the channel A message.
d, e	The VDED executes L1 and L2 in the respective sequence. Inside the listeners, usually, there is a call to the `zbus_chan_const_msg()` function, which provides a direct constant reference to channel A's message. It is quick, and no copy is needed here.
f	The VDED copies the message and sends that to MS1. MS1 preempts T1 and starts working. After that, the T1 regain MCU.
g	The VDED copies the message and sends that to MS2. MS2 preempts T1 and starts working. After that, the T1 regain MCU.
h	The VDED pushes the notification message to the queue of S1.
i	VDED finishes the publishing by unlocking channel A.
j, k, l	The S1 leaves the pending state since channel A is not locked. It gets in the CPU again and starts executing. As it did receive a notification from channel A, it performs a channel read (as simple as lock, memory copy, unlock), continues its execution, and goes out the CPU.

HLP priority boost ZBus implements the Highest Locker Protocol that relies on the observers' thread priority to determine a temporary publisher priority. The protocol considers the channel's Highest Observer Priority (HOP); even if the observer is not waiting for a message on the channel, it is considered in the calculation. The VDED will elevate the publisher's priority based on the HOP to ensure small latency and as few preemptions as possible.

Note

The priority boost is enabled by default. To deactivate it, you must set the CONFIG_ZBUS_PRIORITY_BOOST configuration.

Warning

ZBus priority boost does not consider runtime observers on the HOP calculations.

The figure below illustrates the actions performed during the VDED execution when T1 publishes to channel A. The scenario considers the priority boost feature and the following priorities: T1 < MS1 < MS2 < S1.

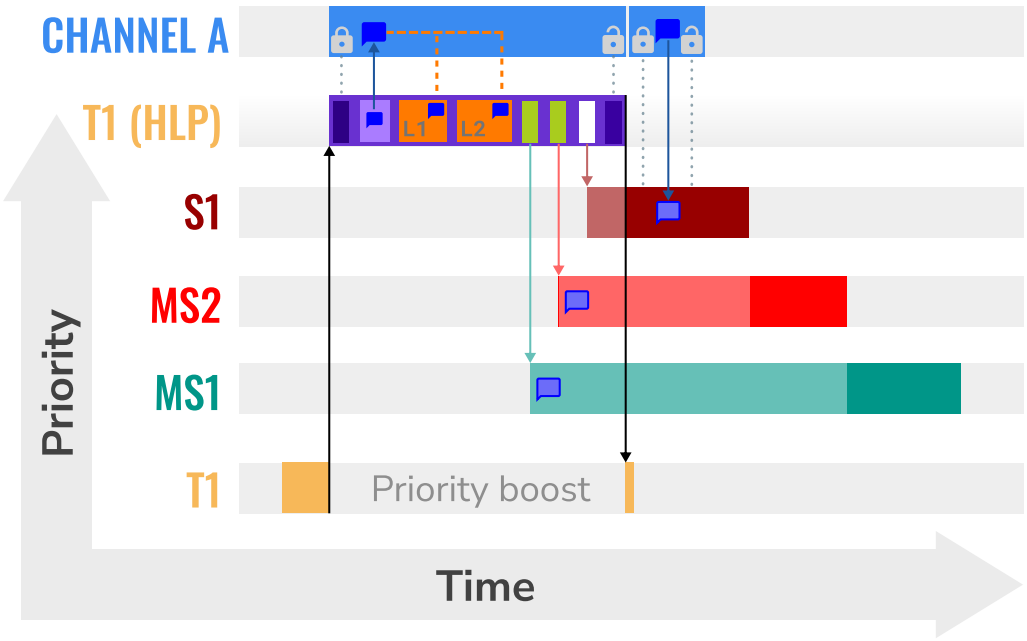


Fig. 33: ZBus VDED execution detail with priority boost enabled and for priority T1 < MS1 < MS2 < S1.

To properly use the priority boost, attaching the observer to a thread is necessary. When the subscriber is attached to a thread, it assumes its priority, and the priority boost algorithm will consider the observer's priority. The following code illustrates the thread-attaching function.

```
ZBUS_SUBSCRIBER_DEFINE(s1, 4);
void s1_thread(void *ptr1, void *ptr2, void *ptr3)
{
        ARG_UNUSED(ptr1);
        ARG_UNUSED(ptr2);
        ARG_UNUSED(ptr3);

        const struct zbus_channel *chan;

        zbus_obs_attach_to_thread(&s1);

        while (1) {
                zbus_sub_wait(&s1, &chan, K_FOREVER);
```

(continues on next page)

(continued from previous page)

```
                /* Subscriber implementation */

        }
}
K_THREAD_DEFINE(s1_id, CONFIG_MAIN_STACK_SIZE, s1_thread, NULL, NULL, NULL, 2, 0, 0);
```

On the above code, the *zbus_obs_attach_to_thread()* will set the s1 observer with priority two as the thread has that priority. It is possible to reverse that by detaching the observer using the *zbus_obs_detach_from_thread()*. Only enabled observers and observations will be considered on the channel HOP calculation. Masking a specific observation of a channel will affect the channel HOP.

In summary, the benefits of the feature are:

- The HLP is more effective for zbus than the mutexes priority inheritance;
- No bounded priority inversion will happen among the publisher and the observers;
- No other threads (that are not involved in the communication) with priority between T1 and S1 can preempt T1, avoiding unbounded priority inversion;
- Message subscribers will wait for the VDED to finish the message delivery process. So the VDED execution will be faster and more consistent;
- The HLP priority is dynamic and can change in execution;
- ZBus operations can be used inside ISRs;
- The priority boosting feature can be turned off, and plain semaphores can be used as the channel lock mechanism;
- The Highest Locker Protocol's major disadvantage, the Inheritance-related Priority Inversion, is acceptable in the zbus scenario since it will ensure a small bus latency.

Limitations

Based on the fact that developers can use zbus to solve many different problems, some challenges arise. ZBus will not solve every problem, so it is necessary to analyze the situation to be sure zbus is applicable. For instance, based on the zbus benchmark, it would not be well suited to a high-speed stream of bytes between threads. The *Pipe* kernel object solves this kind of need.

Delivery guarantees ZBus always delivers the messages to the listeners and message subscribers. However, there are no message delivery guarantees for subscribers because zbus only sends the notification, but the message reading depends on the subscriber's implementation. It is possible to increase the delivery rate by following design tips:

- Keep the listeners quick-as-possible (deal with them as ISRs). If some processing is needed, consider submitting a work item to a work-queue;
- Try to give producers a high priority to avoid losses;
- Leave spare CPU for observers to consume data produced;
- Consider using message queues or pipes for intensive byte transfers.

Warning

ZBus uses include/zephyr/net/buf.h (network buffers) to exchange data with message subscribers. Thus, choose carefully the configurations CONFIG_ZBUS_MSG_SUBSCRIBER_NET_BUF_POOL_SIZE and CONFIG_HEAP_MEM_POOL_SIZE. They

are crucial to a proper VDED execution (delivery guarantee) considering message subscribers. If you want to keep an isolated pool for a specific set of channels, you can use CONFIG_ZBUS_MSG_SUBSCRIBER_NET_BUF_POOL_ISOLATION with a dedicated pool. Look at the zbus-msg-subscriber to see the isolation in action.

Warning

Subscribers will receive only the reference of the changing channel. A data loss may be perceived if the channel is published twice before the subscriber reads it. The second publication overwrites the value from the first. Thus, the subscriber will receive two notifications, but only the last data is there.

Message delivery sequence The message delivery will follow the precedence:

1. Observers defined in a channel using the *ZBUS_CHAN_DEFINE* (following the definition sequence);
2. Observers defined using the *ZBUS_CHAN_ADD_OBS* based on the sequence priority (parameter of the macro);
3. The latest is the runtime observers in the addition sequence using the *zbus_chan_add_obs()*.

Note

The VDED will ignore all disabled observers or observations.

4.33.2 Usage

ZBus operation depends on channels and observers. Therefore, it is necessary to determine its message and observers list during the channel definition. A message is a regular C struct; the observer can be a subscriber (asynchronous), a message subscriber (asynchronous), or a listener (synchronous).

The following code defines and initializes a regular channel and its dependencies. This channel exchanges accelerometer data, for example.

```
struct acc_msg {
        int x;
        int y;
        int z;
};

ZBUS_CHAN_DEFINE(acc_chan,                              /* Name */
         struct acc_msg,                                /* Message type */

         NULL,                                          /* Validator */
         NULL,                                          /* User Data */
         ZBUS_OBSERVERS(my_listener, my_subscriber,
                        my_msg_subscriber),             /* observers */
         ZBUS_MSG_INIT(.x = 0, .y = 0, .z = 0)          /* Initial value */
);

void listener_callback_example(const struct zbus_channel *chan)
{
```

(continues on next page)

(continued from previous page)

```
        const struct acc_msg *acc;
        if (&acc_chan == chan) {
                acc = zbus_chan_const_msg(chan); // Direct message access
                LOG_DBG("From listener -> Acc x=%d, y=%d, z=%d", acc->x, acc->y, acc->z);
        }
}

ZBUS_LISTENER_DEFINE(my_listener, listener_callback_example);

ZBUS_LISTENER_DEFINE(my_listener2, listener_callback_example);

ZBUS_CHAN_ADD_OBS(acc_chan, my_listener2, 3);

ZBUS_SUBSCRIBER_DEFINE(my_subscriber, 4);
void subscriber_task(void)
{
        const struct zbus_channel *chan;

        while (!zbus_sub_wait(&my_subscriber, &chan, K_FOREVER)) {
                struct acc_msg acc = {0};

                if (&acc_chan == chan) {
                        // Indirect message access
                        zbus_chan_read(&acc_chan, &acc, K_NO_WAIT);
                        LOG_DBG("From subscriber -> Acc x=%d, y=%d, z=%d", acc.x, acc.y,␣
↪acc.z);
                }
        }
}
K_THREAD_DEFINE(subscriber_task_id, 512, subscriber_task, NULL, NULL, NULL, 3, 0, 0);

ZBUS_MSG_SUBSCRIBER_DEFINE(my_msg_subscriber);
static void msg_subscriber_task(void *ptr1, void *ptr2, void *ptr3)
{
        ARG_UNUSED(ptr1);
        ARG_UNUSED(ptr2);
        ARG_UNUSED(ptr3);
        const struct zbus_channel *chan;

        struct acc_msg acc = {0};

        while (!zbus_sub_wait_msg(&my_msg_subscriber, &chan, &acc, K_FOREVER)) {
                if (&acc_chan == chan) {
                        LOG_INF("From msg subscriber -> Acc x=%d, y=%d, z=%d", acc.x, acc.y,
↪ acc.z);
                }
        }
}
K_THREAD_DEFINE(msg_subscriber_task_id, 1024, msg_subscriber_task, NULL, NULL, NULL, 3, 0,␣
↪0);
```

It is possible to add static observers to a channel using the *ZBUS_CHAN_ADD_OBS*. We call that a post-definition static observer. The command enables us to indicate an initialization priority that affects the observers' initialization order. The sequence priority param only affects the post-definition static observers. There is no possibility to overwrite the message delivery sequence of the static observers.

Note

It is unnecessary to claim/lock a channel before accessing the message inside the listener since

the event dispatcher calls listeners with the notifying channel already locked. Subscribers, however, must claim/lock that or use regular read operations to access the message after being notified.

Channels can have a *validator function* that enables a channel to accept only valid messages. Publish attempts invalidated by hard channels will return immediately with an error code. This allows original creators of a channel to exert some authority over other developers/publishers who may want to piggy-back on their channels. The following code defines and initializes a *hard channel* and its dependencies. Only valid messages can be published to a *hard channel*. It is possible because a *validator function* was passed to the channel's definition. In this example, only messages with `move` equal to 0, -1, and 1 are valid. Publish function will discard all other values to `move`.

```
struct control_msg {
        int move;
};

bool control_validator(const void* msg, size_t msg_size) {
        const struct control_msg* cm = msg;
        bool is_valid = (cm->move == -1) || (cm->move == 0) || (cm->move == 1);
        return is_valid;
}

static int message_count = 0;

ZBUS_CHAN_DEFINE(control_chan,    /* Name */
         struct control_msg,      /* Message type */

         control_validator,       /* Validator */
         &message_count,          /* User data */
         ZBUS_OBSERVERS_EMPTY,    /* observers */
         ZBUS_MSG_INIT(.move = 0) /* Initial value */
);
```

The following sections describe in detail how to use zbus features.

Publishing to a channel

Messages are published to a channel in zbus by calling *zbus_chan_pub()*. For example, the following code builds on the examples above and publishes to channel `acc_chan`. The code is trying to publish the message `acc1` to channel `acc_chan`, and it will wait up to one second for the message to be published. Otherwise, the operation fails. As can be inferred from the code sample, it's OK to use stack allocated messages since VDED copies the data internally.

```
struct acc_msg acc1 = {.x = 1, .y = 1, .z = 1};
zbus_chan_pub(&acc_chan, &acc1, K_SECONDS(1));
```

Warning

Only use this function inside an ISR with a *K_NO_WAIT* timeout.

Reading from a channel

Messages are read from a channel in zbus by calling *zbus_chan_read()*. So, for example, the following code tries to read the channel `acc_chan`, which will wait up to 500 milliseconds to read the message. Otherwise, the operation fails.

```
struct acc_msg acc = {0};
zbus_chan_read(&acc_chan, &acc, K_MSEC(500));
```

Warning

Only use this function inside an ISR with a *K_NO_WAIT* timeout.

Warning

Choose the timeout of *zbus_chan_read()* after receiving a notification from *zbus_sub_wait()* carefully because the channel will always be unavailable during the VDED execution. Using K_NO_WAIT for reading is highly likely to return a timeout error if there are more than one subscriber. For example, consider the VDED illustration again and notice how S1 read attempts would definitely fail with K_NO_WAIT. For more details, check the *Virtual Distributed Event Dispatcher* section.

Notifying a channel

It is possible to force zbus to notify a channel's observers by calling *zbus_chan_notify()*. For example, the following code builds on the examples above and forces a notification for the channel acc_chan. Note this can send events with no message, which does not require any data exchange. See the code example under *Claim and finish a channel* where this may become useful.

```
zbus_chan_notify(&acc_chan, K_NO_WAIT);
```

Warning

Only use this function inside an ISR with a *K_NO_WAIT* timeout.

Declaring channels and observers

For accessing channels or observers from files other than its defining files, it is necessary to declare them by calling *ZBUS_CHAN_DECLARE* and *ZBUS_OBS_DECLARE*. In other words, zbus channel definitions and declarations with the same channel names in different files would point to the same (global) channel. Thus, developers should be careful about existing channels, and naming new channels or linking will fail. It is possible to declare more than one channel or observer on the same call. The following code builds on the examples above and displays the defined channels and observers.

```
ZBUS_OBS_DECLARE(my_listener, my_subscriber);
ZBUS_CHAN_DECLARE(acc_chan, version_chan);
```

Iterating over channels and observers

ZBus subsystem also implements *Iterable Sections* for channels and observers, for which there are supporting APIs like *zbus_iterate_over_channels()*, *zbus_iterate_over_channels_with_user_data()*, *zbus_iterate_over_observers()* and *zbus_iterate_over_observers_with_user_data()*. This feature enables developers to call a procedure over all declared channels, where the procedure parameter is a *zbus_channel*.

The execution sequence is in the alphabetical name order of the channels (see *Iterable Sections* documentation for details). ZBus also implements this feature for *zbus_observer*.

```
static bool print_channel_data_iterator(const struct zbus_channel *chan, void *user_data)
{
	int *count = user_data;

	LOG_INF("%d - Channel %s:", *count, zbus_chan_name(chan));
	LOG_INF("	Message size: %d", zbus_chan_msg_size(chan));
	LOG_INF("	Observers:");

	++(*count);

	struct zbus_channel_observation *observation;

	for (int16_t i = *chan->observers_start_idx, limit = *chan->observers_end_idx; i <
↪limit;
		++i) {
		STRUCT_SECTION_GET(zbus_channel_observation, i, &observation);

		LOG_INF("	- %s", observation->obs->name);
	}

	struct zbus_observer_node *obs_nd, *tmp;

	SYS_SLIST_FOR_EACH_CONTAINER_SAFE(chan->observers, obs_nd, tmp, node) {
		LOG_INF("	- %s", obs_nd->obs->name);
	}

	return true;
}

static bool print_observer_data_iterator(const struct zbus_observer *obs, void *user_data)
{
	int *count = user_data;

	LOG_INF("%d - %s %s", *count, obs->queue ? "Subscriber" : "Listener", zbus_obs_
↪name(obs));

	++(*count);

	return true;
}

int main(void)
{
	int count = 0;

	LOG_INF("Channel list:");

	zbus_iterate_over_channels_with_user_data(print_channel_data_iterator, &count);

	count = 0;

	LOG_INF("Observers list:");

	zbus_iterate_over_observers_with_user_data(print_observer_data_iterator, &count);

	return 0;
}
```

The code will log the following output:

```
D: Channel list:
D: 0 - Channel acc_chan:
D:       Message size: 12
D:       Observers:
D:       - my_listener
D:       - my_subscriber
D: 1 - Channel version_chan:
D:       Message size: 4
D:       Observers:
D: Observers list:
D: 0 - Listener my_listener
D: 1 - Subscriber my_subscriber
```

Advanced channel control

ZBus was designed to be as flexible and extensible as possible. Thus, there are some features designed to provide some control and extensibility to the bus.

Listeners message access For performance purposes, listeners can access the receiving channel message directly since they already have the channel locked for it. To access the channel's message, the listener should use the *zbus_chan_const_msg()* because the channel passed as an argument to the listener function is a constant pointer to the channel. The const pointer return type tells developers not to modify the message.

```
void listener_callback_example(const struct zbus_channel *chan)
{
        const struct acc_msg *acc;
        if (&acc_chan == chan) {
                acc = zbus_chan_const_msg(chan); // Use this
                // instead of zbus_chan_read(chan, &acc, K_MSEC(200))
                // or zbus_chan_msg(chan)

                LOG_DBG("From listener -> Acc x=%d, y=%d, z=%d", acc->x, acc->y, acc->z);
        }
}
```

User Data It is possible to pass custom data into the channel's `user_data` for various purposes, such as writing channel metadata. That can be achieved by passing a pointer to the channel definition macro's `user_data` field, which will then be accessible by others. Note that `user_data` is individual for each channel. Also, note that `user_data` access is not thread-safe. For thread-safe access to `user_data`, see the next section.

Claim and finish a channel To take more control over channels, two functions were added *zbus_chan_claim()* and *zbus_chan_finish()*. With these functions, it is possible to access the channel's metadata safely. When a channel is claimed, no actions are available to that channel. After finishing the channel, all the actions are available again.

Warning

Never change the fields of the channel struct directly. It may cause zbus behavior inconsistencies and scheduling issues.

Warning

Only use this function inside an ISR with a *K_NO_WAIT* timeout.

The following code builds on the examples above and claims the acc_chan to set the user_data to the channel. Suppose we would like to count how many times the channels exchange messages. We defined the user_data to have the 32 bits integer. This code could be added to the listener code described above.

```
if (!zbus_chan_claim(&acc_chan, K_MSEC(200))) {
        int *message_counting = (int *) zbus_chan_user_data(&acc_chan);
        *message_counting += 1;
        zbus_chan_finish(&acc_chan);
}
```

The following code has the exact behavior of the code in *Publishing to a channel*.

```
if (!zbus_chan_claim(&acc_chan, K_MSEC(200))) {
        struct acc_msg *acc1 = (struct acc_msg *) zbus_chan_msg(&acc_chan);
        acc1.x = 1;
        acc1.y = 1;
        acc1.z = 1;
        zbus_chan_finish(&acc_chan);
        zbus_chan_notify(&acc_chan, K_SECONDS(1));
}
```

The following code has the exact behavior of the code in *Reading from a channel*.

```
if (!zbus_chan_claim(&acc_chan, K_MSEC(200))) {
        const struct acc_msg *acc1 = (const struct acc_msg *) zbus_chan_const_msg(&acc_
↪chan);
        // access the acc_msg fields directly.
        zbus_chan_finish(&acc_chan);
}
```

Runtime observer registration It is possible to add observers to channels in runtime. This feature uses the heap to allocate the nodes dynamically. The heap size limits the number of dynamic observers zbus can create. Therefore, set the CONFIG_ZBUS_RUNTIME_OBSERVERS to enable the feature. It is possible to adjust the heap size by changing the configuration CONFIG_HEAP_MEM_POOL_SIZE. The following example illustrates the runtime registration usage.

```
ZBUS_LISTENER_DEFINE(my_listener, callback);
// ...
void thread_entry(void) {
        // ...
        /* Adding the observer to channel chan1 */
        zbus_chan_add_obs(&chan1, &my_listener, K_NO_WAIT);
        /* Removing the observer from channel chan1 */
        zbus_chan_rm_obs(&chan1, &my_listener, K_NO_WAIT);
```

4.33.3 Samples

For a complete overview of zbus usage, take a look at the samples. There are the following samples available:

- zbus-hello-world illustrates the code used above in action;

- zbus-work-queue shows how to define and use different kinds of observers. Note there is an example of using a work queue instead of executing the listener as an execution option;
- zbus-msg-subscriber illustrates how to use message subscribers;
- zbus-dyn-channel demonstrates how to use dynamically allocated exchanging data in zbus;
- zbus-uart-bridge shows an example of sending the operation of the channel to a host via serial;
- zbus-remote-mock illustrates how to implement an external mock (on the host) to send and receive messages to and from the bus;
- zbus-priority-boost illustrates zbus priority boost feature with a priority inversion scenario;
- zbus-runtime-obs-registration illustrates a way of using the runtime observer registration feature;
- zbus-confirmed-channel implements a way of implement confirmed channel only with subscribers;
- zbus-benchmark implements a benchmark with different combinations of inputs.

4.33.4 Suggested Uses

Use zbus to transfer data (messages) between threads in one-to-one, one-to-many, and many-to-many synchronously or asynchronously. Choosing the proper observer type is crucial. Use subscribers for scenarios that can tolerate message losses and duplications; when they cannot, use message subscribers (if you need a thread) or listeners (if you need to be lean and fast). In addition to the listener, another asynchronous message processing mechanism (like *message queues*) may be necessary to retain the pending message until it gets processed.

Note

ZBus can be used to transfer streams from the producer to the consumer. However, this can increase zbus' communication latency. So maybe consider a Pipe a good alternative for this communication topology.

4.33.5 Configuration Options

For enabling zbus, it is necessary to enable the `CONFIG_ZBUS` option.

Related configuration options:

- `CONFIG_ZBUS_PRIORITY_BOOST` zbus Highest Locker Protocol implementation;
- `CONFIG_ZBUS_CHANNELS_SYS_INIT_PRIORITY` determine the `SYS_INIT` priority used by zbus to organize the channels observations by channel;
- `CONFIG_ZBUS_CHANNEL_NAME` enables the name of channels to be available inside the channels metadata. The log uses this information to show the channels' names;
- `CONFIG_ZBUS_OBSERVER_NAME` enables the name of observers to be available inside the channels metadata;
- `CONFIG_ZBUS_MSG_SUBSCRIBER` enables the message subscriber observer type;
- `CONFIG_ZBUS_MSG_SUBSCRIBER_BUF_ALLOC_DYNAMIC` uses the heap to allocate message buffers;
- `CONFIG_ZBUS_MSG_SUBSCRIBER_BUF_ALLOC_STATIC` uses the stack to allocate message buffers;

- CONFIG_ZBUS_MSG_SUBSCRIBER_NET_BUF_POOL_SIZE the available number of message buffers to be used simultaneously;
- CONFIG_ZBUS_MSG_SUBSCRIBER_NET_BUF_POOL_ISOLATION enables the developer to isolate a pool for the message subscriber for a set of channels;
- CONFIG_ZBUS_MSG_SUBSCRIBER_NET_BUF_STATIC_DATA_SIZE the biggest message of zbus channels to be transported into a message buffer;
- CONFIG_ZBUS_RUNTIME_OBSERVERS enables the runtime observer registration.

4.33.6 API Reference

Related code samples

Benchmarking
Measure the time for sending 256KB from a producer to N consumers.

Confirmed channel
Use confirmed zbus channels to ensure all subscribers consume a message.

Dynamic channel
Use zbus channels with dynamically allocated messages.

Message subscriber
Use zbus message subscribers to listen to messages published to channels.

Remote mock sample
Publish to a zbus instance using UART as a bridge.

Runtime observer registration
Use zbus' runtime observer registration to filter data generated by a producer.

UART bridge
Redirect channel events to the host over UART.

Work queue
Use a work queue to process zbus messages in various ways.

zbus Hello World
Make three threads talk to each other using zbus.

zbus Priority Boost
Illustrates zbus priority boost feature with a priority inversion scenario.

group zbus_apis

Zbus API.

Defines

ZBUS_CHAN_ADD_OBS_WITH_MASK(_chan, _obs, _masked, _prio)

Add a static channel observervation.

This macro initializes a channel observation by receiving the channel and the observer.

Parameters

- **_chan** – Channel instance.
- **_obs** – Observer instance.
- **_masked** – Observation state.

- **_prio** – Observer notification sequence priority.

ZBUS_CHAN_ADD_OBS(_chan, _obs, _prio)

Add a static channel observervation.

This macro initializes a channel observation by receiving the channel and the observer.

Parameters

- **_chan** – Channel instance.
- **_obs** – Observer instance.
- **_prio** – Observer notification sequence priority.

ZBUS_OBS_DECLARE(...)

This macro list the observers to be used in a file.

Internally, it declares the observers with the extern statement. Note it is only necessary when the observers are declared outside the file.

ZBUS_CHAN_DECLARE(...)

This macro list the channels to be used in a file.

Internally, it declares the channels with the extern statement. Note it is only necessary when the channels are declared outside the file.

ZBUS_OBSERVERS_EMPTY

This macro indicates the channel has no observers.

ZBUS_OBSERVERS(...)

This macro indicates the channel has listed observers.

Note the sequence of observer notification will follow the same as listed.

ZBUS_CHAN_DEFINE(_name, _type, _validator, _user_data, _observers, _init_val)

Zbus channel definition.

This macro defines a channel.

See also

struct *zbus_channel*

Parameters

- **_name** – The channel's name.
- **_type** – The Message type. It must be a struct or union.
- **_validator** – The validator function.
- **_user_data** – A pointer to the user data.
- **_observers** – The observers list. The sequence indicates the priority of the observer. The first the highest priority.
- **_init_val** – The message initialization.

ZBUS_MSG_INIT(_val, ...)

Initialize a message.

This macro initializes a message by passing the values to initialize the message struct or union.

Parameters

- **_val** – **[in]** Variadic with the initial values. ZBUS_INIT(0) means {0}, as ZBUS_INIT(.a=10, .b=30) means {.a=10, .b=30}.

ZBUS_SUBSCRIBER_DEFINE_WITH_ENABLE(_name, _queue_size, _enable)

Define and initialize a subscriber.

This macro defines an observer of subscriber type. It defines a message queue where the subscriber will receive the notification asynchronously, and initialize the struct *zbus_observer* defining the subscriber.

Parameters

- **_name** – **[in]** The subscriber's name.
- **_queue_size** – **[in]** The notification queue's size.
- **_enable** – **[in]** The subscriber initial enable state.

ZBUS_SUBSCRIBER_DEFINE(_name, _queue_size)

Define and initialize a subscriber.

This macro defines an observer of subscriber type. It defines a message queue where the subscriber will receive the notification asynchronously, and initialize the struct *zbus_observer* defining the subscriber. The subscribers are defined in the enabled state with this macro.

Parameters

- **_name** – **[in]** The subscriber's name.
- **_queue_size** – **[in]** The notification queue's size.

ZBUS_LISTENER_DEFINE_WITH_ENABLE(_name, _cb, _enable)

Define and initialize a listener.

This macro defines an observer of listener type. This macro establishes the callback where the listener will be notified synchronously, and initialize the struct *zbus_observer* defining the listener.

Parameters

- **_name** – **[in]** The listener's name.
- **_cb** – **[in]** The callback function.
- **_enable** – **[in]** The listener initial enable state.

ZBUS_LISTENER_DEFINE(_name, _cb)

Define and initialize a listener.

This macro defines an observer of listener type. This macro establishes the callback where the listener will be notified synchronously and initialize the struct *zbus_observer* defining the listener. The listeners are defined in the enabled state with this macro.

Parameters

- **_name** – **[in]** The listener's name.
- **_cb** – **[in]** The callback function.

ZBUS_MSG_SUBSCRIBER_DEFINE_WITH_ENABLE(_name, _enable)

Define and initialize a message subscriber.

This macro defines an observer of *ZBUS_OBSERVER_SUBSCRIBER_TYPE* type. It defines a FIFO where the subscriber will receive the message asynchronously and initialize the *zbus_observer* defining the subscriber.

Parameters

- **_name** – **[in]** The subscriber's name.
- **_enable** – **[in]** The subscriber's initial state.

ZBUS_MSG_SUBSCRIBER_DEFINE(_name)

Define and initialize an enabled message subscriber.

This macro defines an observer of message subscriber type. It defines a FIFO where the subscriber will receive the message asynchronously and initialize the *zbus_observer* defining the subscriber. The message subscribers are defined in the enabled state with this macro.

Parameters

- **_name** – **[in]** The subscriber's name.

Enums

enum **zbus_observer_type**

Type used to represent an observer type.

A observer can be a listener or a subscriber.

Values:

enumerator **ZBUS_OBSERVER_LISTENER_TYPE**

enumerator **ZBUS_OBSERVER_SUBSCRIBER_TYPE**

enumerator **ZBUS_OBSERVER_MSG_SUBSCRIBER_TYPE**

Functions

int **zbus_chan_pub**(const struct *zbus_channel* *chan, const void *msg, *k_timeout_t* timeout)

Publish to a channel.

This routine publishes a message to a channel.

Parameters

- **chan** – The channel's reference.
- **msg** – Reference to the message where the publish function copies the channel's message data from.
- **timeout** – Waiting period to publish the channel, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- **0** – Channel published.
- **-ENOMSG** – The message is invalid based on the validator function or some of the observers could not receive the notification.
- **-EBUSY** – The channel is busy.
- **-EAGAIN** – Waiting period timed out.

- `-EFAULT` – A parameter is incorrect, the notification could not be sent to one or more observer, or the function context is invalid (inside an ISR). The function only returns this value when the `CONFIG_ZBUS_ASSERT_MOCK` is enabled.

int **zbus_chan_read**(const struct *zbus_channel* *chan, void *msg, *k_timeout_t* timeout)

Read a channel.

This routine reads a message from a channel.

Parameters

- `chan` – **[in]** The channel's reference.
- `msg` – **[out]** Reference to the message where the read function copies the channel's message data to.
- `timeout` – **[in]** Waiting period to read the channel, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- `0` – Channel read.
- `-EBUSY` – The channel is busy.
- `-EAGAIN` – Waiting period timed out.
- `-EFAULT` – A parameter is incorrect, or the function context is invalid (inside an ISR). The function only returns this value when the `CONFIG_ZBUS_ASSERT_MOCK` is enabled.

int **zbus_chan_claim**(const struct *zbus_channel* *chan, *k_timeout_t* timeout)

Claim a channel.

This routine claims a channel. During the claiming period the channel is blocked for publishing, reading, notifying or claiming again. Finishing is the only available action.

Warning

After calling this routine, the channel cannot be used by other thread until the zbus_chan_finish routine is performed.

Warning

This routine should only be called once before a zbus_chan_finish.

Parameters

- `chan` – **[in]** The channel's reference.
- `timeout` – **[in]** Waiting period to claim the channel, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- `0` – Channel claimed.
- `-EBUSY` – The channel is busy.
- `-EAGAIN` – Waiting period timed out.
- `-EFAULT` – A parameter is incorrect, or the function context is invalid (inside an ISR). The function only returns this value when the `CONFIG_ZBUS_ASSERT_MOCK` is enabled.

int **zbus_chan_finish**(const struct *zbus_channel* *chan)

Finish a channel claim.

This routine finishes a channel claim. After calling this routine with success, the channel will be able to be used by other thread.

> **Warning**
>
> This routine must only be used after a zbus_chan_claim.

Parameters

- **chan** – The channel's reference.

Return values

- **0** – Channel finished.
- **-EFAULT** – A parameter is incorrect, or the function context is invalid (inside an ISR). The function only returns this value when the CONFIG_ZBUS_ASSERT_MOCK is enabled.

int **zbus_chan_notify**(const struct *zbus_channel* *chan, *k_timeout_t* timeout)

Force a channel notification.

This routine forces the event dispatcher to notify the channel's observers even if the message has no changes. Note this function could be useful after claiming/finishing actions.

Parameters

- **chan** – The channel's reference.
- **timeout** – Waiting period to notify the channel, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- **0** – Channel notified.
- **-EBUSY** – The channel's semaphore returned without waiting.
- **-EAGAIN** – Timeout to take the channel's semaphore.
- **-ENOMEM** – There is not more buffer on the messgage buffers pool.
- **-EFAULT** – A parameter is incorrect, the notification could not be sent to one or more observer, or the function context is invalid (inside an ISR). The function only returns this value when the CONFIG_ZBUS_ASSERT_MOCK is enabled.

static inline const char *__zbus_chan_name__(const struct *zbus_channel* *chan)

Get the channel's name.

This routine returns the channel's name reference.

Parameters

- **chan** – The channel's reference.

Returns

Channel's name reference.

static inline void *zbus_chan_msg(const struct *zbus_channel* *chan)

Get the reference for a channel message directly.

This routine returns the reference of a channel message.

> **Warning**
>
> This function must only be used directly for already locked channels. This can be done inside a listener for the receiving channel or after claim a channel.

Parameters

- chan – The channel's reference.

Returns

Channel's message reference.

static inline const void *zbus_chan_const_msg(const struct *zbus_channel* *chan)

Get a constant reference for a channel message directly.

This routine returns a constant reference of a channel message. This should be used inside listeners to access the message directly. In this way zbus prevents the listener of changing the notifying channel's message during the notification process.

> **Warning**
>
> This function must only be used directly for already locked channels. This can be done inside a listener for the receiving channel or after claim a channel.

Parameters

- chan – The channel's constant reference.

Returns

A constant channel's message reference.

static inline uint16_t zbus_chan_msg_size(const struct *zbus_channel* *chan)

Get the channel's message size.

This routine returns the channel's message size.

Parameters

- chan – The channel's reference.

Returns

Channel's message size.

static inline void *zbus_chan_user_data(const struct *zbus_channel* *chan)

Get the channel's user data.

This routine returns the channel's user data.

Parameters

- chan – The channel's reference.

Returns

Channel's user data.

static inline void **zbus_chan_set_msg_sub_pool**(const struct *zbus_channel* *chan, struct *net_buf_pool* *pool)

Set the channel's msg subscriber *net_buf* pool.

Parameters

- **chan** – The channel's reference.
- **pool** – The reference to the *net_buf* memory pool.

int **zbus_chan_add_obs**(const struct *zbus_channel* *chan, const struct *zbus_observer* *obs, *k_timeout_t* timeout)

Add an observer to a channel.

This routine adds an observer to the channel.

Parameters

- **chan** – The channel's reference.
- **obs** – The observer's reference to be added.
- **timeout** – Waiting period to add an observer, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- **0** – Observer added to the channel.
- **-EALREADY** – The observer is already present in the channel's runtime observers list.
- **-ENOMEM** – Returned without waiting.
- **-EAGAIN** – Waiting period timed out.
- **-EINVAL** – Some parameter is invalid.

int **zbus_chan_rm_obs**(const struct *zbus_channel* *chan, const struct *zbus_observer* *obs, *k_timeout_t* timeout)

Remove an observer from a channel.

This routine removes an observer to the channel.

Parameters

- **chan** – The channel's reference.
- **obs** – The observer's reference to be removed.
- **timeout** – Waiting period to remove an observer, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- **0** – Observer removed to the channel.
- **-EINVAL** – Invalid data supplied.
- **-EBUSY** – Returned without waiting.
- **-EAGAIN** – Waiting period timed out.
- **-ENODATA** – no observer found in channel's runtime observer list.
- **-ENOMEM** – Returned without waiting.

int **zbus_obs_set_enable**(struct *zbus_observer* *obs, bool enabled)

Change the observer state.

This routine changes the observer state. A channel when disabled will not receive notifications from the event dispatcher.

Parameters

- `obs` – **[in]** The observer's reference.
- `enabled` – **[in]** State to be. When false the observer stops to receive notifications.

Return values

- `0` – Observer set enable.
- `-EFAULT` – A parameter is incorrect, or the function context is invalid (inside an ISR). The function only returns this value when the `CONFIG_ZBUS_ASSERT_MOCK` is enabled.

static inline int **zbus_obs_is_enabled**(struct *zbus_observer* *obs, bool *enable)

Get the observer state.

This routine retrieves the observer state.

Parameters

- `obs` – **[in]** The observer's reference.
- `enable` – **[out]** The boolean output's reference.

Returns

Observer state.

int **zbus_obs_set_chan_notification_mask**(const struct *zbus_observer* *obs, const struct *zbus_channel* *chan, bool masked)

Mask notifications from a channel to an observer.

The observer can mask notifications from a specific observing channel by calling this function.

Parameters

- `obs` – The observer's reference to be added.
- `chan` – The channel's reference.
- `masked` – The mask state. When the mask is true, the observer will not receive notifications from the channel.

Return values

- `0` – Channel notifications masked to the observer.
- `-ESRCH` – No observation found for the related pair chan/obs.
- `-EINVAL` – Some parameter is invalid.

int **zbus_obs_is_chan_notification_masked**(const struct *zbus_observer* *obs, const struct *zbus_channel* *chan, bool *masked)

Get the notifications masking state from a channel to an observer.

Parameters

- `obs` – The observer's reference to be added.
- `chan` – The channel's reference.
- `masked` – **[out]** The mask state. When the mask is true, the observer will not receive notifications from the channel.

Return values

- `0` – Retrieved the masked state.
- `-ESRCH` – No observation found for the related pair chan/obs.

- **-EINVAL** – Some parameter is invalid.

static inline const char ***zbus_obs_name**(const struct *zbus_observer* *obs)

Get the observer's name.

This routine returns the observer's name reference.

Parameters

- **obs** – The observer's reference.

Returns

The observer's name reference.

int **zbus_obs_attach_to_thread**(const struct *zbus_observer* *obs)

Set the observer thread priority by attaching it to a thread.

Parameters

- **obs** – **[in]** The observer's reference.

Return values

- **0** – Observer detached from the thread.
- **-EFAULT** – A parameter is incorrect, or the function context is invalid (inside an ISR). The function only returns this value when the CONFIG_ZBUS_ASSERT_MOCK is enabled.

int **zbus_obs_detach_from_thread**(const struct *zbus_observer* *obs)

Clear the observer thread priority by detaching it from a thread.

Parameters

- **obs** – **[in]** The observer's reference.

Return values

- **0** – Observer detached from the thread.
- **-EFAULT** – A parameter is incorrect, or the function context is invalid (inside an ISR). The function only returns this value when the CONFIG_ZBUS_ASSERT_MOCK is enabled.

int **zbus_sub_wait**(const struct *zbus_observer* *sub, const struct *zbus_channel* **chan, *k_timeout_t* timeout)

Wait for a channel notification.

This routine makes the subscriber to wait a notification. The notification comes as a channel reference.

Parameters

- **sub** – **[in]** The subscriber's reference.
- **chan** – **[out]** The notification channel's reference.
- **timeout** – **[in]** Waiting period for a notification arrival, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- **0** – Notification received.
- **-ENOMSG** – Returned without waiting.
- **-EAGAIN** – Waiting period timed out.
- **-EINVAL** – The observer is not a subscriber.

- `-EFAULT` – A parameter is incorrect, or the function context is invalid (inside an ISR). The function only returns this value when the `CONFIG_ZBUS_ASSERT_MOCK` is enabled.

int **zbus_sub_wait_msg**(const struct *zbus_observer* *sub, const struct *zbus_channel* **chan, void *msg, *k_timeout_t* timeout)

Wait for a channel message.

This routine makes the subscriber wait for the new message in case of channel publication.

Parameters

- `sub` – **[in]** The subscriber's reference.
- `chan` – **[out]** The notification channel's reference.
- `msg` – **[out]** A reference to a copy of the published message.
- `timeout` – **[in]** Waiting period for a notification arrival, or one of the special values, K_NO_WAIT and K_FOREVER.

Return values

- `0` – Message received.
- `-EINVAL` – The observer is not a subscriber.
- `-ENOMSG` – Could not retrieve the *net_buf* from the subscriber FIFO.
- `-EILSEQ` – Received an invalid channel reference.
- `-EFAULT` – A parameter is incorrect, or the function context is invalid (inside an ISR). The function only returns this value when the `CONFIG_ZBUS_ASSERT_MOCK` is enabled.

bool **zbus_iterate_over_channels**(bool (*iterator_func)(const struct *zbus_channel* *chan))

Iterate over channels.

Enables the developer to iterate over the channels giving to this function an iterator_func which is called for each channel. If the iterator_func returns false all the iteration stops.

Parameters

- `iterator_func` – **[in]** The function that will be execute on each iteration.

Return values

- `true` – Iterator executed for all channels.
- `false` – Iterator could not be executed. Some iterate returned false.

bool **zbus_iterate_over_channels_with_user_data**(bool (*iterator_func)(const struct *zbus_channel* *chan, void *user_data), void *user_data)

Iterate over channels with user data.

Enables the developer to iterate over the channels giving to this function an iterator_func which is called for each channel. If the iterator_func returns false all the iteration stops.

Parameters

- `iterator_func` – **[in]** The function that will be execute on each iteration.
- `user_data` – **[in]** The user data that can be passed in the function.

Return values

- `true` – Iterator executed for all channels.
- `false` – Iterator could not be executed. Some iterate returned false.

bool **zbus_iterate_over_observers**(bool (*iterator_func)(const struct *zbus_observer* *obs))

Iterate over observers.

Enables the developer to iterate over the observers giving to this function an iterator_func which is called for each observer. If the iterator_func returns false all the iteration stops.

Parameters

- `iterator_func` – **[in]** The function that will be execute on each iteration.

Return values

- `true` – Iterator executed for all channels.
- `false` – Iterator could not be executed. Some iterate returned false.

bool **zbus_iterate_over_observers_with_user_data**(bool (*iterator_func)(const struct *zbus_observer* *obs, void *user_data), void *user_data)

Iterate over observers with user data.

Enables the developer to iterate over the observers giving to this function an iterator_func which is called for each observer. If the iterator_func returns false all the iteration stops.

Parameters

- `iterator_func` – **[in]** The function that will be execute on each iteration.
- `user_data` – **[in]** The user data that can be passed in the function.

Return values

- `true` – Iterator executed for all channels.
- `false` – Iterator could not be executed. Some iterate returned false.

struct **zbus_channel_data**

#include <zbus.h> Type used to represent a channel mutable data.

Every channel has a *zbus_channel_data* structure associated.

Public Members

int16_t **observers_start_idx**

Static channel observer list start index.

Considering the ITERABLE SECTIONS allocation order.

int16_t **observers_end_idx**

Static channel observer list end index.

Considering the ITERABLE SECTIONS allocation order.

struct k_sem **sem**

Access control semaphore.

Points to the semaphore used to avoid race conditions for accessing the channel.

sys_slist_t **observers**

Channel observer list.

Represents the channel's observers list, it can be empty or have listeners and subscribers mixed in any sequence. It can be changed in runtime.

struct *net_buf_pool* ***msg_subscriber_pool**

Net buf pool for message subscribers.

It can be either the global or a separated one.

struct **zbus_channel**

#include <zbus.h> Type used to represent a channel.

Every channel has a *zbus_channel* structure associated used to control the channel access and usage.

Public Members

const char *const **name**

Channel name.

void *const **message**

Message reference.

Represents the message's reference that points to the actual shared memory region.

const size_t **message_size**

Message size.

Represents the channel's message size.

void *const **user_data**

User data available to extend zbus features.

The channel must be claimed before using this field.

bool (*const **validator**)(const void *msg, size_t msg_size)

Message validator.

Stores the reference to the function to check the message validity before actually performing the publishing. No invalid messages can be published. Every message is valid when this field is empty.

struct *zbus_channel_data* *const **data**

Mutable channel data struct.

struct **zbus_observer_data**

#include <zbus.h>

Public Members

bool **enabled**

Enabled flag.

Indicates if observer is receiving notification.

struct **zbus_observer**

#include <zbus.h> Type used to represent an observer.

Every observer has an representation structure containing the relevant information. An observer is a code portion interested in some channel. The observer can be notified synchronously or asynchronously and it is called listener and subscriber respectively. The observer can be enabled or disabled during runtime by change the enabled boolean field of the structure. The listeners have a callback function that is executed by the bus with the index of the changed channel as argument when the notification is sent. The subscribers have a message queue where the bus enqueues the index of the changed channel when a notification is sent.

See also

zbus_obs_set_enable function to properly change the observer's enabled field.

Public Members

const char *const **name**

Observer name.

enum *zbus_observer_type* **type**

Type indication.

struct *zbus_observer_data* *const **data**

Mutable observer data struct.

struct *k_msgq* *const **queue**

Observer message queue.

It turns the observer into a subscriber.

void (*const **callback**)(const struct *zbus_channel* *chan)

Observer callback function.

It turns the observer into a listener.

struct k_fifo *const **message_fifo**

Observer message FIFO.

It turns the observer into a message subscriber. It only exists if the `CONFIG_ZBUS_MSG_SUBSCRIBER` is enabled.

4.34 Miscellaneous

4.34.1 Checksum APIs

CRC

group `crc`

Enums

enum `crc_type`

CRC algorithm enumeration.

These values should be used with the *CRC* dispatch function.

Values:

enumerator `CRC4`

Use *crc4*.

enumerator `CRC4_TI`

Use *crc4_ti*.

enumerator `CRC7_BE`

Use *crc7_be*.

enumerator `CRC8`

Use *crc8*.

enumerator `CRC8_CCITT`

Use *crc8_ccitt*.

enumerator `CRC16`

Use *crc16*.

enumerator `CRC16_ANSI`

Use *crc16_ansi*.

enumerator `CRC16_CCITT`

Use *crc16_ccitt*.

enumerator `CRC16_ITU_T`

Use *crc16_itu_t*.

enumerator `CRC24_PGP`

Use *crc24_pgp*.

enumerator `CRC32_C`

Use *crc32_c*.

enumerator `CRC32_IEEE`

Use *crc32_ieee*.

Functions

uint16_t `crc16`(uint16_t poly, uint16_t seed, const uint8_t *src, size_t len)

Generic function for computing a CRC-16 without input or output reflection.

Compute CRC-16 by passing in the address of the input, the input length and polynomial used in addition to the initial value. This is O(n*8) where n is the length of the buffer provided. No reflection is performed.

Note

If you are planning to use a CRC based on poly 0x1012 the functions *crc16_itu_t()* is faster and thus recommended over this one.

Parameters

- `poly` – The polynomial to use omitting the leading x^16 coefficient
- `seed` – Initial value for the CRC computation
- `src` – Input bytes for the computation
- `len` – Length of the input in bytes

Returns

The computed CRC16 value (without any XOR applied to it)

uint16_t `crc16_reflect`(uint16_t poly, uint16_t seed, const uint8_t *src, size_t len)

Generic function for computing a CRC-16 with input and output reflection.

Compute CRC-16 by passing in the address of the input, the input length and polynomial used in addition to the initial value. This is O(n*8) where n is the length of the buffer provided. Both input and output are reflected.

The following checksums can, among others, be calculated by this function, depending on the value provided for the initial seed and the value the final calculated CRC is XORed with:

- CRC-16/ANSI, CRC-16/MODBUS, CRC-16/USB, CRC-16/IBM https://reveng.sourceforge.io/crc-catalogue/16.htm#crc.cat.crc-16-modbus poly: 0x8005 (0xA001) initial seed: 0xffff, xor output: 0x0000

Note

If you are planning to use a CRC based on poly 0x1012 the function *crc16_ccitt()* is faster and thus recommended over this one.

Parameters

- `poly` – The polynomial to use omitting the leading x^16 coefficient. Important: please reflect the poly. For example, use 0xA001 instead of 0x8005 for CRC-16-MODBUS.

- **seed** – Initial value for the CRC computation
- **src** – Input bytes for the computation
- **len** – Length of the input in bytes

Returns
The computed CRC16 value (without any XOR applied to it)

uint8_t **crc8**(const uint8_t *src, size_t len, uint8_t polynomial, uint8_t initial_value, bool reversed)

Generic function for computing CRC 8.

Compute CRC 8 by passing in the address of the input, the input length and polynomial used in addition to the initial value.

Parameters

- **src** – Input bytes for the computation
- **len** – Length of the input in bytes
- **polynomial** – The polynomial to use omitting the leading x^8 coefficient
- **initial_value** – Initial value for the CRC computation
- **reversed** – Should we use reflected/reversed values or not

Returns
The computed CRC8 value

uint16_t **crc16_ccitt**(uint16_t seed, const uint8_t *src, size_t len)

Compute the checksum of a buffer with polynomial 0x1021, reflecting input and output.

This function is able to calculate any CRC that uses 0x1021 as it polynomial and requires reflecting both the input and the output. It is a fast variant that runs in O(n) time, where n is the length of the input buffer.

The following checksums can, among others, be calculated by this function, depending on the value provided for the initial seed and the value the final calculated CRC is XORed with:

- CRC-16/CCITT, CRC-16/CCITT-TRUE, CRC-16/KERMIT https://reveng.sourceforge.io/crc-catalogue/16.htm#crc.cat.crc-16-kermit initial seed: 0x0000, xor output: 0x0000
- CRC-16/X-25, CRC-16/IBM-SDLC, CRC-16/ISO-HDLC https://reveng.sourceforge.io/crc-catalogue/16.htm#crc.cat.crc-16-ibm-sdlc initial seed: 0xffff, xor output: 0xffff

See ITU-T Recommendation V.41 (November 1988).

Note

To calculate the CRC across non-contiguous blocks use the return value from block N-1 as the seed for block N.

Parameters

- **seed** – Value to seed the CRC with
- **src** – Input bytes for the computation
- **len** – Length of the input in bytes

Returns
The computed CRC16 value (without any XOR applied to it)

uint16_t `crc16_itu_t`(uint16_t seed, const uint8_t *src, size_t len)

Compute the checksum of a buffer with polynomial 0x1021, no reflection of input or output.

This function is able to calculate any CRC that uses 0x1021 as it polynomial and requires no reflection on both the input and the output. It is a fast variant that runs in O(n) time, where n is the length of the input buffer.

The following checksums can, among others, be calculated by this function, depending on the value provided for the initial seed and the value the final calculated CRC is XORed with:

- CRC-16/XMODEM, CRC-16/ACORN, CRC-16/LTE https://reveng.sourceforge.io/crc-catalogue/16.htm#crc.cat.crc-16-xmodem initial seed: 0x0000, xor output: 0x0000
- CRC16/CCITT-FALSE, CRC-16/IBM-3740, CRC-16/AUTOSAR https://reveng.sourceforge.io/crc-catalogue/16.htm#crc.cat.crc-16-ibm-3740 initial seed: 0xffff, xor output: 0x0000
- CRC-16/GSM https://reveng.sourceforge.io/crc-catalogue/16.htm#crc.cat.crc-16-gsm initial seed: 0x0000, xor output: 0xffff

See ITU-T Recommendation V.41 (November 1988) (MSB first).

Note

To calculate the CRC across non-contiguous blocks use the return value from block N-1 as the seed for block N.

Parameters

- `seed` – Value to seed the CRC with
- `src` – Input bytes for the computation
- `len` – Length of the input in bytes

Returns
The computed CRC16 value (without any XOR applied to it)

static inline uint16_t `crc16_ansi`(const uint8_t *src, size_t len)

Compute the ANSI (or Modbus) variant of CRC-16.

The ANSI variant of CRC-16 uses 0x8005 (0xA001 reflected) as its polynomial with the initial * value set to 0xffff.

Parameters

- `src` – Input bytes for the computation
- `len` – Length of the input in bytes

Returns
The computed CRC16 value

uint32_t **crc32_ieee**(const uint8_t *data, size_t len)

Generate IEEE conform CRC32 checksum.

Parameters

- **data** – Pointer to data on which the CRC should be calculated.
- **len** – Data length.

Returns
CRC32 value.

uint32_t **crc32_ieee_update**(uint32_t crc, const uint8_t *data, size_t len)

Update an IEEE conforming CRC32 checksum.

Parameters

- **crc** – CRC32 checksum that needs to be updated.
- **data** – Pointer to data on which the CRC should be calculated.
- **len** – Data length.

Returns
CRC32 value.

uint32_t **crc32_c**(uint32_t crc, const uint8_t *data, size_t len, bool first_pkt, bool last_pkt)

Calculate CRC32C (Castagnoli) checksum.

Parameters

- **crc** – CRC32C checksum that needs to be updated.
- **data** – Pointer to data on which the CRC should be calculated.
- **len** – Data length.
- **first_pkt** – Whether this is the first packet in the stream.
- **last_pkt** – Whether this is the last packet in the stream.

Returns
CRC32 value.

uint8_t **crc8_ccitt**(uint8_t initial_value, const void *buf, size_t len)

Compute CCITT variant of CRC 8.

Normal CCITT variant of CRC 8 is using 0x07.

Parameters

- **initial_value** – Initial value for the CRC computation
- **buf** – Input bytes for the computation
- **len** – Length of the input in bytes

Returns
The computed CRC8 value

uint8_t **crc7_be**(uint8_t seed, const uint8_t *src, size_t len)

Compute the CRC-7 checksum of a buffer.

See JESD84-A441. Used by the MMC protocol. Uses 0x09 as the polynomial with no reflection. The CRC is left justified, so bit 7 of the result is bit 6 of the CRC.

Parameters

- **seed** – Value to seed the CRC with
- **src** – Input bytes for the computation

- **len** – Length of the input in bytes

Returns
The computed CRC7 value

uint8_t **crc4_ti**(uint8_t seed, const uint8_t *src, size_t len)

Compute the CRC-4 checksum of a buffer.

Used by the TMAG5170 sensor. Uses 0x03 as the polynomial with no reflection. 4 most significant bits of the CRC result will be set to zero.

Parameters

- **seed** – Value to seed the CRC with
- **src** – Input bytes for the computation
- **len** – Length of the input in bytes

Returns
The computed CRC4 value

uint8_t **crc4**(const uint8_t *src, size_t len, uint8_t polynomial, uint8_t initial_value, bool reversed)

Generic function for computing CRC 4.

Compute CRC 4 by passing in the address of the input, the input length and polynomial used in addition to the initial value. The input buffer must be aligned to a whole byte. It is guaranteed that 4 most significant bits of the result will be set to zero.

Parameters

- **src** – Input bytes for the computation
- **len** – Length of the input in bytes
- **polynomial** – The polynomial to use omitting the leading x^4 coefficient
- **initial_value** – Initial value for the CRC computation
- **reversed** – Should we use reflected/reversed values or not

Returns
The computed CRC4 value

uint32_t **crc24_pgp**(const uint8_t *data, size_t len)

Generate an OpenPGP CRC-24 checksum as defined in RFC 4880 section 6.1.

Parameters

- **data** – A pointer to the data on which the CRC will be calculated.
- **len** – Data length in bytes.

Returns
The CRC-24 value.

uint32_t **crc24_pgp_update**(uint32_t crc, const uint8_t *data, size_t len)

Update an OpenPGP CRC-24 checksum.

Parameters

- **crc** – The CRC-24 checksum that needs to be updated. The full 32-bit value of the CRC needs to be used between calls, do not mask the value to keep only the last 24 bits.
- **data** – A pointer to the data on which the CRC will be calculated.
- **len** – Data length in bytes.

Returns
The CRC-24 value. When the last buffer of data has been processed, mask the value with CRC24_FINAL_VALUE_MASK to keep only the meaningful 24 bits of the CRC result.

static inline uint32_t **crc_by_type**(enum *crc_type* type, const uint8_t *src, size_t len, uint32_t seed, uint32_t poly, bool reflect, bool first, bool last)

Compute a CRC checksum, in a generic way.

This is a dispatch function that calls the individual CRC routine determined by `type`.

For 7, 8, 16 and 24-bit CRCs, the relevant `seed` and `poly` values should be passed in via the least-significant byte(s).

Similarly, for 7, 8, 16 and 24-bit CRCs, the relevant result is stored in the least-significant byte(s) of the returned value.

Parameters

- **type** – CRC algorithm to use.
- **src** – Input bytes for the computation
- **len** – Length of the input in bytes
- **seed** – Value to seed the CRC with
- **poly** – The polynomial to use omitting the leading coefficient
- **reflect** – Should we use reflected/reversed values or not
- **first** – Whether this is the first packet in the stream.
- **last** – Whether this is the last packet in the stream.

Returns
uint32_t the computed CRC value

4.34.2 Structured Data APIs

JSON

Related code samples

AWS IoT Core MQTT
Connect to AWS IoT Core and publish messages using MQTT.

group `json`

Defines

`JSON_OBJ_DESCR_PRIM`(struct_, field_name_, type_)
Helper macro to declare a descriptor for supported primitive values.

Here's an example of use:

```
struct foo {
    int32_t some_int;
};

struct json_obj_descr foo[] = {
    JSON_OBJ_DESCR_PRIM(struct foo, some_int, JSON_TOK_NUMBER),
};
```

Parameters

- **struct_** – Struct packing the values
- **field_name_** – Field name in the struct
- **type_** – Token type for JSON value corresponding to a primitive type. Must be one of: JSON_TOK_STRING for strings, JSON_TOK_NUMBER for numbers, JSON_TOK_TRUE (or JSON_TOK_FALSE) for booleans.

JSON_OBJ_DESCR_OBJECT(struct_, field_name_, sub_descr_)

Helper macro to declare a descriptor for an object value.

Here's an example of use:

```
struct nested {
    int32_t foo;
    struct {
       int32_t baz;
    } bar;
};

struct json_obj_descr nested_bar[] = {
    { ... declare bar.baz descriptor ... },
};
struct json_obj_descr nested[] = {
    { ... declare foo descriptor ... },
    JSON_OBJ_DESCR_OBJECT(struct nested, bar, nested_bar),
};
```

Parameters

- **struct_** – Struct packing the values
- **field_name_** – Field name in the struct
- **sub_descr_** – Array of *json_obj_descr* describing the subobject

JSON_OBJ_DESCR_ARRAY(struct_, field_name_, max_len_, len_field_, elem_type_)

Helper macro to declare a descriptor for an array of primitives.

Here's an example of use:

```
struct example {
    int32_t foo[10];
    size_t foo_len;
};

struct json_obj_descr array[] = {
     JSON_OBJ_DESCR_ARRAY(struct example, foo, 10, foo_len,
                          JSON_TOK_NUMBER)
};
```

Parameters

- **struct_** – Struct packing the values
- **field_name_** – Field name in the struct
- **max_len_** – Maximum number of elements in array
- **len_field_** – Field name in the struct for the number of elements in the array
- **elem_type_** – Element type, must be a primitive type

JSON_OBJ_DESCR_OBJ_ARRAY(struct_, field_name_, max_len_, len_field_, elem_descr_, elem_descr_len_)

Helper macro to declare a descriptor for an array of objects.

Here's an example of use:

```
struct person_height {
    const char *name;
    int32_t height;
};

struct people_heights {
    struct person_height heights[10];
    size_t heights_len;
};

struct json_obj_descr person_height_descr[] = {
     JSON_OBJ_DESCR_PRIM(struct person_height, name, JSON_TOK_STRING),
     JSON_OBJ_DESCR_PRIM(struct person_height, height, JSON_TOK_NUMBER),
};

struct json_obj_descr array[] = {
     JSON_OBJ_DESCR_OBJ_ARRAY(struct people_heights, heights, 10,
                              heights_len, person_height_descr,
                              ARRAY_SIZE(person_height_descr)),
};
```

Parameters

- **struct_** – Struct packing the values
- **field_name_** – Field name in the struct containing the array
- **max_len_** – Maximum number of elements in the array
- **len_field_** – Field name in the struct for the number of elements in the array
- **elem_descr_** – Element descriptor, pointer to a descriptor array
- **elem_descr_len_** – Number of elements in elem_descr_

JSON_OBJ_DESCR_ARRAY_ARRAY(struct_, field_name_, max_len_, len_field_, elem_descr_, elem_descr_len_)

Helper macro to declare a descriptor for an array of array.

Here's an example of use:

```
struct person_height {
    const char *name;
    int32_t height;
};

struct person_heights_array {
    struct person_height heights;
}

struct people_heights {
    struct person_height_array heights[10];
    size_t heights_len;
};

struct json_obj_descr person_height_descr[] = {
    JSON_OBJ_DESCR_PRIM(struct person_height, name, JSON_TOK_STRING),
    JSON_OBJ_DESCR_PRIM(struct person_height, height, JSON_TOK_NUMBER),
};

struct json_obj_descr person_height_array_descr[] = {
    JSON_OBJ_DESCR_OBJECT(struct person_heights_array,
                          heights, person_height_descr),
};

struct json_obj_descr array_array[] = {
     JSON_OBJ_DESCR_ARRAY_ARRAY(struct people_heights, heights, 10,
                                heights_len, person_height_array_descr,
                                ARRAY_SIZE(person_height_array_descr)),
};
```

Parameters

- **struct_** – Struct packing the values
- **field_name_** – Field name in the struct containing the array
- **max_len_** – Maximum number of elements in the array
- **len_field_** – Field name in the struct for the number of elements in the array
- **elem_descr_** – Element descriptor, pointer to a descriptor array
- **elem_descr_len_** – Number of elements in elem_descr_

JSON_OBJ_DESCR_ARRAY_ARRAY_NAMED(struct_, json_field_name_, struct_field_name_, max_len_, len_field_, elem_descr_, elem_descr_len_)

Variant of JSON_OBJ_DESCR_ARRAY_ARRAY that can be used when the structure and JSON field names differ.

This is useful when the JSON field is not a valid C identifier.

See also

JSON_OBJ_DESCR_ARRAY_ARRAY

Parameters

- **struct_** – Struct packing the values
- **json_field_name_** – String, field name in JSON strings

- **struct_field_name_** – Field name in the struct containing the array
- **max_len_** – Maximum number of elements in the array
- **len_field_** – Field name in the struct for the number of elements in the array
- **elem_descr_** – Element descriptor, pointer to a descriptor array
- **elem_descr_len_** – Number of elements in elem_descr_

JSON_OBJ_DESCR_PRIM_NAMED(struct_, json_field_name_, struct_field_name_, type_)

Variant of JSON_OBJ_DESCR_PRIM that can be used when the structure and JSON field names differ.

This is useful when the JSON field is not a valid C identifier.

See also

JSON_OBJ_DESCR_PRIM

Parameters

- **struct_** – Struct packing the values.
- **json_field_name_** – String, field name in JSON strings
- **struct_field_name_** – Field name in the struct
- **type_** – Token type for JSON value corresponding to a primitive type.

JSON_OBJ_DESCR_OBJECT_NAMED(struct_, json_field_name_, struct_field_name_, sub_descr_)

Variant of JSON_OBJ_DESCR_OBJECT that can be used when the structure and JSON field names differ.

This is useful when the JSON field is not a valid C identifier.

See also

JSON_OBJ_DESCR_OBJECT

Parameters

- **struct_** – Struct packing the values
- **json_field_name_** – String, field name in JSON strings
- **struct_field_name_** – Field name in the struct
- **sub_descr_** – Array of *json_obj_descr* describing the subobject

JSON_OBJ_DESCR_ARRAY_NAMED(struct_, json_field_name_, struct_field_name_, max_len_, len_field_, elem_type_)

Variant of JSON_OBJ_DESCR_ARRAY that can be used when the structure and JSON field names differ.

This is useful when the JSON field is not a valid C identifier.

See also

JSON_OBJ_DESCR_ARRAY

Parameters

- **struct_** – Struct packing the values
- **json_field_name_** – String, field name in JSON strings
- **struct_field_name_** – Field name in the struct
- **max_len_** – Maximum number of elements in array
- **len_field_** – Field name in the struct for the number of elements in the array
- **elem_type_** – Element type, must be a primitive type

JSON_OBJ_DESCR_OBJ_ARRAY_NAMED(struct_, json_field_name_, struct_field_name_, max_len_, len_field_, elem_descr_, elem_descr_len_)

Variant of JSON_OBJ_DESCR_OBJ_ARRAY that can be used when the structure and JSON field names differ.

This is useful when the JSON field is not a valid C identifier.

Here's an example of use:

```
struct person_height {
    const char *name;
    int32_t height;
};

struct people_heights {
    struct person_height heights[10];
    size_t heights_len;
};

struct json_obj_descr person_height_descr[] = {
     JSON_OBJ_DESCR_PRIM(struct person_height, name, JSON_TOK_STRING),
     JSON_OBJ_DESCR_PRIM(struct person_height, height, JSON_TOK_NUMBER),
};

struct json_obj_descr array[] = {
     JSON_OBJ_DESCR_OBJ_ARRAY_NAMED(struct people_heights,
                                    "people-heights", heights,
                                    10, heights_len,
                                    person_height_descr,
                                    ARRAY_SIZE(person_height_descr)),
};
```

Parameters

- **struct_** – Struct packing the values
- **json_field_name_** – String, field name of the array in JSON strings
- **struct_field_name_** – Field name in the struct containing the array
- **max_len_** – Maximum number of elements in the array
- **len_field_** – Field name in the struct for the number of elements in the array

- **elem_descr_** – Element descriptor, pointer to a descriptor array
- **elem_descr_len_** – Number of elements in elem_descr_

Typedefs

typedef int (*json_append_bytes_t)(const char *bytes, size_t len, void *data)

Function pointer type to append bytes to a buffer while encoding JSON data.

Param bytes
Contents to write to the output

Param len
Number of bytes to append to output

Param data
User-provided pointer

Return
This callback function should return a negative number on error (which will be propagated to the return value of *json_obj_encode()*), or 0 on success.

Enums

enum json_tokens

Values:

enumerator JSON_TOK_NONE = '_'

enumerator JSON_TOK_OBJECT_START = '{'

enumerator JSON_TOK_OBJECT_END = '}'

enumerator JSON_TOK_ARRAY_START = '['

enumerator JSON_TOK_ARRAY_END = ']'

enumerator JSON_TOK_STRING = '"'

enumerator JSON_TOK_COLON = ':'

enumerator JSON_TOK_COMMA = ','

enumerator JSON_TOK_NUMBER = '0'

enumerator JSON_TOK_FLOAT = '1'

enumerator JSON_TOK_OPAQUE = '2'

enumerator JSON_TOK_OBJ_ARRAY = '3'

enumerator `JSON_TOK_ENCODED_OBJ` = '4'

enumerator `JSON_TOK_TRUE` = 't'

enumerator `JSON_TOK_FALSE` = 'f'

enumerator `JSON_TOK_NULL` = 'n'

enumerator `JSON_TOK_ERROR` = '!'

enumerator `JSON_TOK_EOF` = '\0'

Functions

int64_t **json_obj_parse**(char *json, size_t len, const struct *json_obj_descr* *descr, size_t descr_len, void *val)

Parses the JSON-encoded object pointed to by *json*, with size *len*, according to the descriptor pointed to by *descr*.

Values are stored in a struct pointed to by *val*. Set up the descriptor like this:

struct s { int32_t foo; char *bar; } struct *json_obj_descr* descr[] = { *JSON_OBJ_DESCR_PRIM(struct s, foo, JSON_TOK_NUMBER)*, *JSON_OBJ_DESCR_PRIM(struct s, bar, JSON_TOK_STRING)*, };

Since this parser is designed for machine-to-machine communications, some liberties were taken to simplify the design: (1) strings are not unescaped (but only valid escape sequences are accepted); (2) no UTF-8 validation is performed; and (3) only integer numbers are supported (no strtod() in the minimal libc).

Parameters

- `json` – Pointer to JSON-encoded value to be parsed
- `len` – Length of JSON-encoded value
- `descr` – Pointer to the descriptor array
- `descr_len` – Number of elements in the descriptor array. Must be less than 63 due to implementation detail reasons (if more fields are necessary, use two descriptors)
- `val` – Pointer to the struct to hold the decoded values

Returns

< 0 if error, bitmap of decoded fields on success (bit 0 is set if first field in the descriptor has been properly decoded, etc).

int **json_arr_parse**(char *json, size_t len, const struct *json_obj_descr* *descr, void *val)

Parses the JSON-encoded array pointed to by *json*, with size *len*, according to the descriptor pointed to by *descr*.

Values are stored in a struct pointed to by *val*. Set up the descriptor like this:

struct s { int32_t foo; char *bar; } struct *json_obj_descr* descr[] = { *JSON_OBJ_DESCR_PRIM(struct s, foo, JSON_TOK_NUMBER)*, *JSON_OBJ_DESCR_PRIM(struct s, bar, JSON_TOK_STRING)*, }; struct a { struct s baz[10]; size_t count; } struct *json_obj_descr* array[] = { *JSON_OBJ_DESCR_OBJ_ARRAY(struct a, baz, 10, count,descr, ARRAY_SIZE(descr))*, };

Since this parser is designed for machine-to-machine communications, some liberties were taken to simplify the design: (1) strings are not unescaped (but only valid escape sequences are accepted); (2) no UTF-8 validation is performed; and (3) only integer numbers are supported (no strtod() in the minimal libc).

Parameters

- `json` – Pointer to JSON-encoded array to be parsed
- `len` – Length of JSON-encoded array
- `descr` – Pointer to the descriptor array
- `val` – Pointer to the struct to hold the decoded values

Returns

0 if array has been successfully parsed. A negative value indicates an error (as defined on errno.h).

int `json_arr_separate_object_parse_init`(struct *json_obj* *json, char *payload, size_t len)

Initialize single-object array parsing.

JSON-encoded array data is going to be parsed one object at a time. Data is provided by *payload* with the size of *len* bytes.

Function validate that Json Array start is detected and initialize *json* object for Json object parsing separately.

Parameters

- `json` – Provide storage for parser states. To be used when parsing the array.
- `payload` – Pointer to JSON-encoded array to be parsed
- `len` – Length of JSON-encoded array

Returns

0 if array start is detected and initialization is successful or negative error code in case of failure.

int `json_arr_separate_parse_object`(struct *json_obj* *json, const struct *json_obj_descr* *descr, size_t descr_len, void *val)

Parse a single object from array.

Parses the JSON-encoded object pointed to by *json* object array, with size *len*, according to the descriptor pointed to by *descr*.

Parameters

- `json` – Pointer to JSON-object message state
- `descr` – Pointer to the descriptor array
- `descr_len` – Number of elements in the descriptor array. Must be less than 31.
- `val` – Pointer to the struct to hold the decoded values

Returns

< 0 if error, 0 for end of message, bitmap of decoded fields on success (bit 0 is set if first field in the descriptor has been properly decoded, etc).

ssize_t `json_escape`(char *str, size_t *len, size_t buf_size)

Escapes the string so it can be used to encode JSON objects.

Parameters

- **str** – The string to escape; the escape string is stored the buffer pointed to by this parameter
- **len** – Points to a size_t containing the size before and after the escaping process
- **buf_size** – The size of buffer str points to

Returns
0 if string has been escaped properly, or -ENOMEM if there was not enough space to escape the buffer

size_t **json_calc_escaped_len**(const char *str, size_t len)

Calculates the JSON-escaped string length.

Parameters

- **str** – The string to analyze
- **len** – String size

Returns
The length str would have if it were escaped

ssize_t **json_calc_encoded_len**(const struct *json_obj_descr* *descr, size_t descr_len, const void *val)

Calculates the string length to fully encode an object.

Parameters

- **descr** – Pointer to the descriptor array
- **descr_len** – Number of elements in the descriptor array
- **val** – Struct holding the values

Returns
Number of bytes necessary to encode the values if >0, an error code is returned.

ssize_t **json_calc_encoded_arr_len**(const struct *json_obj_descr* *descr, const void *val)

Calculates the string length to fully encode an array.

Parameters

- **descr** – Pointer to the descriptor array
- **val** – Struct holding the values

Returns
Number of bytes necessary to encode the values if >0, an error code is returned.

int **json_obj_encode_buf**(const struct *json_obj_descr* *descr, size_t descr_len, const void *val, char *buffer, size_t buf_size)

Encodes an object in a contiguous memory location.

Parameters

- **descr** – Pointer to the descriptor array
- **descr_len** – Number of elements in the descriptor array
- **val** – Struct holding the values
- **buffer** – Buffer to store the JSON data
- **buf_size** – Size of buffer, in bytes, with space for the terminating NUL character

Returns
0 if object has been successfully encoded. A negative value indicates an error (as defined on errno.h).

int **json_arr_encode_buf**(const struct *json_obj_descr* *descr, const void *val, char *buffer, size_t buf_size)

Encodes an array in a contiguous memory location.

Parameters

- **descr** – Pointer to the descriptor array
- **val** – Struct holding the values
- **buffer** – Buffer to store the JSON data
- **buf_size** – Size of buffer, in bytes, with space for the terminating NUL character

Returns
0 if object has been successfully encoded. A negative value indicates an error (as defined on errno.h).

int **json_obj_encode**(const struct *json_obj_descr* *descr, size_t descr_len, const void *val, *json_append_bytes_t* append_bytes, void *data)

Encodes an object using an arbitrary writer function.

Parameters

- **descr** – Pointer to the descriptor array
- **descr_len** – Number of elements in the descriptor array
- **val** – Struct holding the values
- **append_bytes** – Function to append bytes to the output
- **data** – Data pointer to be passed to the append_bytes callback function.

Returns
0 if object has been successfully encoded. A negative value indicates an error.

int **json_arr_encode**(const struct *json_obj_descr* *descr, const void *val, *json_append_bytes_t* append_bytes, void *data)

Encodes an array using an arbitrary writer function.

Parameters

- **descr** – Pointer to the descriptor array
- **val** – Struct holding the values
- **append_bytes** – Function to append bytes to the output
- **data** – Data pointer to be passed to the append_bytes callback function.

Returns
0 if object has been successfully encoded. A negative value indicates an error.

struct **json_token**

#include <json.h>

struct **json_lexer**

#include <json.h>

struct **json_obj**

#include <json.h>

struct **json_obj_token**

#include <json.h>

struct **json_obj_descr**

#include <json.h>

JWT

JSON Web Tokens (JWT) are an open, industry standard [RFC 7519](https://tools.ietf.org/html/rfc7519) method for representing claims securely between two parties. Although JWT is fairly flexible, this API is limited to creating the simplistic tokens needed to authenticate with the Google Core IoT infrastructure.

group **jwt**

JSON Web Token (JWT)

Functions

int **jwt_init_builder**(struct *jwt_builder* *builder, char *buffer, size_t buffer_size)

Initialize the JWT builder.

Initialize the given JWT builder for the creation of a fresh token. The buffer size should at least be as long as JWT_BUILDER_MAX_SIZE returns.

Parameters

- **builder** – The builder to initialize.
- **buffer** – The buffer to write the token to.
- **buffer_size** – The size of this buffer. The token will be NULL terminated, which needs to be allowed for in this size.

Return values

- **0** – Success
- **-ENOSPC** – Buffer is insufficient to initialize

int **jwt_add_payload**(struct *jwt_builder* *builder, int32_t exp, int32_t iat, const char *aud)

add JWT primary payload.

int **jwt_sign**(struct *jwt_builder* *builder, const char *der_key, size_t der_key_len)

Sign the JWT token.

static inline size_t **jwt_payload_len**(struct *jwt_builder* *builder)

struct **jwt_builder**

#include <jwt.h> JWT data tracking.

JSON Web Tokens contain several sections, each encoded in base-64. This structure tracks the token as it is being built, including limits on the amount of available space. It should be initialized with jwt_init().

Public Members

char *base
: The base of the buffer we are writing to.

char *buf
: The place in this buffer where we are currently writing.

size_t len
: The length remaining to write.

bool overflowed
: Flag that is set if we try to write past the end of the buffer.

  If set, the token is not valid.

Chapter 5

Build and Configuration Systems

5.1 Build System (CMake)

CMake is used to build your application together with the Zephyr kernel. A CMake build is done in two stages. The first stage is called **configuration**. During configuration, the CMakeLists.txt build scripts are executed. After configuration is finished, CMake has an internal model of the Zephyr build, and can generate build scripts that are native to the host platform.

CMake supports generating scripts for several build systems, but only Ninja and Make are tested and supported by Zephyr. After configuration, you begin the **build** stage by executing the generated build scripts. These build scripts can recompile the application without involving CMake following most code changes. However, after certain changes, the configuration step must be executed again before building. The build scripts can detect some of these situations and reconfigure automatically, but there are cases when this must be done manually.

Zephyr uses CMake's concept of a 'target' to organize the build. A target can be an executable, a library, or a generated file. For application developers, the library target is the most important to understand. All source code that goes into a Zephyr build does so by being included in a library target, even application code.

Library targets have source code, that is added through CMakeLists.txt build scripts like this:

```
target_sources(app PRIVATE src/main.c)
```

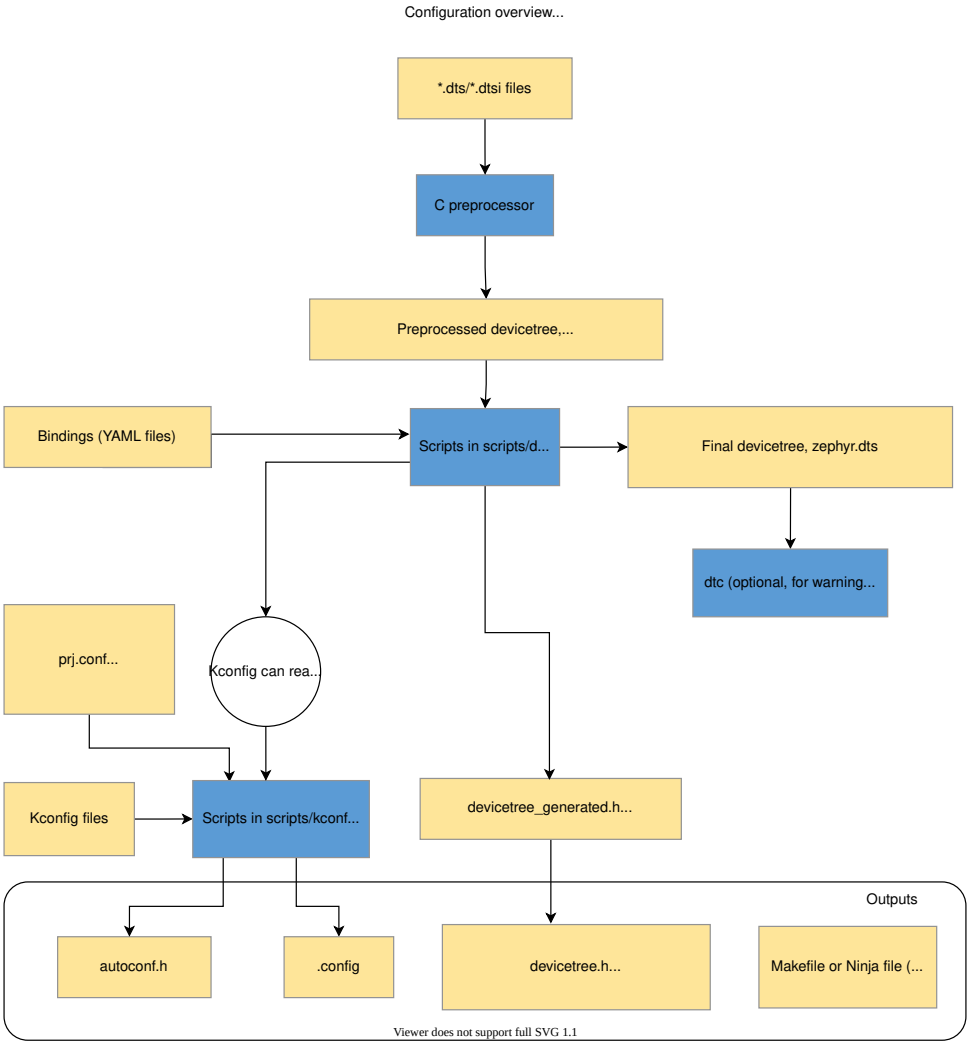
In the above `CMakeLists.txt`, an existing library target named `app` is configured to include the source file `src/main.c`. The `PRIVATE` keyword indicates that we are modifying the internals of how the library is being built. Using the keyword `PUBLIC` would modify how other libraries that link with app are built. In this case, using `PUBLIC` would cause libraries that link with `app` to also include the source file `src/main.c`, behavior that we surely do not want. The `PUBLIC` keyword could however be useful when modifying the include paths of a target library.

5.1.1 Build and Configuration Phases

The Zephyr build process can be divided into two main phases: a configuration phase (driven by CMake) and a build phase (driven by Make or Ninja).

Configuration Phase

The configuration phase begins when the user invokes *CMake* to generate a build system, specifying a source application directory and a board target.


Configuration overview...
.dts/.dtsi files
C preprocessor
Preprocessed devicetree,...
Bindings (YAML files)
Scripts in scripts/d...
Final devicetree, zephyr.dts
dtc (optional, for warning...
prj.conf...
Kconfig can rea...
Kconfig files
Scripts in scripts/kconf...
devicetree_generated.h...
Outputs
autoconf.h
.config
devicetree.h...
Makefile or Ninja file (...
Viewer does not support full SVG 1.1

CMake begins by processing the `CMakeLists.txt` file in the application directory, which refers to the `CMakeLists.txt` file in the Zephyr top-level directory, which in turn refers to `CMakeLists.txt` files throughout the build tree (directly and indirectly). Its primary output is a set of Makefiles or Ninja files to drive the build process, but the CMake scripts also do some processing of their own, which is explained here.

Note that paths beginning with `build/` below refer to the build directory you create when running CMake.

Devicetree

`*.dts` (*devicetree source*) and `*.dtsi` (*devicetree source include*) files are collected from the target's architecture, SoC, board, and application directories.

`*.dtsi` files are included by `*.dts` files via the C preprocessor (often abbreviated *cpp*, which should not be confused with C++). The C preprocessor is also used to merge in any devicetree `*.overlay` files, and to expand macros in `*.dts`, `*.dtsi`, and `*.overlay` files. The preprocessor output is placed in `build/zephyr/zephyr.dts.pre`.

The preprocessed devicetree sources are parsed by gen_defines.py to generate a `build/zephyr/include/generated/zephyr/devicetree_generated.h` header with preprocessor macros.

Source code should access preprocessor macros generated from devicetree by including the devicetree.h header, which includes `devicetree_generated.h`.

`gen_defines.py` also writes the final devicetree to `build/zephyr/zephyr.dts` in the build directory. This file's contents may be useful for debugging.

If the devicetree compiler `dtc` is installed, it is run on `build/zephyr/zephyr.dts` to catch any extra warnings and errors generated by this tool. The output from `dtc` is unused otherwise, and this step is skipped if `dtc` is not installed.

The above is just a brief overview. For more information on devicetree, see *Devicetree Guide*.

Kconfig

`Kconfig` files define available configuration options for the target architecture, SoC, board, and application, as well as dependencies between options.

Kconfig configurations are stored in *configuration files*. The initial configuration is generated by merging configuration fragments from the board and application (e.g. `prj.conf`).

The output from Kconfig is an `autoconf.h` header with preprocessor assignments, and a `.config` file that acts both as a saved configuration and as configuration output (used by CMake). The definitions in `autoconf.h` are automatically exposed at compile time, so there is no need to include this header.

Information from devicetree is available to Kconfig, through the functions defined in kconfigfunctions.py.

See *the Kconfig section of the manual* for more information.

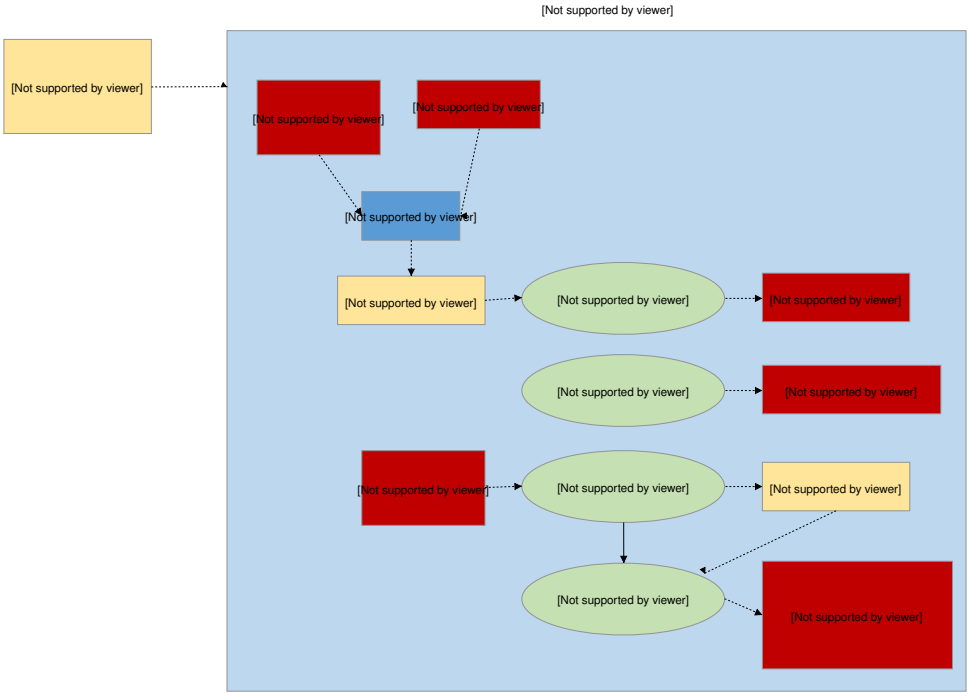
Build Phase

The build phase begins when the user invokes `make` or `ninja`. Its ultimate output is a complete Zephyr application in a format suitable for loading/flashing on the desired target board (`zephyr.elf`, `zephyr.hex`, etc.) The build phase can be broken down, conceptually, into four stages: the pre-build, first-pass binary, final binary, and post-processing.

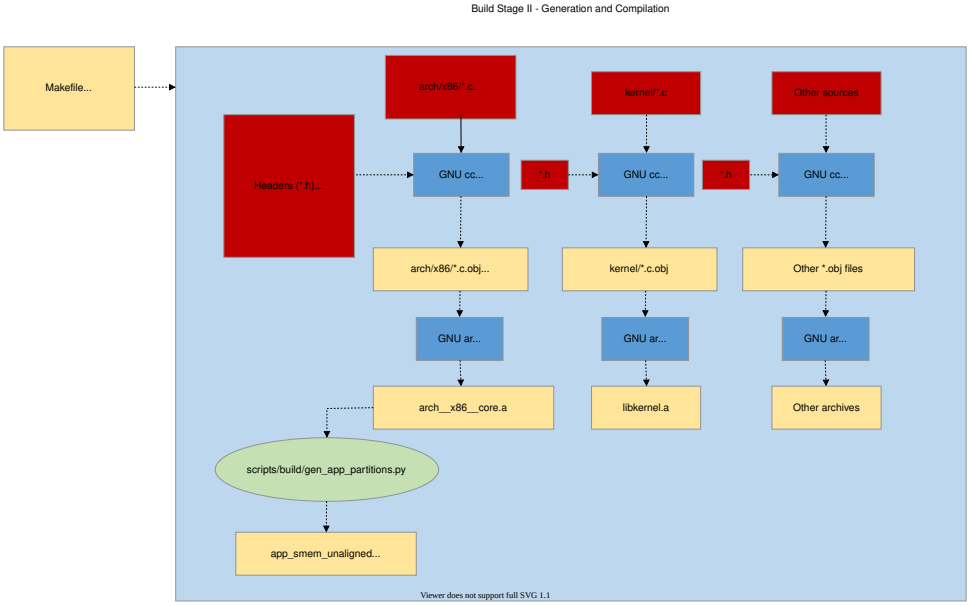
Pre-build Pre-build occurs before any source files are compiled, because during this phase header files used by the source files are generated.

Offset generation
Access to high-level data structures and members is sometimes required when the definitions of those structures is not immediately accessible (e.g., assembly language). The generation of *offsets.h* (by *gen_offset_header.py*) facilitates this.

System call boilerplate
The *gen_syscall.py* and *parse_syscalls.py* scripts work together to bind potential system call functions with their implementations.



Intermediate binaries Compilation proper begins with the first intermediate binary. Source files (C and assembly) are collected from various subsystems (which ones is decided during the configuration phase), and compiled into archives (with reference to header files in the tree, as well as those generated during the configuration phase and the pre-build stage(s)).



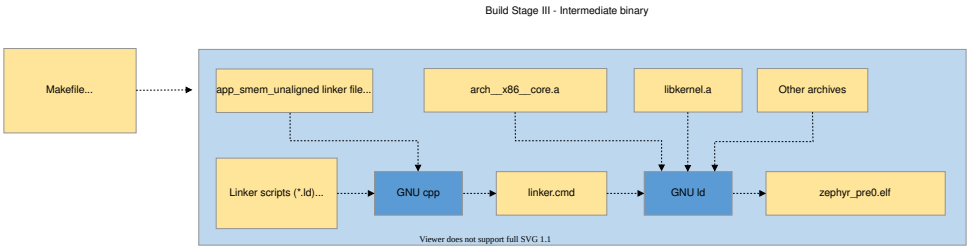
The exact number of intermediate binaries is decided during the configuration phase.

If memory protection is enabled, then:

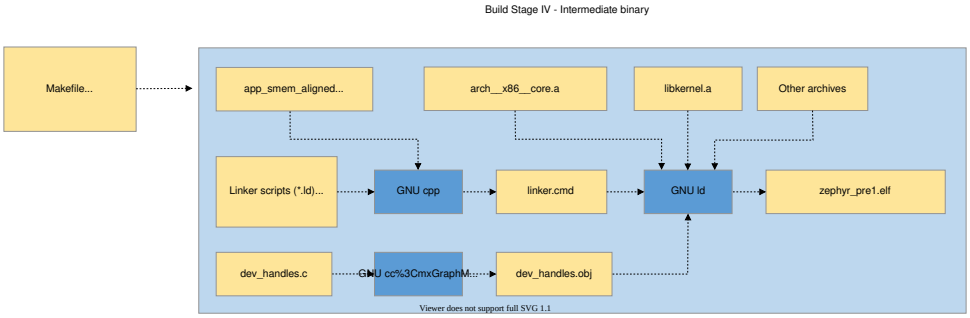
Partition grouping
The *gen_app_partitions.py* script scans all the generated archives and outputs linker scripts to ensure that application partitions are properly grouped and aligned for the target's memory protection hardware.

Then *cpp* is used to combine linker script fragments from the target's architecture/SoC, the kernel tree, optionally the partition output if memory protection is enabled, and any other fragments selected during the configuration process, into a *linker.cmd* file. The compiled archives are then linked with *ld* as specified in the *linker.cmd*.

Unfixed size binary
The unfixed size intermediate binary is produced when *User Mode* is enabled or *Devicetree* is in use. It produces a binary where sizes are not fixed and thus it may be used by post-process steps that will impact the size of the final binary.



Fixed size binary
The fixed size intermediate binary is produced when *User Mode* is enabled or when generated IRQ tables are used, `CONFIG_GEN_ISR_TABLES` It produces a binary where sizes are fixed and thus the size must not change between the intermediate binary and the final binary.

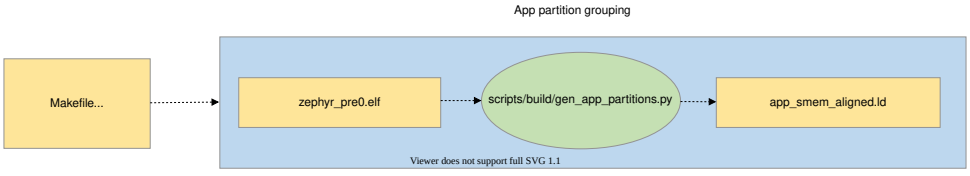


Intermediate binaries post-processing The binaries from the previous stage are incomplete, with empty and/or placeholder sections that must be filled in by, essentially, reflection.

To complete the build procedure the following scripts are executed on the intermediate binaries to produce the missing pieces needed for the final binary.

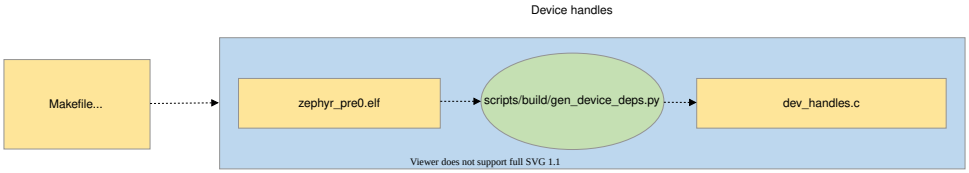
When *User Mode* is enabled:

Partition alignment
The *gen_app_partitions.py* script scans the unfixed size binary and generates an app shared memory aligned linker script snippet where the partitions are sorted in descending order.

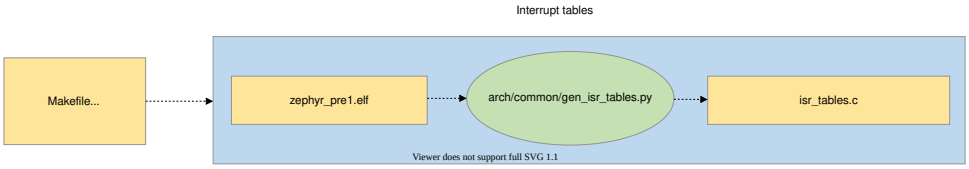


When *Devicetree* is used:

Device dependencies
The *gen_device_deps.py* script scans the unfixed size binary to determine relationships between devices that were recorded from devicetree data, and replaces the encoded relationships with values that are optimized to locate the devices actually present in the application.

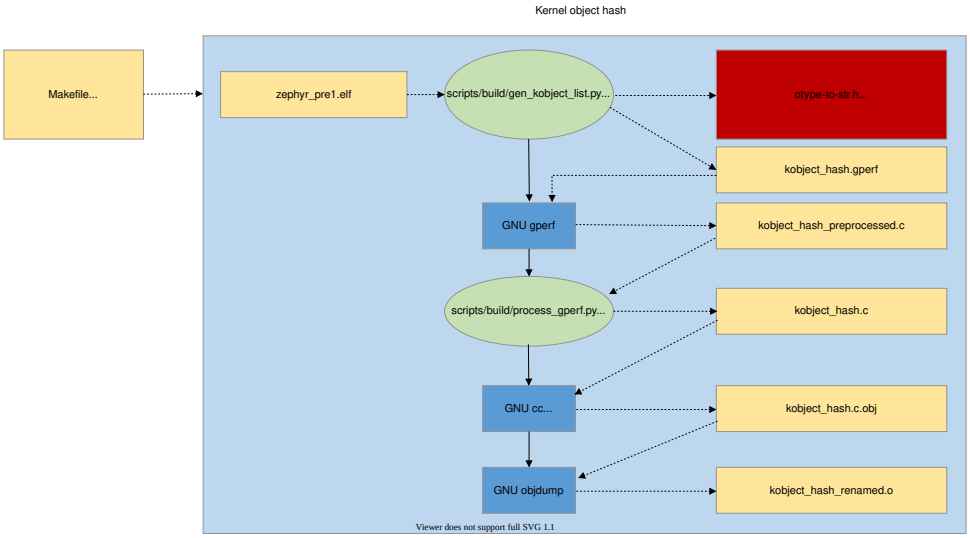


When `CONFIG_GEN_ISR_TABLES` is enabled:
The *gen_isr_tables.py* script scans the fixed size binary and creates an isr_tables.c source file with a hardware vector table and/or software IRQ table.



When *User Mode* is enabled:

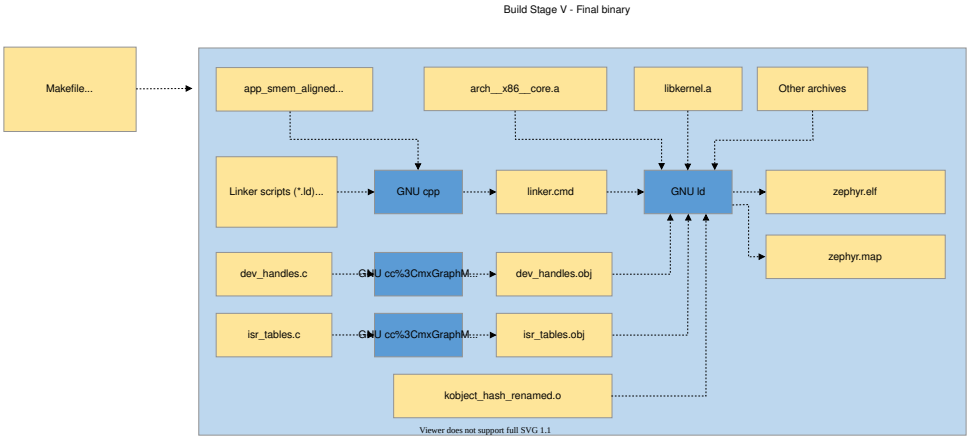
Kernel object hashing
The *gen_kobject_list.py* scans the *ELF DWARF* debug data to find the address of the all kernel objects. This list is passed to *gperf*, which generates a perfect hash function and table of those addresses, then that output is optimized by *process_gperf.py*, using known properties of our special case.



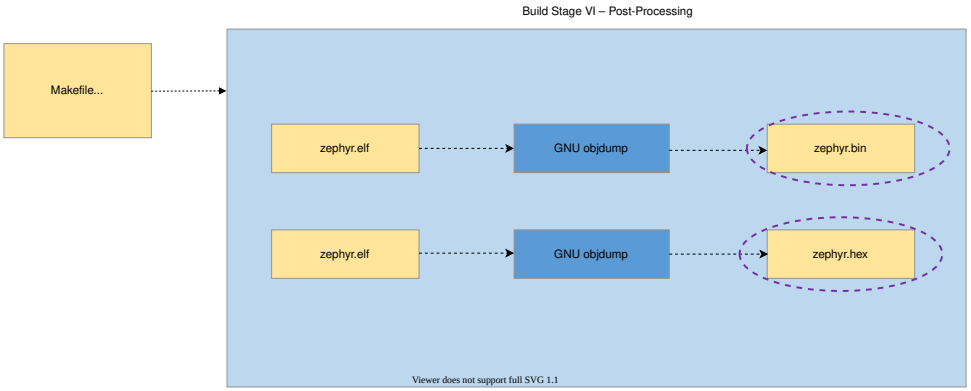
When no intermediate binary post-processing is required then the first intermediate binary will be directly used as the final binary.

Final binary The binary from the previous stage is incomplete, with empty and/or placeholder sections that must be filled in by, essentially, reflection.

The link from the previous stage is repeated, this time with the missing pieces populated.



Post processing Finally, if necessary, the completed kernel is converted from *ELF* to the format expected by the loader and/or flash tool required by the target. This is accomplished in a straightforward manner with *objdump*.



5.1.2 Supporting Scripts and Tools

The following is a detailed description of the scripts used during the build process.

scripts/build/gen_syscalls.py

Script to generate system call invocation macros

This script parses the system call metadata JSON file emitted by parse_syscalls.py to create several files:

- A file containing weak aliases of any potentially unimplemented system calls, as well as the system call dispatch table, which maps system call type IDs to their handler functions.
- A header file defining the system call type IDs, as well as function prototypes for all system call handler functions.
- A directory containing header files. Each header corresponds to a header that was identified as containing system call declarations. These generated headers contain the inline invocation functions for each system call in that header.

scripts/build/gen_device_deps.py

Translate generic handles into ones optimized for the application.

Immutable device data includes information about dependencies, e.g. that a particular sensor is controlled through a specific I2C bus and that it signals event on a pin on a specific GPIO controller. This information is encoded in the first-pass binary using identifiers derived from the devicetree. This script extracts those identifiers and replaces them with ones optimized for use with the devices actually present.

For example the sensor might have a first-pass handle defined by its devicetree ordinal 52, with the I2C driver having ordinal 24 and the GPIO controller ordinal 14. The runtime ordinal is the index of the corresponding device in the static devicetree array, which might be 6, 5, and 3, respectively.

The output is a C source file that provides alternative definitions for the array contents referenced from the immutable device objects. In the final link these definitions supersede the ones in the driver-specific object file.

scripts/build/gen_kobject_list.py

Script to generate gperf tables of kernel object metadata

User mode threads making system calls reference kernel objects by memory address, as the kernel/driver APIs in Zephyr are the same for both user and supervisor contexts. It is necessary for the kernel to be able to validate accesses to kernel objects to make the following assertions:

- That the memory address points to a kernel object
- The kernel object is of the expected type for the API being invoked
- The kernel object is of the expected initialization state
- The calling thread has sufficient permissions on the object

For more details see the *Kernel Objects* section in the documentation.

The zephyr build generates an intermediate ELF binary, zephyr_prebuilt.elf, which this script scans looking for kernel objects by examining the DWARF debug information to look for instances of data structures that are considered kernel objects. For device drivers, the API struct pointer populated at build time is also examined to disambiguate between various device driver instances since they are all 'struct device'.

This script can generate five different output files:

- A gperf script to generate the hash table mapping kernel object memory addresses to kernel object metadata, used to track permissions, object type, initialization state, and any object-specific data.
- A header file containing generated macros for validating driver instances inside the system call handlers for the driver subsystem APIs.
- A code fragment included by kernel.h with one enum constant for each kernel object type and each driver instance.
- The inner cases of a switch/case C statement, included by kernel/userspace.c, mapping the kernel object types and driver instances to their human-readable representation in the otype_to_str() function.
- The inner cases of a switch/case C statement, included by kernel/userspace.c, mapping kernel object types to their sizes. This is used for allocating instances of them at runtime (CONFIG_DYNAMIC_OBJECTS) in the obj_size_get() function.

scripts/build/gen_offset_header.py

This script scans a specified object file and generates a header file that defined macros for the offsets of various found structure members (particularly symbols ending with `_OFFSET` or `_SIZEOF`),

primarily intended for use in assembly code.

scripts/build/parse_syscalls.py

Script to scan Zephyr include directories and emit system call and subsystem metadata

System calls require a great deal of boilerplate code in order to implement completely. This script is the first step in the build system's process of auto-generating this code by doing a text scan of directories containing C or header files, and building up a database of system calls and their function call prototypes. This information is emitted to a generated JSON file for further processing.

This script also scans for struct definitions such as __subsystem and __net_socket, emitting a JSON dictionary mapping tags to all the struct declarations found that were tagged with them.

If the output JSON file already exists, its contents are checked against what information this script would have outputted; if the result is that the file would be unchanged, it is not modified to prevent unnecessary incremental builds.

arch/x86/gen_idt.py

Generate Interrupt Descriptor Table for x86 CPUs.

This script generates the interrupt descriptor table (IDT) for x86. Please consult the IA Architecture SW Developer Manual, volume 3, for more details on this data structure.

This script accepts as input the zephyr_prebuilt.elf binary, which is a link of the Zephyr kernel without various build-time generated data structures (such as the IDT) inserted into it. This kernel image has been properly padded such that inserting these data structures will not disturb the memory addresses of other symbols. From the kernel binary we read a special section "intList" which contains the desired interrupt routing configuration for the kernel, populated by instances of the IRQ_CONNECT() macro.

This script outputs three binary tables:

1. The interrupt descriptor table itself.
2. A bitfield indicating which vectors in the IDT are free for installation of dynamic interrupts at runtime.
3. An array which maps configured IRQ lines to their associated vector entries in the IDT, used to program the APIC at runtime.

arch/x86/gen_gdt.py

Generate a Global Descriptor Table (GDT) for x86 CPUs.

For additional detail on GDT and x86 memory management, please consult the IA Architecture SW Developer Manual, vol. 3.

This script accepts as input the zephyr_prebuilt.elf binary, which is a link of the Zephyr kernel without various build-time generated data structures (such as the GDT) inserted into it. This kernel image has been properly padded such that inserting these data structures will not disturb the memory addresses of other symbols.

The input kernel ELF binary is used to obtain the following information:

- Memory addresses of the Main and Double Fault TSS structures so GDT descriptors can be created for them
- Memory addresses of where the GDT lives in memory, so that this address can be populated in the GDT pseudo descriptor

- whether userspace or HW stack protection are enabled in Kconfig

The output is a GDT whose contents depend on the kernel configuration. With no memory protection features enabled, we generate flat 32-bit code and data segments. If hardware- based stack overflow protection or userspace is enabled, we additionally create descriptors for the main and double- fault IA tasks, needed for userspace privilege elevation and double-fault handling. If userspace is enabled, we also create flat code/data segments for ring 3 execution.

scripts/build/gen_relocate_app.py

This script will relocate .text, .rodata, .data and .bss sections from required files and places it in the required memory region. This memory region and file are given to this python script in the form of a string.

Example of such a string would be:

```
SRAM2:COPY:/home/xyz/zephyr/samples/hello_world/src/main.c,\
SRAM1:COPY:/home/xyz/zephyr/samples/hello_world/src/main2.c, \
FLASH2:NOCOPY:/home/xyz/zephyr/samples/hello_world/src/main3.c
```

One can also specify the program header for a given memory region:

SRAM2\ :phdr0:COPY:/home/xyz/zephyr/samples/hello_world/src/main.c

To invoke this script:

```
python3 gen_relocate_app.py -i input_string -o generated_linker -c generated_code
```

Configuration that needs to be sent to the python script.

- If the memory is like SRAM1/SRAM2/CCD/AON then place full object in the sections
- If the memory type is appended with _DATA / _TEXT/ _RODATA/ _BSS only the selected memory is placed in the required memory region. Others are ignored.
- COPY/NOCOPY defines whether the script should generate the relocation code in code_relocation.c or not
- NOKEEP will suppress the default behavior of marking every relocated symbol with KEEP() in the generated linker script.

Multiple regions can be appended together like SRAM2_DATA_BSS this will place data and bss inside SRAM2.

scripts/build/process_gperf.py

gperf C file post-processor

We use gperf to build up a perfect hashtable of pointer values. The way gperf does this is to create a table 'wordlist' indexed by a string representation of a pointer address, and then doing memcmp() on a string passed in for comparison

We are exclusively working with 4-byte pointer values. This script adjusts the generated code so that we work with pointers directly and not strings. This saves a considerable amount of space.

scripts/build/gen_app_partitions.py

Script to generate a linker script organizing application memory partitions

Applications may declare build-time memory domain partitions with K_APPMEM_PARTITION_DEFINE, and assign globals to them using K_APP_DMEM or K_APP_BMEM macros. For each of these partitions, we need to route all their data into

appropriately-sized memory areas which meet the size/alignment constraints of the memory protection hardware.

This linker script is created very early in the build process, before the build attempts to link the kernel binary, as the linker script this tool generates is a necessary pre-condition for kernel linking. We extract the set of memory partitions to generate by looking for variables which have been assigned to input sections that follow a defined naming convention. We also allow entire libraries to be pulled in to assign their globals to a particular memory partition via command line directives.

This script takes as inputs:

- The base directory to look for compiled objects
- key/value pairs mapping static library files to what partitions their globals should end up in.

The output is a linker script fragment containing the definition of the app shared memory section, which is further divided, for each partition found, into data and BSS for each partition.

scripts/build/check_init_priorities.py

Checks the initialization priorities

This script parses a Zephyr executable file, creates a list of known devices and their effective initialization priorities and compares that with the device dependencies inferred from the devicetree hierarchy.

This can be used to detect devices that are initialized in the incorrect order, but also devices that are initialized at the same priority but depends on each other, which can potentially break if the linking order is changed.

Optionally, it can also produce a human readable list of the initialization calls for the various init levels.

5.2 Devicetree

A *devicetree* is a hierarchical data structure primarily used to describe hardware. Zephyr uses devicetree in two main ways:

- to describe hardware to the *Device Driver Model*
- to provide that hardware's initial configuration

This page links to a high level guide on devicetree as well as reference material.

5.2.1 Devicetree Guide

The pages in this section are a high-level guide to using devicetree for Zephyr development.

Introduction to devicetree

> **Tip**
>
> This is a conceptual overview of devicetree and how Zephyr uses it. For step-by-step guides and examples, see *Devicetree HOWTOs*.

The following pages introduce general devicetree concepts and how they apply to Zephyr.

Scope and purpose A *devicetree* is primarily a hierarchical data structure that describes hardware. The Devicetree specification defines its source and binary representations.

Zephyr uses devicetree to describe:

- the hardware available on its boards
- that hardware's initial configuration

As such, devicetree is both a hardware description language and a configuration language for Zephyr. See *Devicetree versus Kconfig* for some comparisons between devicetree and Zephyr's other main configuration language, Kconfig.

There are two types of devicetree input files: *devicetree sources* and *devicetree bindings*. The sources contain the devicetree itself. The bindings describe its contents, including data types. The *build system* uses devicetree sources and bindings to produce a generated C header. The generated header's contents are abstracted by the `devicetree.h` API, which you can use to get information from your devicetree.

Here is a simplified view of the process:

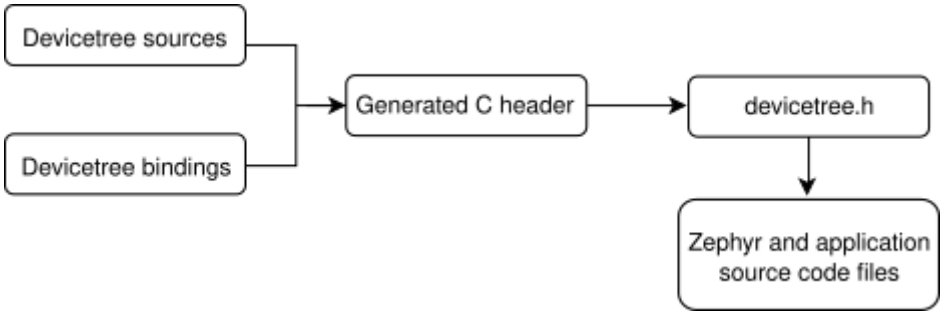


Fig. 1: Devicetree build flow

All Zephyr and application source code files can include and use `devicetree.h`. This includes *device drivers*, *applications*, *tests*, the kernel, etc.

The API itself is based on C macros. The macro names all start with `DT_`. In general, if you see a macro that starts with `DT_` in a Zephyr source file, it's probably a `devicetree.h` macro. The generated C header contains macros that start with `DT_` as well; you might see those in compiler error messages. You always can tell a generated- from a non-generated macro: generated macros have some lowercased letters, while the `devicetree.h` macro names have all capital letters.

Syntax and structure As the name indicates, a devicetree is a tree. The human-readable text format for this tree is called DTS (for devicetree source), and is defined in the Devicetree specification.

This page's purpose is to introduce devicetree in a more gradual way than the specification. However, you may still need to refer to the specification to understand some detailed cases.

Contents

- *Example*
- *Nodes*
- *Properties*

- *Devicetrees reflect hardware*
- *Properties in practice*
- *Unit addresses*
- *Important properties*
- *Writing property values*
- *Aliases and chosen nodes*

Example Here is an example DTS file:

```
/dts-v1/;

/ {
        a-node {
                subnode_nodelabel: a-sub-node {
                        foo = <3>;
                };
        };
};
```

The `/dts-v1/;` line means the file's contents are in version 1 of the DTS syntax, which has replaced a now-obsolete "version 0".

Nodes Like any tree data structure, a devicetree has a hierarchy of *nodes*. The above tree has three nodes:

1. A root node: `/`
2. A node named `a-node`, which is a child of the root node
3. A node named `a-sub-node`, which is a child of `a-node`

Nodes can be assigned *node labels*, which are unique shorthands that refer to the labeled node. Above, `a-sub-node` has the node label `subnode_nodelabel`. A node can have zero, one, or multiple node labels. You can use node labels to refer to the node elsewhere in the devicetree.

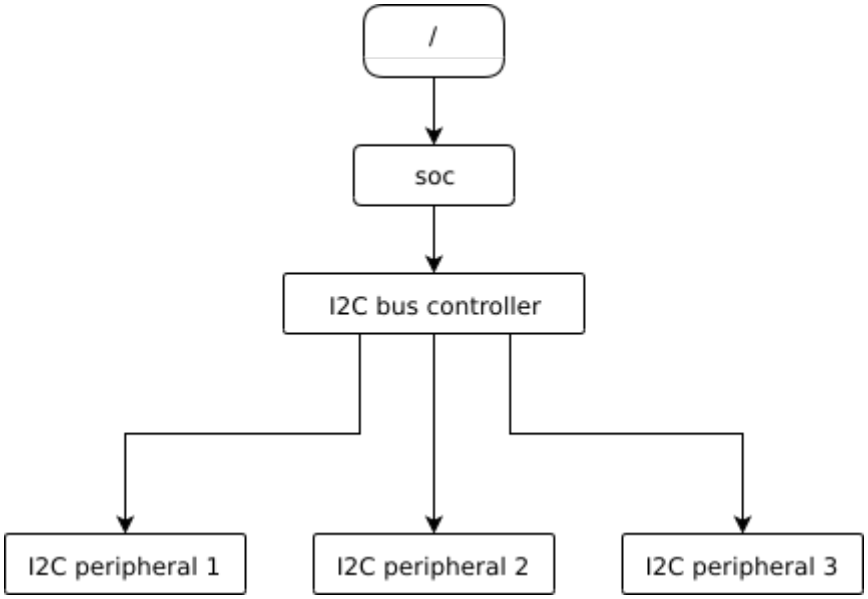
Devicetree nodes have *paths* identifying their locations in the tree. Like Unix file system paths, devicetree paths are strings separated by slashes (`/`), and the root node's path is a single slash: `/`. Otherwise, each node's path is formed by concatenating the node's ancestors' names with the node's own name, separated by slashes. For example, the full path to `a-sub-node` is `/a-node/a-sub-node`.

Properties Devicetree nodes can also have *properties*. Properties are name/value pairs. Property values can be any sequence of bytes. In some cases, the values are an array of what are called *cells*. A cell is just a 32-bit unsigned integer.

Node `a-sub-node` has a property named `foo`, whose value is a cell with value 3. The size and type of `foo`'s value are implied by the enclosing angle brackets (< and >) in the DTS.

See *Writing property values* below for more example property values.

Devicetrees reflect hardware In practice, devicetree nodes usually correspond to some hardware, and the node hierarchy reflects the hardware's physical layout. For example, let's consider a board with three I2C peripherals connected to an I2C bus controller on an SoC, like this:



Nodes corresponding to the I2C bus controller and each I2C peripheral would be present in the devicetree. Reflecting the hardware layout, the I2C peripheral nodes would be children of the bus controller node. Similar conventions exist for representing other types of hardware.

The DTS would look something like this:

```
/dts-v1/;

/ {
        soc {
                i2c-bus-controller {
                        i2c-peripheral-1 {
                        };
                        i2c-peripheral-2 {
                        };
                        i2c-peripheral-3 {
                        };
                };
        };
};
```

Properties in practice In practice, properties usually describe or configure the hardware the node represents. For example, an I2C peripheral's node has a property whose value is the peripheral's address on the bus.

Here's a tree representing the same example, but with real-world node names and properties you might see when working with I2C devices.

This is the corresponding DTS:

```
/dts-v1/;

/ {
        soc {
                i2c@40003000 {
                        compatible = "nordic,nrf-twim";
                        reg = <0x40003000 0x1000>;

                        apds9960@39 {
                                compatible = "avago,apds9960";
```

(continues on next page)

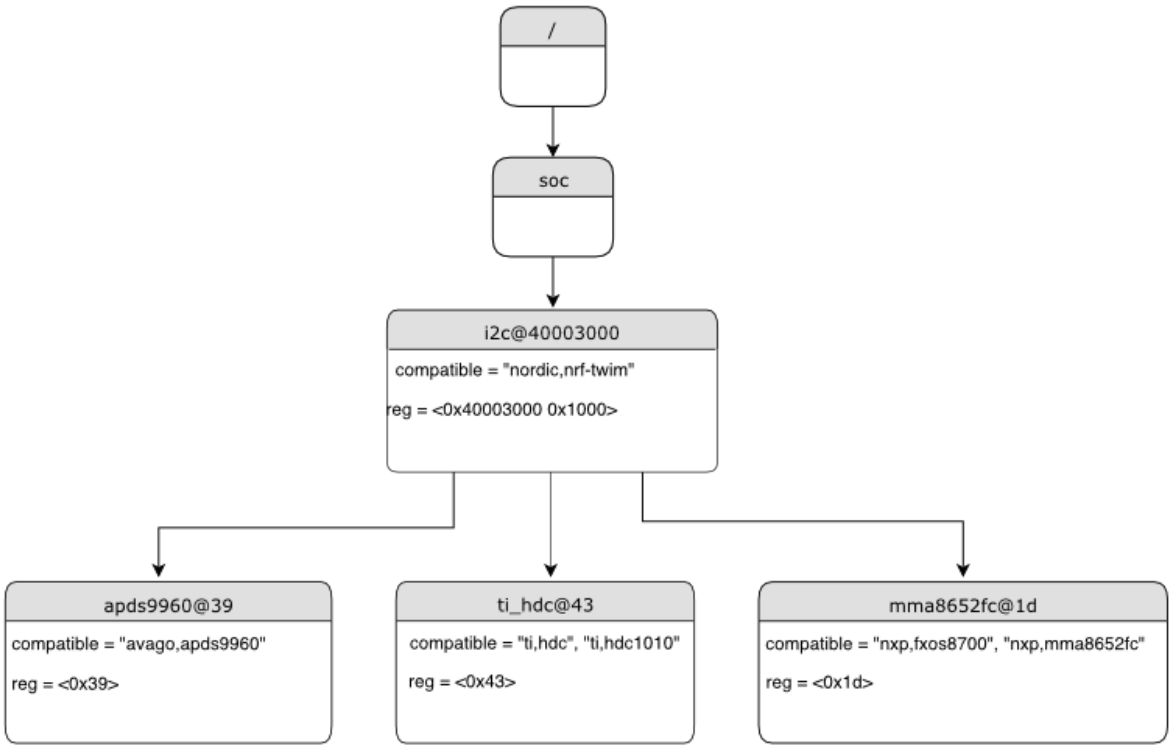


Fig. 2: I2C devicetree example with real-world names and properties. Node names are at the top of each node with a gray background. Properties are shown as "name=value" lines.

(continued from previous page)

```
                        reg = <0x39>;
                };
                ti_hdc@43 {
                        compatible = "ti,hdc", "ti,hdc1010";
                        reg = <0x43>;
                };
                mma8652fc@1d {
                        compatible = "nxp,fxos8700", "nxp,mma8652fc";
                        reg = <0x1d>;
                };
        };
};
};
```

Unit addresses In addition to showing more real-world names and properties, the above example introduces a new devicetree concept: unit addresses. Unit addresses are the parts of node names after an "at" sign (`@`), like `40003000` in `i2c@40003000`, or `39` in `apds9960@39`. Unit addresses are optional: the `soc` node does not have one.

In devicetree, unit addresses give a node's address in the address space of its parent node. Here are some example unit addresses for different types of hardware.

Memory-mapped peripherals
: The peripheral's register map base address. For example, the node named `i2c@40003000` represents an I2C controller whose register map base address is 0x40003000.

I2C peripherals
: The peripheral's address on the I2C bus. For example, the child node `apds9960@39` of the I2C controller in the previous section has I2C address 0x39.

SPI peripherals
An index representing the peripheral's chip select line number. (If there is no chip select line, 0 is used.)

Memory
The physical start address. For example, a node named `memory@2000000` represents RAM starting at physical address 0x2000000.

Memory-mapped flash
Like RAM, the physical start address. For example, a node named `flash@8000000` represents a flash device whose physical start address is 0x8000000.

Fixed flash partitions
This applies when the devicetree is used to store a flash partition table. The unit address is the partition's start offset within the flash memory. For example, take this flash device and its partitions:

```
flash@8000000 {
    /* ... */
    partitions {
            partition@0 { /* ... */ };
            partition@20000 {  /* ... */ };
            /* ... */
    };
};
```

The node named `partition@0` has offset 0 from the start of its flash device, so its base address is 0x8000000. Similarly, the base address of the node named `partition@20000` is 0x8020000.

Important properties The devicetree specification defines several standard properties. Some of the most important ones are:

compatible
The name of the hardware device the node represents.

The recommended format is `"vendor,device"`, like `"avago,apds9960"`, or a sequence of these, like `"ti,hdc", "ti,hdc1010"`. The `vendor` part is an abbreviated name of the vendor. The file dts/bindings/vendor-prefixes.txt contains a list of commonly accepted `vendor` names. The `device` part is usually taken from the datasheet.

It is also sometimes a value like `gpio-keys`, `mmio-sram`, or `fixed-clock` when the hardware's behavior is generic.

The build system uses the compatible property to find the right *bindings* for the node. Device drivers use `devicetree.h` to find nodes with relevant compatibles, in order to determine the available hardware to manage.

The `compatible` property can have multiple values. Additional values are useful when the device is a specific instance of a more general family, to allow the system to match from most- to least-specific device drivers.

Within Zephyr's bindings syntax, this property has type `string-array`.

reg
Information used to address the device. The value is specific to the device (i.e. is different depending on the compatible property).

The `reg` property is a sequence of `(address, length)` pairs. Each pair is called a "register block". Values are conventionally written in hex.

Here are some common patterns:

- Devices accessed via memory-mapped I/O registers (like `i2c@40003000`): `address` is usually the base address of the I/O register space, and `length` is the number of bytes occupied by the registers.
- I2C devices (like `apds9960@39` and its siblings): `address` is a slave address on the I2C bus. There is no `length` value.
- SPI devices: `address` is a chip select line number; there is no `length`.

You may notice some similarities between the `reg` property and common unit addresses described above. This is not a coincidence. The `reg` property can be seen as a more detailed view of the addressable resources within a device than its unit address.

status
A string which describes whether the node is enabled.

The devicetree specification allows this property to have values `"okay"`, `"disabled"`, `"reserved"`, `"fail"`, and `"fail-sss"`. Only the values `"okay"` and `"disabled"` are currently relevant to Zephyr; use of other values currently results in undefined behavior.

A node is considered enabled if its status property is either `"okay"` or not defined (i.e. does not exist in the devicetree source). Nodes with status `"disabled"` are explicitly disabled. (For backwards compatibility, the value `"ok"` is treated the same as `"okay"`, but this usage is deprecated.) Devicetree nodes which correspond to physical devices must be enabled for the corresponding `struct device` in the Zephyr driver model to be allocated and initialized.

interrupts
Information about interrupts generated by the device, encoded as an array of one or more *interrupt specifiers*. Each interrupt specifier has some number of cells. See section 2.4, *Interrupts and Interrupt Mapping*, in the Devicetree Specification release v0.3 for more details.

Zephyr's devicetree bindings language lets you give a name to each cell in an interrupt specifier.

Note

Earlier versions of Zephyr made frequent use of the `label` property, which is distinct from the standard *node label*. Use of the label *property* in new devicetree bindings, as well as use of the `DT_LABEL` macro in new code, are actively discouraged. Label properties continue to persist for historical reasons in some existing bindings and overlays, but should not be used in new bindings or device implementations.

Writing property values This section describes how to write property values in DTS format. The property types in the table below are described in detail in *Devicetree bindings*.

Some specifics are skipped in the interest of keeping things simple; if you're curious about details, see the devicetree specification.

Property type	How to write	Example
string	Double quoted	`a-string = "hello, world!";`
int	between angle brackets (< and >)	`an-int = <1>;`
boolean	for `true`, with no value (for `false`, use `/delete-property/`)	`my-true-boolean;`
array	between angle brackets (< and >), separated by spaces	`foo = <0xdeadbeef 1234 0>;`
uint8-array	in hexadecimal *without* leading `0x`, between square brackets (`[` and `]`).	`a-byte-array = [00 01 ab];`
string-array	separated by commas	`a-string-array = "string one", "string two", "string three";`
phandle	between angle brackets (< and >)	`a-phandle = <&mynode>;`
phandles	between angle brackets (< and >), separated by spaces	`some-phandles = <&mynode0 &mynode1 &mynode2>;`
phandle-array	between angle brackets (< and >), separated by spaces	`a-phandle-array = <&mynode0 1 2>, <&mynode1 3 4>;`

Additional notes on the above:

- The values in the `phandle`, `phandles`, and `phandle-array` types are described further in *Phandles*
- Boolean properties are true if present. They should not have a value. A boolean property is only false if it is completely missing in the DTS.
- The `foo` property value above has three *cells* with values 0xdeadbeef, 1234, and 0, in that order. Note that hexadecimal and decimal numbers are allowed and can be intermixed. Since Zephyr transforms DTS to C sources, it is not necessary to specify the endianness of an individual cell here.
- 64-bit integers are written as two 32-bit cells in big-endian order. The value 0xaaaa0000bbbb1111 would be written `<0xaaaa0000 0xbbbb1111>`.
- The `a-byte-array` property value is the three bytes 0x00, 0x01, and 0xab, in that order.
- Parentheses, arithmetic operators, and bitwise operators are allowed. The `bar` property contains a single cell with value 64:

  ```
  bar = <(2 * (1 << 5))>;
  ```

  Note that the entire expression must be parenthesized.
- Property values refer to other nodes in the devicetree by their *phandles*. You can write a phandle using `&foo`, where `foo` is a *node label*. Here is an example devicetree fragment:

  ```
  foo: device@0 { };
  device@1 {
          sibling = <&foo 1 2>;
  };
  ```

  The `sibling` property of node `device@1` contains three cells, in this order:

  1. The `device@0` node's phandle, which is written here as `&foo` since the `device@0` node has a node label `foo`
  2. The value 1
  3. The value 2

In the devicetree, a phandle value is a cell – which again is just a 32-bit unsigned int. However, the Zephyr devicetree API generally exposes these values as *node identifiers*. Node identifiers are covered in more detail in *Devicetree access from C/C++*.

- Array and similar type property values can be split into several <> blocks, like this:

```
foo = <1 2>, <3 4>;                          // Okay for 'type: array'
foo = <&label1 &label2>, <&label3 &label4>; // Okay for 'type: phandles'
foo = <&label1 1 2>, <&label2 3 4>;          // Okay for 'type: phandle-array'
```

This is recommended for readability when possible if the value can be logically grouped into blocks of sub-values.

Aliases and chosen nodes There are two additional ways beyond *node labels* to refer to a particular node without specifying its entire path: by alias, or by chosen node.

Here is an example devicetree which uses both:

```
/dts-v1/;

/ {
    chosen {
            zephyr,console = &uart0;
    };

    aliases {
            my-uart = &uart0;
    };

    soc {
            uart0: serial@12340000 {
                    ...
            };
    };
};
```

The `/aliases` and `/chosen` nodes do not refer to an actual hardware device. Their purpose is to specify other nodes in the devicetree.

Above, `my-uart` is an alias for the node with path `/soc/serial@12340000`. Using its node label `uart0`, the same node is set as the value of the chosen `zephyr,console` node.

Zephyr sample applications sometimes use aliases to allow overriding the particular hardware device used by the application in a generic way. For example, blinky uses this to abstract the LED to blink via the `led0` alias.

The `/chosen` node's properties are used to configure system- or subsystem-wide values. See *Chosen nodes* for more information.

Input and output files This section describes the input and output files shown in the figure in *Scope and purpose* in more detail.

Input files There are four types of devicetree input files:

- sources (`.dts`)
- includes (`.dtsi`)
- overlays (`.overlay`)
- bindings (`.yaml`)

The devicetree files inside the `zephyr` directory look like this:

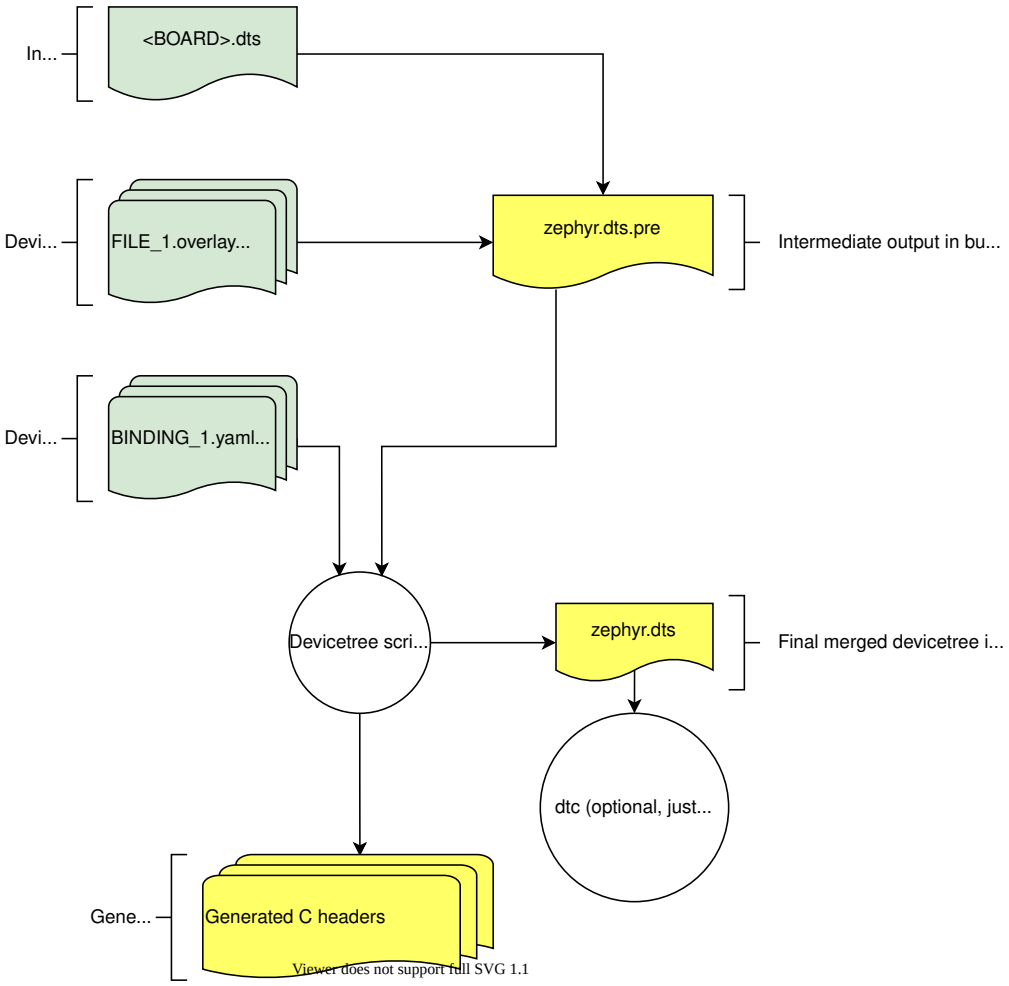


Fig. 3: Devicetree input (green) and output (yellow) files

```
boards/<ARCH>/<BOARD>/<BOARD>.dts
dts/common/skeleton.dtsi
dts/<ARCH>/.../<SOC>.dtsi
dts/bindings/.../binding.yaml
```

Generally speaking, every supported board has a `BOARD.dts` file describing its hardware. For example, the `reel_board` has boards/phytec/reel_board/reel_board.dts.

`BOARD.dts` includes one or more `.dtsi` files. These `.dtsi` files describe the CPU or system-on-chip Zephyr runs on, perhaps by including other `.dtsi` files. They can also describe other common hardware features shared by multiple boards. In addition to these includes, `BOARD.dts` also describes the board's specific hardware.

The `dts/common` directory contains `skeleton.dtsi`, a minimal include file for defining a complete devicetree. Architecture-specific subdirectories (`dts/<ARCH>`) contain `.dtsi` files for CPUs or SoCs which extend `skeleton.dtsi`.

The C preprocessor is run on all devicetree files to expand macro references, and includes are generally done with `#include <filename>` directives, even though DTS has a `/include/ "<filename>"` syntax.

`BOARD.dts` can be extended or modified using *overlays*. Overlays are also DTS files; the `.overlay` extension is just a convention which makes their purpose clear. Overlays adapt the base devicetree for different purposes:

- Zephyr applications can use overlays to enable a peripheral that is disabled by default, select a sensor on the board for an application specific purpose, etc. Along with *Configuration System (Kconfig)*, this makes it possible to reconfigure the kernel and device drivers without modifying source code.
- Overlays are also used when defining *Shields*.

The build system automatically picks up `.overlay` files stored in certain locations. It is also possible to explicitly list the overlays to include, via the `DTC_OVERLAY_FILE` CMake variable. See *Set devicetree overlays* for details.

The build system combines `BOARD.dts` and any `.overlay` files by concatenating them, with the overlays put last. This relies on DTS syntax which allows merging overlapping definitions of nodes in the devicetree. See *Example: FRDM-K64F and Hexiwear K64* for an example of how this works (in the context of `.dtsi` files, but the principle is the same for overlays). Putting the contents of the `.overlay` files last allows them to override `BOARD.dts`.

Devicetree bindings (which are YAML files) are essentially glue. They describe the contents of devicetree sources, includes, and overlays in a way that allows the build system to generate C macros usable by device drivers and applications. The `dts/bindings` directory contains bindings.

Scripts and tools The following libraries and scripts, located in scripts/dts/, create output files from input files. Their sources have extensive documentation.

dtlib.py
: A low-level DTS parsing library.

edtlib.py
: A library layered on top of dtlib that uses bindings to interpret properties and give a higher-level view of the devicetree. Uses dtlib to do the DTS parsing.

gen_defines.py
: A script that uses edtlib to generate C preprocessor macros from the devicetree and bindings.

In addition to these, the standard `dtc` (devicetree compiler) tool is run on the final devicetree if it is installed on your system. This is just to catch errors or warnings. The output is unused.

Boards may need to pass `dtc` additional flags, e.g. for warning suppression. Board directories can contain a file named `pre_dt_board.cmake` which configures these extra flags, like this:

```
list(APPEND EXTRA_DTC_FLAGS "-Wno-simple_bus_reg")
```

Output files These are created in your application's build directory.

Warning

Don't include the header files directly. *Devicetree access from C/C++* explains what to do instead.

`<build>/zephyr/zephyr.dts.pre`
: The preprocessed DTS source. This is an intermediate output file, which is input to `gen_defines.py` and used to create `zephyr.dts` and `devicetree_generated.h`.

`<build>/zephyr/include/generated/zephyr/devicetree_generated.h`
: The generated macros and additional comments describing the devicetree. Included by `devicetree.h`.

`<build>/zephyr/zephyr.dts`
: The final merged devicetree. This file is output by `gen_defines.py`. It is useful for debugging any issues. If the devicetree compiler `dtc` is installed, it is also run on this file, to catch any additional warnings or errors.

Design goals

Zephyr's use of devicetree has evolved significantly over time, and further changes are expected. The following are the general design goals, along with specific examples about how they impact Zephyr's source code, and areas where more work remains to be done.

Single source for hardware information Zephyr's built-in device drivers and sample applications shall obtain configurable hardware descriptions from devicetree.

Examples

- New device drivers shall use devicetree APIs to determine which *devices to create*.
- In-tree sample applications shall use *aliases* to determine which of multiple possible generic devices of a given type will be used in the current build. For example, the blinky sample uses this to determine the LED to blink.
- Boot-time pin muxing and pin control for new SoCs shall be accomplished via a devicetree-based pinctrl driver

Example remaining work

- Zephyr's *Test Runner (Twister)* currently use `board.yaml` files to determine the hardware supported by a board. This should be obtained from devicetree instead.
- Legacy device drivers currently use Kconfig to determine which instances of a particular compatible are enabled. This can and should be done with devicetree overlays instead.
- Board-level documentation still contains tables of hardware support which are generated and maintained by hand. This can and should be obtained from the board level devicetree instead.

- Runtime determination of `struct device` relationships should be done using information obtained from devicetree, e.g. for device power management.

Source compatibility with other operating systems Zephyr's devicetree tooling is based on a generic layer which is interoperable with other devicetree users, such as the Linux kernel.

Zephyr's binding language *semantics* can support Zephyr-specific attributes, but shall not express Zephyr-specific relationships.

Examples

- Zephyr's devicetree source parser, *dtlib.py*, is source-compatible with other tools like dtc in both directions: `dtlib.py` can parse `dtc` output, and `dtc` can parse `dtlib.py` output.
- Zephyr's "extended dtlib" library, `edtlib.py`, shall not include Zephyr-specific features. Its purpose is to provide a higher-level view of the devicetree for common elements like interrupts and buses.

  Only the high-level `gen_defines.py` script, which is built on top of `edtlib.py`, contains Zephyr-specific knowledge and features.

Example remaining work

- Zephyr has a custom *Devicetree bindings* language *syntax*. While Linux's dtschema does not yet meet Zephyr's needs, we should try to follow what it is capable of representing in Zephyr's own bindings.
- Due to inflexibility in the bindings language, Zephyr cannot support the full set of bindings supported by Linux.
- Devicetree source sharing between Zephyr and Linux is not done.

Devicetree bindings

A devicetree on its own is only half the story for describing hardware, as it is a relatively unstructured format. *Devicetree bindings* provide the other half.

A devicetree binding declares requirements on the contents of nodes, and provides semantic information about the contents of valid nodes. Zephyr devicetree bindings are YAML files in a custom format (Zephyr does not use the dt-schema tools used by the Linux kernel).

These pages introduce bindings, describe what they do, note where they are found, and explain their data format.

Note

See the *Bindings index* for reference information on bindings built in to Zephyr.

Note

For a detailed syntax reference, see *Devicetree bindings syntax*.

Introduction to Devicetree Bindings Devicetree nodes are matched to bindings using their *compatible properties*.

During the *Configuration Phase*, the build system tries to match each node in the devicetree to a binding file. When this succeeds, the build system uses the information in the binding file both when validating the node's contents and when generating macros for the node.

A simple example Here is an example devicetree node:

```
/* Node in a DTS file */
bar-device {
     compatible = "foo-company,bar-device";
     num-foos = <3>;
};
```

Here is a minimal binding file which matches the node:

```
# A YAML binding matching the node

compatible: "foo-company,bar-device"

properties:
  num-foos:
    type: int
    required: true
```

The build system matches the `bar-device` node to its YAML binding because the node's `compatible` property matches the binding's `compatible:` line.

What the build system does with bindings The build system uses bindings both to validate devicetree nodes and to convert the devicetree's contents into the generated *devicetree_generated.h* header file.

For example, the build system would use the above binding to check that the required `num-foos` property is present in the `bar-device` node, and that its value, `<3>`, has the correct type.

The build system will then generate a macro for the `bar-device` node's `num-foos` property, which will expand to the integer literal `3`. This macro lets you get the value of the property in C code using the API which is discussed later in this guide in *Devicetree access from C/C++*.

For another example, the following node would cause a build error, because it has no `num-foos` property, and this property is marked required in the binding:

```
bad-node {
     compatible = "foo-company,bar-device";
};
```

Other ways nodes are matched to bindings If a node has more than one string in its `compatible` property, the build system looks for compatible bindings in the listed order and uses the first match.

Take this node as an example:

```
baz-device {
     compatible = "foo-company,baz-device", "generic-baz-device";
};
```

The `baz-device` node would get matched to a binding with a `compatible: "generic-baz-device"` line if the build system can't find a binding with a `compatible: "foo-company,baz-device"` line.

Nodes without compatible properties can be matched to bindings associated with their parent nodes. These are called "child bindings". If a node describes hardware on a bus, like I2C or SPI,

then the bus type is also taken into account when matching nodes to bindings. (See *On-bus* for details).

See *The /zephyr,user node* for information about a special node that doesn't require any binding.

Where bindings are located Binding file names usually match their `compatible:` lines. For example, the above example binding would be named `foo-company,bar-device.yaml` by convention.

The build system looks for bindings in `dts/bindings` subdirectories of the following places:

- the zephyr repository
- your *application source directory*
- your *board directory*
- any *shield directories*
- any directories manually included in the *DTS_ROOT* CMake variable
- any *module* that defines a `dts_root` in its *Build settings*

The build system will consider any YAML file in any of these, including in any subdirectories, when matching nodes to bindings. A file is considered YAML if its name ends with `.yaml` or `.yml`.

Warning

The binding files must be located somewhere inside the `dts/bindings` subdirectory of the above places.

For example, if `my-app` is your application directory, then you must place application-specific bindings inside `my-app/dts/bindings`. So `my-app/dts/bindings/serial/my-company,my-serial-port.yaml` would be found, but `my-app/my-company,my-serial-port.yaml` would be ignored.

Devicetree bindings syntax This page documents the syntax of Zephyr's bindings format. Zephyr bindings files are YAML files. A *simple example* was given in the introduction page.

Contents

- *Top level keys*
- *Description*
- *Compatible*
- *Properties*
  - *Property entry syntax*
  - *Example property definitions*
  - *required*
  - *type*
  - *deprecated*
  - *default*
  - *enum*

      - *const*
      - *specifier-space*
  - *Child-binding*
  - *Bus*
  - *On-bus*
  - *Specifier cell names (*-cells)*
  - *Include*
  - *Nexus nodes and maps*

Top level keys The top level of a bindings file maps keys to values. The top-level keys look like this:

```
# A high level description of the device the binding applies to:
description: |
   This is the Vendomatic company's foo-device.

   Descriptions which span multiple lines (like this) are OK,
   and are encouraged for complex bindings.

   See https://yaml-multiline.info/ for formatting help.

# You can include definitions from other bindings using this syntax:
include: other.yaml

# Used to match nodes to this binding:
compatible: "manufacturer,foo-device"

properties:
  # Requirements for and descriptions of the properties that this
  # binding's nodes need to satisfy go here.

child-binding:
  # You can constrain the children of the nodes matching this binding
  # using this key.

# If the node describes bus hardware, like an SPI bus controller
# on an SoC, use 'bus:' to say which one, like this:
bus: spi

# If the node instead appears as a device on a bus, like an external
# SPI memory chip, use 'on-bus:' to say what type of bus, like this.
# Like 'compatible', this key also influences the way nodes match
# bindings.
on-bus: spi

foo-cells:
  # "Specifier" cell names for the 'foo' domain go here; example 'foo'
  # values are 'gpio', 'pwm', and 'dma'. See below for more information.
```

These keys are explained in the following sections.

Description A free-form description of node hardware goes here. You can put links to datasheets or example nodes or properties as well.

Compatible This key is used to match nodes to this binding as described in *Introduction to Devicetree Bindings*. It should look like this in a binding file:

```
# Note the comma-separated vendor prefix and device name
compatible: "manufacturer,device"
```

This devicetree node would match the above binding:

```
device {
     compatible = "manufacturer,device";
};
```

Assuming no binding has `compatible: "manufacturer,device-v2"`, it would also match this node:

```
device-2 {
    compatible = "manufacturer,device-v2", "manufacturer,device";
};
```

Each node's `compatible` property is tried in order. The first matching binding is used. The *on-bus:* key can be used to refine the search.

If more than one binding for a compatible is found, an error is raised.

The `manufacturer` prefix identifies the device vendor. See dts/bindings/vendor-prefixes.txt for a list of accepted vendor prefixes. The `device` part is usually from the datasheet.

Some bindings apply to a generic class of devices which do not have a specific vendor. In these cases, there is no vendor prefix. One example is the `gpio-leds` compatible which is commonly used to describe board LEDs connected to GPIOs.

Properties The `properties:` key describes properties that nodes which match the binding contain. For example, a binding for a UART peripheral might look something like this:

```
compatible: "manufacturer,serial"

properties:
  reg:
    type: array
    description: UART peripheral MMIO register space
    required: true
  current-speed:
    type: int
    description: current baud rate
    required: true
```

In this example, a node with compatible `"manufacturer,serial"` must contain a node named `current-speed`. The property's value must be a single integer. Similarly, the node must contain a `reg` property.

The build system uses bindings to generate C macros for devicetree properties that appear in DTS files. You can read more about how to get property values in source code from these macros in *Devicetree access from C/C++*. Generally speaking, the build system only generates macros for properties listed in the `properties:` key for the matching binding. Properties not mentioned in the binding are generally ignored by the build system.

The one exception is that the build system will always generate macros for standard properties, like *reg*, whose meaning is defined by the devicetree specification. This happens regardless of whether the node has a matching binding or not.

Property entry syntax Property entries in `properties:` are written in this syntax:

```
<property name>:
  required: <true | false>
  type: <string | int | boolean | array | uint8-array | string-array |
         phandle | phandles | phandle-array | path | compound>
  deprecated: <true | false>
  default: <default>
  description: <description of the property>
  enum:
    - <item1>
    - <item2>
    ...
    - <itemN>
  const: <string | int | array | uint8-array | string-array>
  specifier-space: <space-name>
```

Example property definitions Here are some more examples.

```
properties:
    # Describes a property like 'current-speed = <115200>;'. We pretend that
    # it's obligatory for the example node and set 'required: true'.
    current-speed:
        type: int
        required: true
        description: Initial baud rate for bar-device

    # Describes an optional property like 'keys = "foo", "bar";'
    keys:
        type: string-array
        description: Keys for bar-device

    # Describes an optional property like 'maximum-speed = "full-speed";'
    # the enum specifies known values that the string property may take
    maximum-speed:
        type: string
        description: Configures USB controllers to work up to a specific speed.
        enum:
           - "low-speed"
           - "full-speed"
           - "high-speed"
           - "super-speed"

    # Describes an optional property like 'resolution = <16>;'
    # the enum specifies known values that the int property may take
    resolution:
      type: int
      enum:
       - 8
       - 16
       - 24
       - 32

    # Describes a required property '#address-cells = <1>';  the const
    # specifies that the value for the property is expected to be the value 1
    "#address-cells":
        type: int
        required: true
        const: 1

    int-with-default:
        type: int
```

(continues on next page)

(continued from previous page)

```
    default: 123
    description: Value for int register, default is power-up configuration.

  array-with-default:
    type: array
    default: [1, 2, 3] # Same as 'array-with-default = <1 2 3>'

  string-with-default:
    type: string
    default: "foo"

  string-array-with-default:
    type: string-array
    default: ["foo", "bar"] # Same as 'string-array-with-default = "foo", "bar"'

  uint8-array-with-default:
    type: uint8-array
    default: [0x12, 0x34] # Same as 'uint8-array-with-default = [12 34]'
```

required Adding `required: true` to a property definition will fail the build if a node matches the binding, but does not contain the property.

The default setting is `required: false`; that is, properties are optional by default. Using `required: false` is therefore redundant and strongly discouraged.

type The type of a property constrains its values. The following types are available. See *Writing property values* for more details about writing values of each type in a DTS file. See *Phandles* for more information about the `phandle*` type properties.

Type	Description	Example in DTS
`string`	exactly one string	`status = "disabled";`
`int`	exactly one 32-bit value (cell)	`current-speed = <115200>;`
`boolean`	flags that don't take a value when true, and are absent if false	`hw-flow-control;`
`array`	zero or more 32-bit values (cells)	`offsets = <0x100 0x200 0x300>;`
`uint8-array`	zero or more bytes, in hex ('bytestring' in the Devicetree specification)	`local-mac-address = [de ad be ef 12 34];`
`string-array`	zero or more strings	`dma-names = "tx", "rx";`
`phandle`	exactly one phandle	`interrupt-parent = <&gic>;`
`phandles`	zero or more phandles	`pinctrl-0 = <&usart2_tx_pd5 &usart2_rx_pd6>;`
`phandle-array`	a list of phandles and 32-bit cells (usually specifiers)	`dmas = <&dma0 2>, <&dma0 3>;`
`path`	a path to a node as a phandle path reference or path string	`zephyr,bt-c2h-uart = &uart0;` or `foo = "/path/to/some/node";`
`compound`	a catch-all for more complex types (no macros will be generated)	`foo = <&label>, [01 02];`

deprecated A property with `deprecated: true` indicates to both the user and the tooling that the property is meant to be phased out.

The tooling will report a warning if the devicetree includes the property that is flagged as deprecated. (This warning is upgraded to an error in the *Test Runner (Twister)* for upstream pull requests.)

The default setting is `deprecated: false`. Using `deprecated: false` is therefore redundant and strongly discouraged.

default The optional `default:` setting gives a value that will be used if the property is missing from the devicetree node.

For example, with this binding fragment:

```
properties:
  foo:
    type: int
    default: 3
```

If property `foo` is missing in a matching node, then the output will be as if `foo = <3>;` had appeared in the DTS (except YAML data types are used for the default value).

Note that combining `default:` with `required: true` will raise an error.

For rules related to `default` in upstream Zephyr bindings, see *Rules for default values*.

See *Example property definitions* for examples. Putting `default:` on any property type besides those used in *Example property definitions* will raise an error.

enum The `enum:` line is followed by a list of values the property may contain. If a property value in DTS is not in the `enum:` list in the binding, an error is raised. See *Example property definitions* for examples.

const This specifies a constant value the property must take. It is mainly useful for constraining the values of common properties for a particular piece of hardware.

Warning

It is an abuse of this feature to use it to name properties in unconventional ways.

For example, this feature is not meant for cases like naming a property `my-pin`, then assigning it to the "gpio" specifier space using this feature. Properties which refer to GPIOs should use conventional names, i.e. end in `-gpios` or `-gpio`.

specifier-space This property, if present, manually sets the specifier space associated with a property with type `phandle-array`.

Normally, the specifier space is encoded implicitly in the property name. A property named `foos` with type `phandle-array` implicitly has specifier space `foo`. As a special case, `*-gpios` properties have specifier space "gpio", so that `foo-gpios` will have specifier space "gpio" rather than "foo-gpio".

You can use `specifier-space` to manually provide a space if using this convention would result in an awkward or unconventional name.

For example:

```
compatible: ...
properties:
  bar:
    type: phandle-array
    specifier-space: my-custom-space
```

Above, the bar property's specifier space is set to "my-custom-space".

You could then use the property in a devicetree like this:

```
controller1: custom-controller@1000 {
        #my-custom-space-cells = <2>;
};

controller2: custom-controller@2000 {
        #my-custom-space-cells = <1>;
};

my-node {
        bar = <&controller1 10 20>, <&controller2 30>;
};
```

Generally speaking, you should reserve this feature for cases where the implicit specifier space naming convention doesn't work. One appropriate example is an mboxes property with specifier space "mbox", not "mboxe". You can write this property as follows:

```
properties:
  mboxes:
    type: phandle-array
    specifier-space: mbox
```

Child-binding child-binding can be used when a node has children that all share the same properties. Each child gets the contents of child-binding as its binding, though an explicit compatible = ... on the child node takes precedence, if a binding is found for it.

Consider a binding for a PWM LED node like this one, where the child nodes are required to have a pwms property:

```
pwmleds {
        compatible = "pwm-leds";

        red_pwm_led {
                pwms = <&pwm3 4 15625000>;
        };
        green_pwm_led {
                pwms = <&pwm3 0 15625000>;
        };
        /* ... */
};
```

The binding would look like this:

```
compatible: "pwm-leds"

child-binding:
  description: LED that uses PWM

  properties:
    pwms:
      type: phandle-array
      required: true
```

child-binding also works recursively. For example, this binding:

```
compatible: foo

child-binding:
  child-binding:
    properties:
      my-property:
        type: int
        required: true
```

will apply to the grandchild node in this DTS:

```
parent {
        compatible = "foo";
        child {
                grandchild {
                        my-property = <123>;
                };
        };
};
```

Bus If the node is a bus controller, use bus: in the binding to say what type of bus. For example, a binding for a SPI peripheral on an SoC would look like this:

```
compatible: "manufacturer,spi-peripheral"
bus: spi
# ...
```

The presence of this key in the binding informs the build system that the children of any node matching this binding appear on this type of bus.

This in turn influences the way on-bus: is used to match bindings for the child nodes.

For a single bus supporting multiple protocols, e.g. I3C and I2C, the bus: in the binding can have a list as value:

```
compatible: "manufacturer,i3c-controller"
bus: [i3c, i2c]
# ...
```

On-bus If the node appears as a device on a bus, use on-bus: in the binding to say what type of bus.

For example, a binding for an external SPI memory chip should include this line:

```
on-bus: spi
```

And a binding for an I2C based temperature sensor should include this line:

```
on-bus: i2c
```

When looking for a binding for a node, the build system checks if the binding for the parent node contains bus: <bus type>. If it does, then only bindings with a matching on-bus: <bus type> and bindings without an explicit on-bus are considered. Bindings with an explicit on-bus: <bus type> are searched for first, before bindings without an explicit on-bus. The search repeats for each item in the node's compatible property, in order.

This feature allows the same device to have different bindings depending on what bus it appears on. For example, consider a sensor device with compatible manufacturer,sensor which can be used via either I2C or SPI.

The sensor node may therefore appear in the devicetree as a child node of either an SPI or an I2C controller, like this:

```
spi-bus@0 {
   /* ... some compatible with 'bus: spi', etc. ... */

   sensor@0 {
       compatible = "manufacturer,sensor";
       reg = <0>;
       /* ... */
   };
};

i2c-bus@0 {
   /* ... some compatible with 'bus: i2c', etc. ... */

   sensor@79 {
       compatible = "manufacturer,sensor";
       reg = <79>;
       /* ... */
   };
};
```

You can write two separate binding files which match these individual sensor nodes, even though they have the same compatible:

```
# manufacturer,sensor-spi.yaml, which matches sensor@0 on the SPI bus:
compatible: "manufacturer,sensor"
on-bus: spi

# manufacturer,sensor-i2c.yaml, which matches sensor@79 on the I2C bus:
compatible: "manufacturer,sensor"
properties:
  uses-clock-stretching:
    type: boolean
on-bus: i2c
```

Only `sensor@79` can have a `use-clock-stretching` property. The bus-sensitive logic ignores `manufacturer,sensor-i2c.yaml` when searching for a binding for `sensor@0`.

Specifier cell names (*-cells) This section documents how to name the cells in a specifier within a binding. These concepts are discussed in detail later in this guide in *phandle-array properties*.

Consider a binding for a node whose phandle may appear in a `phandle-array` property, like the PWM controllers `pwm1` and `pwm2` in this example:

```
pwm1: pwm@deadbeef {
    compatible = "foo,pwm";
    #pwm-cells = <2>;
};

pwm2: pwm@deadbeef {
    compatible = "bar,pwm";
    #pwm-cells = <1>;
};

my-node {
    pwms = <&pwm1 1 2000>, <&pwm2 3000>;
};
```

The bindings for compatible "foo,pwm" and "bar,pwm" must give a name to the cells that appear in a PWM specifier using pwm-cells:, like this:

```
# foo,pwm.yaml
compatible: "foo,pwm"
...
pwm-cells:
  - channel
  - period

# bar,pwm.yaml
compatible: "bar,pwm"
...
pwm-cells:
  - period
```

A *-names (e.g. pwm-names) property can appear on the node as well, giving a name to each entry.

This allows the cells in the specifiers to be accessed by name, e.g. using APIs like *DT_PWMS_CHANNEL_BY_NAME*.

If the specifier is empty (e.g. #clock-cells = <0>), then *-cells can either be omitted (recommended) or set to an empty array. Note that an empty array is specified as e.g. clock-cells: [] in YAML.

Include Bindings can include other files, which can be used to share common property definitions between bindings. Use the include: key for this. Its value is either a string or a list.

In the simplest case, you can include another file by giving its name as a string, like this:

```
include: foo.yaml
```

If any file named foo.yaml is found (see *Where bindings are located* for the search process), it will be included into this binding.

Included files are merged into bindings with a simple recursive dictionary merge. The build system will check that the resulting merged binding is well-formed. It is allowed to include at any level, including child-binding, like this:

```
# foo.yaml will be merged with content at this level
include: foo.yaml

child-binding:
  # bar.yaml will be merged with content at this level
  include: bar.yaml
```

It is an error if a key appears with a different value in a binding and in a file it includes, with one exception: a binding can have required: true for a *property definition* for which the included file has required: false. The required: true takes precedence, allowing bindings to strengthen requirements from included files.

Note that weakening requirements by having required: false where the included file has required: true is an error. This is meant to keep the organization clean.

The file base.yaml contains definitions for many common properties. When writing a new binding, it is a good idea to check if base.yaml already defines some of the needed properties, and include it if it does.

Note that you can make a property defined in base.yaml obligatory like this, taking *reg* as an example:

```
reg:
  required: true
```

This relies on the dictionary merge to fill in the other keys for `reg`, like `type`.

To include multiple files, you can use a list of strings:

```
include:
  - foo.yaml
  - bar.yaml
```

This includes the files `foo.yaml` and `bar.yaml`. (You can write this list in a single line of YAML as `include: [foo.yaml, bar.yaml]`.)

When including multiple files, any overlapping `required` keys on properties in the included files are ORed together. This makes sure that a `required: true` is always respected.

In some cases, you may want to include some property definitions from a file, but not all of them. In this case, `include:` should be a list, and you can filter out just the definitions you want by putting a mapping in the list, like this:

```
include:
  - name: foo.yaml
    property-allowlist:
      - i-want-this-one
      - and-this-one
  - name: bar.yaml
    property-blocklist:
      - do-not-include-this-one
      - or-this-one
```

Each map element must have a `name` key which is the filename to include, and may have `property-allowlist` and `property-blocklist` keys that filter which properties are included.

You cannot have a single map element with both `property-allowlist` and `property-blocklist` keys. A map element with neither `property-allowlist` nor `property-blocklist` is valid; no additional filtering is done.

You can freely intermix strings and mappings in a single `include:` list:

```
include:
  - foo.yaml
  - name: bar.yaml
    property-blocklist:
      - do-not-include-this-one
      - or-this-one
```

Finally, you can filter from a child binding like this:

```
include:
  - name: bar.yaml
    child-binding:
      property-allowlist:
        - child-prop-to-allow
```

Nexus nodes and maps All `phandle-array` type properties support mapping through `*-map` properties, e.g. `gpio-map`, as defined by the Devicetree specification.

This is used, for example, to define connector nodes for common breakout headers, such as the `arduino_header` nodes that are conventionally defined in the devicetrees for boards with Arduino compatible expansion headers.

Rules for upstream bindings This section includes general rules for writing bindings that you want to submit to the upstream Zephyr Project. (You don't need to follow these rules for bindings you don't intend to contribute to the Zephyr Project, but it's a good idea.)

Decisions made by the Zephyr devicetree maintainer override the contents of this section. If that happens, though, please let them know so they can update this page, or you can send a patch yourself.

Contents

- *Always check for existing bindings*
- *General rules*
  - *File names*
  - *Recommendations are requirements*
  - *Descriptions*
  - *Naming conventions*
- *Rules for vendor prefixes*
- *Rules for default values*
- *The `zephyr,` prefix*

Always check for existing bindings Zephyr aims for devicetree *Source compatibility with other operating systems*. Therefore, if there is an existing binding for your device in an authoritative location, you should try to replicate its properties when writing a Zephyr binding, and you must justify any Zephyr-specific divergences.

In particular, this rule applies if:

- There is an existing binding in the mainline Linux kernel. See `Documentation/devicetree/bindings` in Linus's tree for existing bindings and the Linux devicetree documentation for more information.
- Your hardware vendor provides an official binding outside of the Linux kernel.

General rules

File names Bindings which match a compatible must have file names based on the compatible.

- For example, a binding for compatible `vnd,foo` must be named `vnd,foo.yaml`.
- If the binding is bus-specific, you can append the bus to the file name; for example, if the binding YAML has `on-bus: bar`, you may name the file `vnd,foo-bar.yaml`.

Recommendations are requirements All recommendations in *default* are requirements when submitting the binding.

In particular, if you use the `default:` feature, you must justify the value in the property's description.

Descriptions There are only two acceptable ways to write property `description:` strings.

If your description is short, it's fine to use this style:

```
description: my short string
```

If your description is long or spans multiple lines, you must use this style:

```
description: |
  My very long string
  goes here.
  Look at all these lines!
```

This | style prevents YAML parsers from removing the newlines in multi-line descriptions. This in turn makes these long strings display properly in the *Bindings index*.

Naming conventions Do not use uppercase letters (`A` through `Z`) or underscores (`_`) in property names. Use lowercase letters (`a` through `z`) instead of uppercase. Use dashes (`-`) instead of underscores. (The one exception to this rule is if you are replicating a well-established binding from somewhere like Linux.)

Rules for vendor prefixes The following general rules apply to vendor prefixes in *compatible* properties.

- If your device is manufactured by a specific vendor, then its compatible should have a vendor prefix.

  If your binding describes hardware with a well known vendor from the list in dts/bindings/vendor-prefixes.txt, you must use that vendor prefix.

- If your device is not manufactured by a specific hardware vendor, do **not** invent a vendor prefix. Vendor prefixes are not mandatory parts of compatible properties, and compatibles should not include them unless they refer to an actual vendor. There are some exceptions to this rule, but the practice is strongly discouraged.

- Do not submit additions to Zephyr's `dts/bindings/vendor-prefixes.txt` file unless you also include users of the new prefix. This means at least a binding and a devicetree using the vendor prefix, and should ideally include a device driver handling that compatible.

  For custom bindings, you can add a custom `dts/bindings/vendor-prefixes.txt` file to any directory in your *DTS_ROOT*. The devicetree tooling will respect these prefixes, and will not generate warnings or errors if you use them in your own bindings or devicetrees.

- We sometimes synchronize Zephyr's vendor-prefixes.txt file with the Linux kernel's equivalent file; this process is exempt from the previous rule.

- If your binding is describing an abstract class of hardware with Zephyr specific drivers handling the nodes, it's usually best to use `zephyr` as the vendor prefix. See *Zephyr-specific binding (zephyr)* for examples.

Rules for default values In any case where `default:` is used in a devicetree binding, the `description:` for that property **must** explain *why* the value was selected and any conditions that would make it necessary to provide a different value. Additionally, if changing one property would require changing another to create a consistent configuration, then those properties should be made required.

There is no need to document the default value itself; this is already present in the *Bindings index* output.

There is a risk in using `default:` when the value in the binding may be incorrect for a particular board or hardware configuration. For example, defaulting the capacity of the connected power cell in a charging IC binding is likely to be incorrect. For such properties it's better to make the property `required: true`, forcing the user to make an explicit choice.

Driver developers should use their best judgment as to whether a value can be safely defaulted. Candidates for default values include:

- delays that would be different only under unusual conditions (such as intervening hardware)

- configuration for devices that have a standard initial configuration (such as a USB audio headset)
- defaults which match the vendor-specified power-on reset value (as long as they are independent from other properties)

Examples of how to write descriptions according to these rules:

```
properties:
  cs-interval:
    type: int
    default: 0
    description: |
      Minimum interval between chip select deassertion and assertion.
      The default corresponds to the reset value of the register field.
  hold-time-ms:
    type: int
    default: 20
    description: |
      Amount of time to hold the power enable GPIO asserted before
      initiating communication. The default was recommended in the
      manufacturer datasheet, and would only change under very
      cold temperatures.
```

Some examples of what **not** to do, and why:

```
properties:
  # Description doesn't mention anything about the default
  foo:
    type: int
    default: 1
    description: number of foos

  # Description mentions the default value instead of why it
  # was chosen
  bar:
    type: int
    default: 2
    description: bar size; default is 2

  # Explanation of the default value is in a comment instead
  # of the description. This won't be shown in the bindings index.
  baz:
    type: int
    # This is the recommended value chosen by the manufacturer.
    default: 2
    description: baz time in milliseconds
```

The `zephyr,` prefix You must add this prefix to property names in the following cases:

- Zephyr-specific extensions to bindings we share with upstream Linux. One example is the `zephyr,vref-mv` ADC channel property which is common to ADC controllers defined in dts/bindings/adc/adc-controller.yaml. This channel binding is partially shared with an analogous Linux binding, and Zephyr-specific extensions are marked as such with the prefix.
- Configuration values that are specific to a Zephyr device driver. One example is the `zephyr,lazy-load` property in the `ti,bq274xx` binding. Though devicetree in general is a hardware description and configuration language, it is Zephyr's only mechanism for configuring driver behavior for an individual `struct device`. Therefore, as a compromise, we do allow some software configuration in Zephyr's devicetree bindings, as long as they use this prefix to show that they are Zephyr specific.

You may use the `zephyr,` prefix when naming a devicetree compatible that is specific to Zephyr. One example is `zephyr,ipc-openamp-static-vrings`. In this case, it's permitted but not required to add the `zephyr,` prefix to properties defined in the binding.

Devicetree access from C/C++

This guide describes Zephyr's `<zephyr/devicetree.h>` API for reading the devicetree from C source files. It assumes you're familiar with the concepts in *Introduction to devicetree* and *Devicetree bindings*. See *Devicetree Reference* for reference material.

A note for Linux developers Linux developers familiar with devicetree should be warned that the API described here differs significantly from how devicetree is used on Linux.

Instead of generating a C header with all the devicetree data which is then abstracted behind a macro API, the Linux kernel would instead read the devicetree data structure in its binary form. The binary representation is parsed at runtime, for example to load and initialize device drivers.

Zephyr does not work this way because the size of the devicetree binary and associated handling code would be too large to fit comfortably on the relatively constrained devices Zephyr supports.

Node identifiers To get information about a particular devicetree node, you need a *node identifier* for it. This is a just a C macro that refers to the node.

These are the main ways to get a node identifier:

By path
: Use *DT_PATH()* along with the node's full path in the devicetree, starting from the root node. This is mostly useful if you happen to know the exact node you're looking for.

By node label
: Use *DT_NODELABEL()* to get a node identifier from a *node label*. Node labels are often provided by SoC `.dtsi` files to give nodes names that match the SoC datasheet, like `i2c1`, `spi2`, etc.

By alias
: Use *DT_ALIAS()* to get a node identifier for a property of the special `/aliases` node. This is sometimes done by applications (like blinky, which uses the `led0` alias) that need to refer to *some* device of a particular type ("the board's user LED") but don't care which one is used.

By instance number
: This is done primarily by device drivers, as instance numbers are a way to refer to individual nodes based on a matching compatible. Get these with *DT_INST()*, but be careful doing so. See below.

By chosen node
: Use *DT_CHOSEN()* to get a node identifier for `/chosen` node properties.

By parent/child
: Use *DT_PARENT()* and *DT_CHILD()* to get a node identifier for a parent or child node, starting from a node identifier you already have.

Two node identifiers which refer to the same node are identical and can be used interchangeably.

Here's a DTS fragment for some imaginary hardware we'll return to throughout this file for examples:

```
/dts-v1/;

/ {
```

(continues on next page)

(continued from previous page)

```
        aliases {
                sensor-controller = &i2c1;
        };

        soc {
                i2c1: i2c@40002000 {
                        compatible = "vnd,soc-i2c";
                        label = "I2C_1";
                        reg = <0x40002000 0x1000>;
                        status = "okay";
                        clock-frequency = < 100000 >;
                };
        };
};
```

Here are a few ways to get node identifiers for the `i2c@40002000` node:

- `DT_PATH(soc, i2c_40002000)`
- `DT_NODELABEL(i2c1)`
- `DT_ALIAS(sensor_controller)`
- `DT_INST(x, vnd_soc_i2c)` for some unknown number `x`. See the *DT_INST()* documentation for details.

Important

Non-alphanumeric characters like dash (`-`) and the at sign (`@`) in devicetree names are converted to underscores (`_`). The names in a DTS are also converted to lowercase.

Node identifiers are not values There is no way to store one in a variable. You cannot write:

```
/* These will give you compiler errors: */

void *i2c_0 = DT_INST(0, vnd_soc_i2c);
unsigned int i2c_1 = DT_INST(1, vnd_soc_i2c);
long my_i2c = DT_NODELABEL(i2c1);
```

If you want something short to save typing, use C macros:

```
/* Use something like this instead: */

#define MY_I2C DT_NODELABEL(i2c1)

#define INST(i) DT_INST(i, vnd_soc_i2c)
#define I2C_0 INST(0)
#define I2C_1 INST(1)
```

Property access The right API to use to read property values depends on the node and property.

- *Checking properties and values*
- *Simple properties*
- *reg properties*
- *interrupts properties*
- *phandle properties*

Checking properties and values You can use *DT_NODE_HAS_PROP()* to check if a node has a property. For the *example devicetree* above:

```
DT_NODE_HAS_PROP(DT_NODELABEL(i2c1), clock_frequency)  /* expands to 1 */
DT_NODE_HAS_PROP(DT_NODELABEL(i2c1), not_a_property)   /* expands to 0 */
```

Simple properties Use `DT_PROP(node_id, property)` to read basic integer, boolean, string, numeric array, and string array properties.

For example, to read the `clock-frequency` property's value in the *above example*:

```
DT_PROP(DT_PATH(soc, i2c_40002000), clock_frequency)  /* This is 100000, */
DT_PROP(DT_NODELABEL(i2c1), clock_frequency)          /* and so is this, */
DT_PROP(DT_ALIAS(sensor_controller), clock_frequency) /* and this. */
```

Important

The DTS property `clock-frequency` is spelled `clock_frequency` in C. That is, properties also need special characters converted to underscores. Their names are also forced to lowercase.

Properties with `string` and `boolean` types work the exact same way. The `DT_PROP()` macro expands to a string literal in the case of strings, and the number 0 or 1 in the case of booleans. For example:

```
#define I2C1 DT_NODELABEL(i2c1)

DT_PROP(I2C1, status)  /* expands to the string literal "okay" */
```

Note

Don't use DT_NODE_HAS_PROP() for boolean properties. Use DT_PROP() instead as shown above. It will expand to either 0 or 1 depending on if the property is present or absent.

Properties with type `array`, `uint8-array`, and `string-array` work similarly, except `DT_PROP()` expands to an array initializer in these cases. Here is an example devicetree fragment:

```
foo: foo@1234 {
        a = <1000 2000 3000>; /* array */
        b = [aa bb cc dd];    /* uint8-array */
        c = "bar", "baz";     /* string-array */
};
```

Its properties can be accessed like this:

```
#define FOO DT_NODELABEL(foo)

int a[] = DT_PROP(FOO, a);           /* {1000, 2000, 3000} */
unsigned char b[] = DT_PROP(FOO, b); /* {0xaa, 0xbb, 0xcc, 0xdd} */
char* c[] = DT_PROP(FOO, c);         /* {"foo", "bar"} */
```

You can use *DT_PROP_LEN()* to get logical array lengths in number of elements.

```
size_t a_len = DT_PROP_LEN(FOO, a); /* 3 */
size_t b_len = DT_PROP_LEN(FOO, b); /* 4 */
size_t c_len = DT_PROP_LEN(FOO, c); /* 2 */
```

`DT_PROP_LEN()` cannot be used with the special `reg` or `interrupts` properties. These have alternative macros which are described next.

reg properties See *Important properties* for an introduction to `reg`.

Given a node identifier `node_id`, `DT_NUM_REGS(node_id)` is the total number of register blocks in the node's `reg` property.

You **cannot** read register block addresses and lengths with `DT_PROP(node, reg)`. Instead, if a node only has one register block, use *DT_REG_ADDR()* or *DT_REG_SIZE()*:

- `DT_REG_ADDR(node_id)`: the given node's register block address
- `DT_REG_SIZE(node_id)`: its size

Use *DT_REG_ADDR_BY_IDX()* or *DT_REG_SIZE_BY_IDX()* instead if the node has multiple register blocks:

- `DT_REG_ADDR_BY_IDX(node_id, idx)`: address of register block at index `idx`
- `DT_REG_SIZE_BY_IDX(node_id, idx)`: size of block at index `idx`

The `idx` argument to these must be an integer literal or a macro that expands to one without requiring any arithmetic. In particular, `idx` cannot be a variable. This won't work:

```
/* This will cause a compiler error. */

for (size_t i = 0; i < DT_NUM_REGS(node_id); i++) {
        size_t addr = DT_REG_ADDR_BY_IDX(node_id, i);
}
```

interrupts properties See *Important properties* for a brief introduction to `interrupts`.

Given a node identifier `node_id`, `DT_NUM_IRQS(node_id)` is the total number of interrupt specifiers in the node's `interrupts` property.

The most general purpose API macro for accessing these is *DT_IRQ_BY_IDX()*:

```
DT_IRQ_BY_IDX(node_id, idx, val)
```

Here, `idx` is the logical index into the `interrupts` array, i.e. it is the index of an individual interrupt specifier in the property. The `val` argument is the name of a cell within the interrupt specifier. To use this macro, check the bindings file for the node you are interested in to find the `val` names.

Most Zephyr devicetree bindings have a cell named `irq`, which is the interrupt number. You can use *DT_IRQN()* as a convenient way to get a processed view of this value.

Warning

Here, "processed" reflects Zephyr's devicetree *Scripts and tools*, which change the `irq` number in *zephyr.dts* to handle hardware constraints on some SoCs and in accordance with Zephyr's multilevel interrupt numbering.

This is currently not very well documented, and you'll need to read the scripts' source code and existing drivers for more details if you are writing a device driver.

Note

See *Phandles* for a detailed guide to phandles.

phandle properties Property values can refer to other nodes using the `&another-node` phandle syntax introduced in *Writing property values*. Properties which contain phandles have type

phandle, phandles, or phandle-array in their bindings. We'll call these "phandle properties" for short.

You can convert a phandle to a node identifier using *DT_PHANDLE()*, *DT_PHANDLE_BY_IDX()*, or *DT_PHANDLE_BY_NAME()*, depending on the type of property you are working with.

One common use case for phandle properties is referring to other hardware in the tree. In this case, you usually want to convert the devicetree-level phandle to a Zephyr driver-level *struct device*. See *Get a struct device from a devicetree node* for ways to do that.

Another common use case is accessing specifier values in a phandle array. The general purpose APIs for this are *DT_PHA_BY_IDX()* and *DT_PHA()*. There are also hardware-specific shorthands like *DT_GPIO_CTLR_BY_IDX()*, *DT_GPIO_CTLR()*, *DT_GPIO_PIN_BY_IDX()*, *DT_GPIO_PIN()*, *DT_GPIO_FLAGS_BY_IDX()*, and *DT_GPIO_FLAGS()*.

See *DT_PHA_HAS_CELL_AT_IDX()* and *DT_PROP_HAS_IDX()* for ways to check if a specifier value is present in a phandle property.

Other APIs Here are pointers to some other available APIs.

- *DT_CHOSEN()*, *DT_HAS_CHOSEN()*: for properties of the special /chosen node
- *DT_HAS_COMPAT_STATUS_OKAY()*, *DT_NODE_HAS_COMPAT()*: global- and node-specific tests related to the compatible property
- *DT_BUS()*: get a node's bus controller, if there is one
- *DT_ENUM_IDX()*: for properties whose values are among a fixed list of choices
- *Fixed flash partitions*: APIs for managing fixed flash partitions. Also see *Flash map*, which wraps this in a more user-friendly API.

Device driver conveniences Special purpose macros are available for writing device drivers, which usually rely on *instance identifiers*.

To use these, you must define DT_DRV_COMPAT to the compat value your driver implements support for. This compat value is what you would pass to *DT_INST()*.

If you do that, you can access the properties of individual instances of your compatible with less typing, like this:

```
#include <zephyr/devicetree.h>

#define DT_DRV_COMPAT my_driver_compat

/* This is same thing as DT_INST(0, my_driver_compat): */
DT_DRV_INST(0)

/*
 * This is the same thing as
 * DT_PROP(DT_INST(0, my_driver_compat), clock_frequency)
 */
DT_INST_PROP(0, clock_frequency)
```

See *Instance-based APIs* for a generic API reference.

Hardware specific APIs Convenience macros built on top of the above APIs are also defined to help readability for hardware specific code. See *Hardware specific APIs* for details.

Generated macros While the `zephyr/devicetree.h` API is not generated, it does rely on a generated C header which is put into every application build directory: *devicetree_generated.h*. This file contains macros with devicetree data.

These macros have tricky naming conventions which the *Devicetree API* API abstracts away. They should be considered an implementation detail, but it's useful to understand them since they will frequently be seen in compiler error messages.

This section contains an Augmented Backus-Naur Form grammar for these generated macros, with examples and more details in comments. See RFC 7405 (which extends RFC 5234) for a syntax specification.

```
; An RFC 7405 ABNF grammar for devicetree macros.
;
; This does *not* cover macros pulled out of DT via Kconfig,
; like CONFIG_SRAM_BASE_ADDRESS, etc. It only describes the
; ones that start with DT_ and are directly generated.

; ------------------------------------------------------------------
; dt-macro: the top level nonterminal for a devicetree macro
;
; A dt-macro starts with uppercase "DT_", and is one of:
;
; - a <node-macro>, generated for a particular node
; - some <other-macro>, a catch-all for other types of macros
dt-macro = node-macro / other-macro

; ------------------------------------------------------------------
; node-macro: a macro related to a node

; A macro about a property value
node-macro =  property-macro
; A macro about the pinctrl properties in a node.
node-macro =/ pinctrl-macro
; A macro about the GPIO hog properties in a node.
node-macro =/ gpiohogs-macro
; EXISTS macro: node exists in the devicetree
node-macro =/ %s"DT_N" path-id %s"_EXISTS"
; Bus macros: the plain BUS is a way to access a node's bus controller.
; The additional dt-name suffix is added to match that node's bus type;
; the dt-name in this case is something like "spi" or "i2c".
node-macro =/ %s"DT_N" path-id %s"_BUS" ["_" dt-name]
; The reg property is special and has its own macros.
node-macro =/ %s"DT_N" path-id %s"_REG_NUM"
node-macro =/ %s"DT_N" path-id %s"_REG_IDX_" DIGIT "_EXISTS"
node-macro =/ %s"DT_N" path-id %s"_REG_IDX_" DIGIT
              %s"_VAL_" ( %s"ADDRESS" / %s"SIZE")
node-macro =/ %s"DT_N" path-id %s"_REG_NAME_" dt-name
              %s"_VAL_" ( %s"ADDRESS" / %s"SIZE")
node-macro =/ %s"DT_N" path-id %s"_REG_NAME_" dt-name "_EXISTS"
; The interrupts property is also special.
node-macro =/ %s"DT_N" path-id %s"_IRQ_NUM"
node-macro =/ %s"DT_N" path-id %s"_IRQ_LEVEL"
node-macro =/ %s"DT_N" path-id %s"_IRQ_IDX_" DIGIT "_EXISTS"
node-macro =/ %s"DT_N" path-id %s"_IRQ_IDX_" DIGIT
              %s"_VAL_" dt-name [ %s"_EXISTS" ]
node-macro =/ %s"DT_N" path-id %s"_CONTROLLER"
node-macro =/ %s"DT_N" path-id %s"_IRQ_NAME_" dt-name
              %s"_VAL_" dt-name [ %s"_EXISTS" ]
node-macro =/ %s"DT_N" path-id %s"_IRQ_NAME_" dt-name "_CONTROLLER"
; The ranges property is also special.
node-macro =/ %s"DT_N" path-id %s"_RANGES_NUM"
node-macro =/ %s"DT_N" path-id %s"_RANGES_IDX_" DIGIT "_EXISTS"
```

(continues on next page)

(continued from previous page)

```
node-macro =/ %s"DT_N" path-id %s"_RANGES_IDX_" DIGIT
              %s"_VAL_" ( %s"CHILD_BUS_FLAGS" / %s"CHILD_BUS_ADDRESS" /
                          %s"PARENT_BUS_ADDRESS" / %s"LENGTH")
node-macro =/ %s"DT_N" path-id %s"_RANGES_IDX_" DIGIT
              %s"_VAL_CHILD_BUS_FLAGS_EXISTS"
node-macro =/ %s"DT_N" path-id %s"_FOREACH_RANGE"
; Subnodes of the fixed-partitions compatible get macros which contain
; a unique ordinal value for each partition
node-macro =/ %s"DT_N" path-id %s"_PARTITION_ID" DIGIT
; Macros are generated for each of a node's compatibles;
; dt-name in this case is something like "vnd_device".
node-macro =/ %s"DT_N" path-id %s"_COMPAT_MATCHES_" dt-name
node-macro =/ %s"DT_N" path-id %s"_COMPAT_VENDOR_IDX_" DIGIT "_EXISTS"
node-macro =/ %s"DT_N" path-id %s"_COMPAT_VENDOR_IDX_" DIGIT
node-macro =/ %s"DT_N" path-id %s"_COMPAT_MODEL_IDX_" DIGIT "_EXISTS"
node-macro =/ %s"DT_N" path-id %s"_COMPAT_MODEL_IDX_" DIGIT
; Every non-root node gets one of these macros, which expands to the node
; identifier for that node's parent in the devicetree.
node-macro =/ %s"DT_N" path-id %s"_PARENT"
; These are used internally by DT_FOREACH_PROP_ELEM(_SEP)(_VARGS), which
; iterates over each property element.
node-macro =/ %s"DT_N" path-id %s"_P_" prop-id %s"_FOREACH_PROP_ELEM"
node-macro =/ %s"DT_N" path-id %s"_P_" prop-id %s"_FOREACH_PROP_ELEM_SEP"
node-macro =/ %s"DT_N" path-id %s"_P_" prop-id %s"_FOREACH_PROP_ELEM_VARGS"
node-macro =/ %s"DT_N" path-id %s"_P_" prop-id %s"_FOREACH_PROP_ELEM_SEP_VARGS"
; These are used by DT_CHILD_NUM and DT_CHILD_NUM_STATUS_OKAY macros
node-macro =/ %s"DT_N" path-id %s"_CHILD_NUM"
node-macro =/ %s"DT_N" path-id %s"_CHILD_NUM_STATUS_OKAY"
; These are used internally by DT_FOREACH_CHILD, which iterates over
; each child node.
node-macro =/ %s"DT_N" path-id %s"_FOREACH_CHILD"
node-macro =/ %s"DT_N" path-id %s"_FOREACH_CHILD_SEP"
node-macro =/ %s"DT_N" path-id %s"_FOREACH_CHILD_VARGS"
node-macro =/ %s"DT_N" path-id %s"_FOREACH_CHILD_SEP_VARGS"
; These are used internally by DT_FOREACH_CHILD_STATUS_OKAY, which iterates
; over each child node with status "okay".
node-macro =/ %s"DT_N" path-id %s"_FOREACH_CHILD_STATUS_OKAY"
node-macro =/ %s"DT_N" path-id %s"_FOREACH_CHILD_STATUS_OKAY_SEP"
node-macro =/ %s"DT_N" path-id %s"_FOREACH_CHILD_STATUS_OKAY_VARGS"
node-macro =/ %s"DT_N" path-id %s"_FOREACH_CHILD_STATUS_OKAY_SEP_VARGS"
; These are used internally by DT_FOREACH_NODELABEL and
; DT_FOREACH_NODELABEL_VARGS, which iterate over a node's node labels.
node-macro =/ %s"DT_N" path-id %s"_FOREACH_NODELABEL" [ %s"_VARGS" ]
; These are used internally by DT_NUM_NODELABELS
node-macro =/ %s"DT_N" path-id %s"_NODELABEL_NUM"
; The node's zero-based index in the list of it's parent's child nodes.
node-macro =/ %s"DT_N" path-id %s"_CHILD_IDX"
; The node's status macro; dt-name in this case is something like "okay"
; or "disabled".
node-macro =/ %s"DT_N" path-id %s"_STATUS_" dt-name
; The node's dependency ordinal. This is a non-negative integer
; value that is used to represent dependency information.
node-macro =/ %s"DT_N" path-id %s"_ORD"
; The node's path, as a string literal
node-macro =/ %s"DT_N" path-id %s"_PATH"
; The node's name@unit-addr, as a string literal
node-macro =/ %s"DT_N" path-id %s"_FULL_NAME"
; The dependency ordinals of a node's requirements (direct dependencies).
node-macro =/ %s"DT_N" path-id %s"_REQUIRES_ORDS"
; The dependency ordinals of a node supports (reverse direct dependencies).
node-macro =/ %s"DT_N" path-id %s"_SUPPORTS_ORDS"
```

(continues on next page)

(continued from previous page)

```
; --------------------------------------------------------------------
; pinctrl-macro: a macro related to the pinctrl properties in a node
;
; These are a bit of a special case because they kind of form an array,
; but the array indexes correspond to pinctrl-DIGIT properties in a node.
;
; So they're related to a node, but not just one property within the node.
;
; The following examples assume something like this:
;
;      foo {
;              pinctrl-0 = <&bar>;
;              pinctrl-1 = <&baz>;
;              pinctrl-names = "default", "sleep";
;      };

; Total number of pinctrl-DIGIT properties in the node. May be zero.
;
;   #define DT_N_<node path>_PINCTRL_NUM 2
pinctrl-macro = %s"DT_N" path-id %s"_PINCTRL_NUM"
; A given pinctrl-DIGIT property exists.
;
;     #define DT_N_<node path>_PINCTRL_IDX_0_EXISTS 1
;     #define DT_N_<node path>_PINCTRL_IDX_1_EXISTS 1
pinctrl-macro =/ %s"DT_N" path-id %s"_PINCTRL_IDX_" DIGIT %s"_EXISTS"
; A given pinctrl property name exists.
;
;     #define DT_N_<node path>_PINCTRL_NAME_default_EXISTS 1
;     #define DT_N_<node path>_PINCTRL_NAME_sleep_EXISTS 1
pinctrl-macro =/ %s"DT_N" path-id %s"_PINCTRL_NAME_" dt-name %s"_EXISTS"
; The corresponding index number of a named pinctrl property.
;
;     #define DT_N_<node path>_PINCTRL_NAME_default_IDX 0
;     #define DT_N_<node path>_PINCTRL_NAME_sleep_IDX 1
pinctrl-macro =/ %s"DT_N" path-id %s"_PINCTRL_NAME_" dt-name %s"_IDX"
; The node identifier for the phandle in a named pinctrl property.
;
;    #define DT_N_<node path>_PINCTRL_NAME_default_IDX_0_PH <node id for 'bar'>
;
; There's no need for a separate macro for access by index: that's
; covered by property-macro. We only need this because the map from
; names to properties is implicit in the structure of the DT.
pinctrl-macro =/ %s"DT_N" path-id %s"_PINCTRL_NAME_" dt-name %s"_IDX_" DIGIT %s"_PH"

; --------------------------------------------------------------------
; gpiohogs-macro: a macro related to GPIO hog nodes
;
; The following examples assume something like this:
;
;     gpio1: gpio@... {
;       compatible = "vnd,gpio";
;       #gpio-cells = <2>;
;
;       node-1 {
;               gpio-hog;
;               gpios = <0x0 0x10>, <0x1 0x20>;
;               output-high;
;       };
;
;       node-2 {
```

(continues on next page)

(continued from previous page)

```
;               gpio-hog;
;               gpios = <0x2 0x30>;
;               output-low;
;       };
;     };
;
; Bindings fragment for the vnd,gpio compatible:
;
;     gpio-cells:
;       - pin
;       - flags

; The node contains GPIO hogs.
;
;   #define DT_N_<node-1 path>_GPIO_HOGS_EXISTS 1
;   #define DT_N_<node-2 path>_GPIO_HOGS_EXISTS 1
gpioshogs-macro = %s"DT_N" path-id %s"_GPIO_HOGS_EXISTS"
; Number of hogged GPIOs in a node.
;
;   #define DT_N_<node-1 path>_GPIO_HOGS_NUM 2
;   #define DT_N_<node-2 path>_GPIO_HOGS_NUM 1
gpioshogs-macro =/ %s"DT_N" path-id %s"_GPIO_HOGS_NUM"
; A given logical GPIO hog array index exists.
;
;   #define DT_N_<node-1 path>_GPIO_HOGS_IDX_0_EXISTS 1
;   #define DT_N_<node-1 path>_GPIO_HOGS_IDX_1_EXISTS 1
;   #define DT_N_<node-2 path>_GPIO_HOGS_IDX_0_EXISTS 1
gpiohogs-macro =/ %s"DT_N" path-id %s"_GPIO_HOGS_IDX_" DIGIT %s"_EXISTS"
; The node identifier for the phandle of a logical index in the GPIO hogs array.
; These macros are currently unused by Zephyr.
;
;   #define DT_N_<node-1 path>_GPIO_HOGS_IDX_0_PH <node id for 'gpio1'>
;   #define DT_N_<node-1 path>_GPIO_HOGS_IDX_1_PH <node id for 'gpio1'>
;   #define DT_N_<node-2 path>_GPIO_HOGS_IDX_0_PH <node id for 'gpio1'>
gpiohogs-macro =/ %s"DT_N" path-id %s"_GPIO_HOGS_IDX_" DIGIT %s"_PH"
; The pin cell of a logical index in the GPIO hogs array exists.
;
;   #define DT_N_<node-1 path>_GPIO_HOGS_IDX_0_VAL_pin_EXISTS 1
;   #define DT_N_<node-1 path>_GPIO_HOGS_IDX_1_VAL_pin_EXISTS 1
;   #define DT_N_<node-2 path>_GPIO_HOGS_IDX_0_VAL_pin_EXISTS 1
gpiohogs-macro =/ %s"DT_N" path-id %s"_GPIO_HOGS_IDX_" DIGIT %s"_VAL_pin_EXISTS"
; The value of the pin cell of a logical index in the GPIO hogs array.
;
;   #define DT_N_<node-1 path>_GPIO_HOGS_IDX_0_VAL_pin 0
;   #define DT_N_<node-1 path>_GPIO_HOGS_IDX_1_VAL_pin 1
;   #define DT_N_<node-2 path>_GPIO_HOGS_IDX_0_VAL_pin 2
gpiohogs-macro =/ %s"DT_N" path-id %s"_GPIO_HOGS_IDX_" DIGIT %s"_VAL_pin"
; The flags cell of a logical index in the GPIO hogs array exists.
;
;   #define DT_N_<node-1 path>_GPIO_HOGS_IDX_0_VAL_flags_EXISTS 1
;   #define DT_N_<node-1 path>_GPIO_HOGS_IDX_1_VAL_flags_EXISTS 1
;   #define DT_N_<node-2 path>_GPIO_HOGS_IDX_0_VAL_flags_EXISTS 1
gpiohogs-macro =/ %s"DT_N" path-id %s"_GPIO_HOGS_IDX_" DIGIT %s"_VAL_flags_EXISTS"
; The value of the flags cell of a logical index in the GPIO hogs array.
;
;   #define DT_N_<node-1 path>_GPIO_HOGS_IDX_0_VAL_flags 0x10
;   #define DT_N_<node-1 path>_GPIO_HOGS_IDX_1_VAL_flags 0x20
;   #define DT_N_<node-2 path>_GPIO_HOGS_IDX_0_VAL_flags 0x30
gpiohogs-macro =/ %s"DT_N" path-id %s"_GPIO_HOGS_IDX_" DIGIT %s"_VAL_flags"

; ------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
; property-macro: a macro related to a node property
;
; These combine a node identifier with a "lowercase-and-underscores form"
; property name. The value expands to something related to the property's
; value.
;
; The optional prop-suf suffix is when there's some specialized
; subvalue that deserves its own macro, like the macros for an array
; property's individual elements
;
; The "plain vanilla" macro for a property's value, with no prop-suf,
; looks like this:
;
;   DT_N_<node path>_P_<property name>
;
; Components:
;
; - path-id: node's devicetree path converted to a C token
; - prop-id: node's property name converted to a C token
; - prop-suf: an optional property-specific suffix
property-macro =  %s"DT_N" path-id %s"_P_" prop-id [prop-suf]

; ------------------------------------------------------------------------
; path-id: a node's path-based macro identifier
;
; This in "lowercase-and-underscores" form. I.e. it is
; the node's devicetree path converted to a C token by changing:
;
; - each slash (/) to _S_
; - all letters to lowercase
; - non-alphanumerics characters to underscores
;
; For example, the leaf node "bar-BAZ" in this devicetree:
;
;   / {
;           foo@123 {
;                   bar-BAZ {};
;           };
;   };
;
; has path-id "_S_foo_123_S_bar_baz".
path-id = 1*( %s"_S_" dt-name )

; ------------------------------------------------------------------------
; prop-id: a property identifier
;
; A property name converted to a C token by changing:
;
; - all letters to lowercase
; - non-alphanumeric characters to underscores
;
; Example node:
;
;   chosen {
;       zephyr,console = &uart1;
;       WHY,AM_I_SHOUTING = "unclear";
;   };
;
; The 'zephyr,console' property has prop-id 'zephyr_console'.
; 'WHY,AM_I_SHOUTING' has prop-id 'why_am_i_shouting'.
prop-id = dt-name
```

(continues on next page)

(continued from previous page)

```
; ------------------------------------------------------------------------
; prop-suf: a property-specific macro suffix
;
; Extra macros are generated for properties:
;
; - that are special to the specification ("reg", "interrupts", etc.)
; - with array types (uint8-array, phandle-array, etc.)
; - with "enum:" in their bindings
; - that have zephyr device API specific macros for phandle-arrays
; - related to phandle specifier names ("foo-names")
;
; Here are some examples:
;
; - _EXISTS: property, index or name existence flag
; - _SIZE: logical property length
; - _IDX_<i>: values of individual array elements
; - _IDX_<DIGIT>_VAL_<dt-name>: values of individual specifier
;   cells within a phandle array
; - _ADDR_<i>: for reg properties, the i-th register block address
; - _LEN_<i>: for reg properties, the i-th register block length
;
; The different cases are not exhaustively documented here to avoid
; this file going stale. Please see devicetree.h if you need to know
; the details.
prop-suf = 1*( "_" gen-name ["_" dt-name] )

; ------------------------------------------------------------------------
; other-macro: grab bag for everything that isn't a node-macro.

; See examples below.
other-macro =  %s"DT_N_" alternate-id
; Total count of enabled instances of a compatible.
other-macro =/ %s"DT_N_INST_" dt-name %s"_NUM_OKAY"
; These are used internally by DT_FOREACH_NODE and
; DT_FOREACH_STATUS_OKAY_NODE respectively.
other-macro =/ %s"DT_FOREACH_HELPER"
other-macro =/ %s"DT_FOREACH_OKAY_HELPER"
; These are used internally by DT_FOREACH_STATUS_OKAY,
; which iterates over each enabled node of a compatible.
other-macro =/ %s"DT_FOREACH_OKAY_" dt-name
other-macro =/ %s"DT_FOREACH_OKAY_VARGS_" dt-name
; These are used internally by DT_INST_FOREACH_STATUS_OKAY,
; which iterates over each enabled instance of a compatible.
other-macro =/ %s"DT_FOREACH_OKAY_INST_" dt-name
other-macro =/ %s"DT_FOREACH_OKAY_INST_VARGS_" dt-name
; E.g.: #define DT_CHOSEN_zephyr_flash
other-macro =/ %s"DT_CHOSEN_" dt-name
; Declares that a compatible has at least one node on a bus.
; Example:
;
;   #define DT_COMPAT_vnd_dev_BUS_spi 1
other-macro =/ %s"DT_COMPAT_" dt-name %s"_BUS_" dt-name
; Declares that a compatible has at least one status "okay" node.
; Example:
;
;   #define DT_COMPAT_HAS_OKAY_vnd_dev 1
other-macro =/ %s"DT_COMPAT_HAS_OKAY_" dt-name
; Currently used to allow mapping a lowercase-and-underscores "label"
; property to a fixed-partitions node. See the flash map API docs
; for an example.
```

(continues on next page)

(continued from previous page)

```
other-macro =/ %s"DT_COMPAT_" dt-name %s"_LABEL_" dt-name

; ------------------------------------------------------------------
; alternate-id: another way to specify a node besides a path-id
;
; Example devicetree:
;
;   / {
;           aliases {
;                   dev = &dev_1;
;           };
;
;           soc {
;               dev_1: device@123 {
;                   compatible = "vnd,device";
;               };
;           };
;   };
;
; Node device@123 has these alternate-id values:
;
; - ALIAS_dev
; - NODELABEL_dev_1
; - INST_0_vnd_device
;
; The full alternate-id macros are:
;
;   #define DT_N_INST_0_vnd_device     DT_N_S_soc_S_device_123
;   #define DT_N_ALIAS_dev             DT_N_S_soc_S_device_123
;   #define DT_N_NODELABEL_dev_1       DT_N_S_soc_S_device_123
;
; These mainly exist to allow pasting an alternate-id macro onto a
; "_P_<prop-id>" to access node properties given a node's alias, etc.
;
; Notice that "inst"-type IDs have a leading instance identifier,
; which is generated by the devicetree scripts. The other types of
; alternate-id begin immediately with names taken from the devicetree.
alternate-id =  ( %s"ALIAS" / %s"NODELABEL" ) dt-name
alternate-id =/ %s"INST_" 1*DIGIT "_" dt-name

; ------------------------------------------------------------------
; miscellaneous helper definitions

; A dt-name is one or more:
; - lowercase ASCII letters (a-z)
; - numbers (0-9)
; - underscores ("_")
;
; They are the result of converting names or combinations of names
; from devicetree to a valid component of a C identifier by
; lowercasing letters (in practice, this is a no-op) and converting
; non-alphanumeric characters to underscores.
;
; You'll see these referred to as "lowercase-and-underscores" forms of
; various devicetree identifiers throughout the documentation.
dt-name = 1*( lower / DIGIT / "_" )

; gen-name is used as a stand-in for a component of a generated macro
; name which does not come from devicetree (dt-name covers that case).
;
; - uppercase ASCII letters (a-z)
```

(continues on next page)

(continued from previous page)

```
; - numbers (0-9)
; - underscores ("_")
gen-name = upper 1*( upper / DIGIT / "_" )

; "lowercase ASCII letter" turns out to be pretty annoying to specify
; in RFC-7405 syntax.
;
; This is just ASCII letters a (0x61) through z (0x7a).
lower = %x61-7A

; "uppercase ASCII letter" in RFC-7405 syntax
upper = %x41-5A
```

Phandles

The devicetree concept of a *phandle* is very similar to pointers in C. You can use phandles to refer to nodes in devicetree similarly to the way you can use pointers to refer to structures in C.

Contents

- *Getting phandles*
- *Using phandles*
  - *One node: phandle type*
  - *Zero or more nodes: phandles type*
  - *Zero or more nodes with metadata: phandle-array type*
- *phandle-array properties*
  - *High level description*
  - *Example phandle-arrays: GPIOs*
- *Specifier spaces*
  - *High level description*
  - *Example specifier space: gpio*
- *Associating properties with specifier spaces*
  - *High level description*
  - *Special case: GPIO*
  - *Manually specifying a space*
- *Naming the cells in a specifier*
- *See also*

Getting phandles The usual way to get a phandle for a devicetree node is from one of its node labels. For example, with this devicetree:

```
/ {
        lbl_a: node-1 {};
        lbl_b: lbl_c: node-2 {};
};
```

You can write the phandle for:

- /node-1 as &lbl_a
- /node-2 as either &lbl_b or &lbl_c

Notice how the &nodelabel devicetree syntax is similar to the "address of" C syntax.

Note

"Type" in this section refers to one of the type names documented in *Properties* in the devicetree bindings documentation.

Using phandles Here are the main ways you will use phandles.

One node: phandle type You can use phandles to refer to node-b from node-a, where node-b is related to node-a in some way.

One common example is when node-a represents some hardware that generates an interrupt, and node-b represents the interrupt controller that receives the asserted interrupt. In this case, you could write:

```
node_b: node-b {
        interrupt-controller;
};

node-a {
        interrupt-parent = <&node_b>;
};
```

This uses the standard interrupt-parent property defined in the devicetree specification to capture the relationship between the two nodes.

These properties have type phandle.

Zero or more nodes: phandles type You can use phandles to make an array of references to other nodes.

One common example occurs in *pin control*. Pin control properties like pinctrl-0, pinctrl-1 etc. may contain multiple phandles, each of which "points" to a node containing information related to pin configuration for that hardware peripheral. Here's an example of six phandles in a single property:

```
pinctrl-0 = <&quadspi_clk_pe10 &quadspi_ncs_pe11
             &quadspi_bk1_io0_pe12 &quadspi_bk1_io1_pe13
             &quadspi_bk1_io2_pe14 &quadspi_bk1_io3_pe15>;
```

These properties have type phandles.

Zero or more nodes with metadata: phandle-array type You can use phandles to refer to and configure one or more resources that are "owned" by some other node.

This is the most complex case. There are examples and more details in the next section.

These properties have type phandle-array.

phandle-array properties These properties are commonly used to specify a resource that is owned by another node along with additional metadata about the resource.

High level description Usually, properties with this type are written like `phandle-array-prop` in this example:

```
node {
        phandle-array-prop = <&foo 1 2>, <&bar 3>, <&baz 4 5>;
};
```

That is, the property's value is written as a comma-separated sequence of "groups", where each "group" is written inside of angle brackets (`< ... >`). Each "group" starts with a phandle (`&foo`, `&bar`, `&baz`). The values that follow the phandle in each "group" are called *specifiers*. There are three specifiers in the above example:

1. `1 2`
2. `3`
3. `4 5`

The phandle in each "group" is used to "point" to the hardware that controls the resource you are interested in. The specifier describes the resource itself, along with any additional necessary metadata.

The rest of this section describes a common example. Subsequent sections document more rules about how to use phandle-array properties in practice.

Example phandle-arrays: GPIOs Perhaps the most common use case for phandle-array properties is specifying one or more GPIOs on your SoC that another chip on your board connects to. For that reason, we'll focus on that use case here. However, there are **many other use cases** that are handled in devicetree with phandle-array properties.

For example, consider an external chip with an interrupt pin that is connected to a GPIO on your SoC. You will typically need to provide that GPIO's information (GPIO controller and pin number) to the *device driver* for that chip. You usually also need to provide other metadata about the GPIO, like whether it is active low or high, what kind of internal pull resistor within the SoC should be enabled in order to communicate with the device, etc., to the driver.

In the devicetree, there will be a node that represents the GPIO controller that controls a group of pins. This reflects the way GPIO IP blocks are usually developed in hardware. Therefore, there is no single node in the devicetree that represents a GPIO pin, and you can't use a single phandle to represent it.

Instead, you would use a phandle-array property, like this:

```
my-external-ic {
        irq-gpios = <&gpioX pin flags>;
};
```

In this example, `irq-gpios` is a phandle-array property with just one "group" in its value. `&gpioX` is the phandle for the GPIO controller node that controls the pin. `pin` is the pin number (0, 1, 2, ...). `flags` is a bit mask describing pin metadata (for example `(GPIO_ACTIVE_LOW | GPIO_PULL_UP)`); see include/zephyr/dt-bindings/gpio/gpio.h for more details.

The device driver handling the `my-external-ic` node can then use the `irq-gpios` property's value to set up interrupt handling for the chip as it is used on your board. This lets you configure the device driver in devicetree, without changing the driver's source code.

Such properties can contain multiple values as well:

```
my-other-external-ic {
        handshake-gpios = <&gpioX pinX flagsX>, <&gpioY pinY flagsY>;
};
```

The above example specifies two pins:

- `pinX` on the GPIO controller with phandle `&gpioX`, flags `flagsX`
- `pinY` on `&gpioY`, flags `flagsY`

You may be wondering how the "pin and flags" convention is established and enforced. To answer this question, we'll need to introduce a concept called specifier spaces before moving on to some information about devicetree bindings.

Specifier spaces *Specifier spaces* are a way to allow nodes to describe how you should use them in phandle-array properties.

We'll start with an abstract, high level description of how specifier spaces work in DTS files, before moving on to a concrete example and providing references to further reading for how this all works in practice using DTS files and bindings files.

High level description As described above, a phandle-array property is a sequence of "groups" of phandles followed by some number of cells:

```
node {
        phandle-array-prop = <&foo 1 2>, <&bar 3>;
};
```

The cells that follow each phandle are called a *specifier*. In this example, there are two specifiers:

1. `1 2`: two cells
2. `3`: one cell

Every phandle-array property has an associated *specifier space*. This sounds complex, but it's really just a way to assign a meaning to the cells that follow each phandle in a hardware specific way. Every specifier space has a unique name. There are a few "standard" names for commonly used hardware, but you can create your own as well.

Devicetree nodes encode the number of cells that must appear in a specifier, by name, using the `#SPACE_NAME-cells` property. For example, let's assume that `phandle-array-prop`'s specifier space is named `baz`. Then we would need the `foo` and `bar` nodes to have the following `#baz-cells` properties:

```
foo: node@1000 {
        #baz-cells = <2>;
};

bar: node@2000 {
        #baz-cells = <1>;
};
```

Without the `#baz-cells` property, the devicetree tooling would not be able to validate the number of cells in each specifier in `phandle-array-prop`.

This flexibility allows you to write down an array of hardware resources in a single devicetree property, even though the amount of metadata you need to describe each resource might be different for different nodes.

A single node can also have different numbers of cells in different specifier spaces. For example, we might have:

```
foo: node@1000 {
        #baz-cells = <2>;
        #bob-cells = <1>;
};
```

With that, if `phandle-array-prop-2` has specifier space `bob`, we could write:

```
node {
        phandle-array-prop = <&foo 1 2>, <&bar 3>;
        phandle-array-prop-2 = <&foo 4>;
};
```

This flexibility allows you to have a node that manages multiple different kinds of resources at the same time. The node describes the amount of metadata needed to describe each kind of resource (how many cells are needed in each case) using different `#SPACE_NAME-cells` properties.

Example specifier space: gpio From the above example, you're already familiar with how one specifier space works: in the "gpio" space, specifiers almost always have two cells:

1. a pin number
2. a bit mask of flags related to the pin

Therefore, almost all GPIO controller nodes you will see in practice will look like this:

```
gpioX: gpio-controller@deadbeef {
        gpio-controller;
        #gpio-cells = <2>;
};
```

Associating properties with specifier spaces Above, we have described that:

- each phandle-array property has an associated specifier space
- specifier spaces are identified by name
- devicetree nodes use `#SPECIFIER_NAME-cells` properties to configure the number of cells which must appear in a specifier

In this section, we explain how phandle-array properties get their specifier spaces.

High level description In general, a `phandle-array` property named `foos` implicitly has specifier space `foo`. For example:

```
properties:
  dmas:
    type: phandle-array
  pwms:
    type: phandle-array
```

The `dmas` property's specifier space is "dma". The `pwm` property's specifier space is `pwm`.

Special case: GPIO `*-gpios` properties are special-cased so that e.g. `foo-gpios` resolves to `#gpio-cells` rather than `#foo-gpio-cells`.

Manually specifying a space You can manually specify the specifier space for any `phandle-array` property. See *specifier-space*.

Naming the cells in a specifier You should name the cells in each specifier space your hardware supports when writing bindings. For details on how to do this, see *Specifier cell names (*-cells)*.

This allows C code to query information about and retrieve the values of cells in a specifier by name using devicetree APIs like these:

- *DT_PHA_BY_IDX*
- *DT_PHA_BY_NAME*

This feature and these macros are used internally by numerous hardware-specific APIs. Here are a few examples:

- *DT_GPIO_PIN_BY_IDX*
- *DT_PWMS_CHANNEL_BY_IDX*
- *DT_DMAS_CELL_BY_NAME*
- *DT_IO_CHANNELS_INPUT_BY_IDX*
- *DT_CLOCKS_CELL_BY_NAME*

See also

- *Writing property values*: how to write phandles in devicetree properties
- *Properties*: how to write bindings for properties with phandle types (`phandle`, `phandles`, `phandle-array`)
- *specifier-space*: how to manually specify a phandle-array property's specifier space

The `/zephyr,user` node

Zephyr's devicetree scripts handle the `/zephyr,user` node as a special case: you can put essentially arbitrary properties inside it and retrieve their values without having to write a binding. It is meant as a convenient container when only a few simple properties are needed.

Note

This node is meant for sample code and user applications. It should not be used in the upstream Zephyr source code for device drivers, subsystems, etc.

Simple values You can store numeric or array values in `/zephyr,user` if you want them to be configurable at build time via devicetree.

For example, with this devicetree overlay:

```
/ {
    zephyr,user {
            boolean;
            bytes = [81 82 83];
            number = <23>;
            numbers = <1>, <2>, <3>;
            string = "text";
            strings = "a", "b", "c";
    };
};
```

You can get the above property values in C/C++ code like this:

```
#define ZEPHYR_USER_NODE DT_PATH(zephyr_user)

DT_PROP(ZEPHYR_USER_NODE, boolean) // 1
DT_PROP(ZEPHYR_USER_NODE, bytes)   // {0x81, 0x82, 0x83}
DT_PROP(ZEPHYR_USER_NODE, number)  // 23
```

(continues on next page)

(continued from previous page)

```
DT_PROP(ZEPHYR_USER_NODE, numbers) // {1, 2, 3}
DT_PROP(ZEPHYR_USER_NODE, string)  // "text"
DT_PROP(ZEPHYR_USER_NODE, strings) // {"a", "b", "c"}
```

Devices You can store *phandles* in `/zephyr,user` if you want to be able to reconfigure which devices your application uses in simple cases using devicetree overlays.

For example, with this devicetree overlay:

```
/ {
    zephyr,user {
            handle = <&gpio0>;
            handles = <&gpio0>, <&gpio1>;
    };
};
```

You can convert the phandles in the `handle` and `handles` properties to device pointers like this:

```
/*
 * Same thing as:
 *
 * ... my_dev = DEVICE_DT_GET(DT_NODELABEL(gpio0));
 */
const struct device *my_device =
    DEVICE_DT_GET(DT_PROP(ZEPHYR_USER_NODE, handle));

#define PHANDLE_TO_DEVICE(node_id, prop, idx) \
    DEVICE_DT_GET(DT_PHANDLE_BY_IDX(node_id, prop, idx)),

/*
 * Same thing as:
 *
 * ... *my_devices[] = {
 *         DEVICE_DT_GET(DT_NODELABEL(gpio0)),
 *         DEVICE_DT_GET(DT_NODELABEL(gpio1)),
 * };
 */
const struct device *my_devices[] = {
    DT_FOREACH_PROP_ELEM(ZEPHYR_USER_NODE, handles, PHANDLE_TO_DEVICE)
};
```

GPIOs The `/zephyr,user` node is a convenient place to store application-specific GPIOs that you want to be able to reconfigure with a devicetree overlay.

For example, with this devicetree overlay:

```
#include <zephyr/dt-bindings/gpio/gpio.h>

/ {
    zephyr,user {
            signal-gpios = <&gpio0 1 GPIO_ACTIVE_HIGH>;
    };
};
```

You can convert the pin defined in `signal-gpios` to a `struct gpio_dt_spec` in your source code, then use it like this:

```
#include <zephyr/drivers/gpio.h>

#define ZEPHYR_USER_NODE DT_PATH(zephyr_user)

const struct gpio_dt_spec signal =
        GPIO_DT_SPEC_GET(ZEPHYR_USER_NODE, signal_gpios);

/* Configure the pin */
gpio_pin_configure_dt(&signal, GPIO_OUTPUT_INACTIVE);

/* Set the pin to its active level */
gpio_pin_set_dt(&signal, 1);
```

(See *gpio_dt_spec*, *GPIO_DT_SPEC_GET*, and *gpio_pin_configure_dt()* for details on these APIs.)

Devicetree HOWTOs

This page has step-by-step advice for getting things done with devicetree.

> **Tip**
>
> See *Troubleshooting devicetree* for troubleshooting advice.

Get your devicetree and generated header A board's devicetree (*BOARD.dts*) pulls in common node definitions via `#include` preprocessor directives. This at least includes the SoC's `.dtsi`. One way to figure out the devicetree's contents is by opening these files, e.g. by looking in `dts/<ARCH>/<vendor>/<soc>.dtsi`, but this can be time consuming.

If you just want to see the "final" devicetree for your board, build an application and open the `zephyr.dts` file in the build directory.

> **Tip**
>
> You can build hello_world to see the "base" devicetree for your board without any additional changes from *overlay files*.

For example, using the qemu_cortex_m3 board to build hello_world:

```
# --cmake-only here just forces CMake to run, skipping the
# build process to save time.
west build -b qemu_cortex_m3 samples/hello_world --cmake-only
```

You can change `qemu_cortex_m3` to match your board.

CMake prints the input and output file locations like this:

```
-- Found BOARD.dts: .../zephyr/boards/arm/qemu_cortex_m3/qemu_cortex_m3.dts
-- Generated zephyr.dts: .../zephyr/build/zephyr/zephyr.dts
-- Generated devicetree_generated.h: .../zephyr/build/zephyr/include/generated/zephyr/
↪devicetree_generated.h
```

The `zephyr.dts` file is the final devicetree in DTS format.

The `devicetree_generated.h` file is the corresponding generated header.

See *Input and output files* for details about these files.

Get a struct device from a devicetree node When writing Zephyr applications, you'll often want to get a driver-level *struct device* corresponding to a devicetree node.

For example, with this devicetree fragment, you might want the struct device for serial@40002000:

```
/ {
        soc {
                serial0: serial@40002000 {
                        status = "okay";
                        current-speed = <115200>;
                        /* ... */
                };
        };

        aliases {
                my-serial = &serial0;
        };

        chosen {
                zephyr,console = &serial0;
        };
};
```

Start by making a *node identifier* for the device you are interested in. There are different ways to do this; pick whichever one works best for your requirements. Here are some examples:

```
/* Option 1: by node label */
#define MY_SERIAL DT_NODELABEL(serial0)

/* Option 2: by alias */
#define MY_SERIAL DT_ALIAS(my_serial)

/* Option 3: by chosen node */
#define MY_SERIAL DT_CHOSEN(zephyr_console)

/* Option 4: by path */
#define MY_SERIAL DT_PATH(soc, serial_40002000)
```

Once you have a node identifier there are two ways to proceed. One way to get a device is to use *DEVICE_DT_GET()*:

```
const struct device *const uart_dev = DEVICE_DT_GET(MY_SERIAL);

if (!device_is_ready(uart_dev)) {
        /* Not ready, do not use */
        return -ENODEV;
}
```

There are variants of *DEVICE_DT_GET()* such as *DEVICE_DT_GET_OR_NULL()*, *DEVICE_DT_GET_ONE()* or *DEVICE_DT_GET_ANY()*. This idiom fetches the device pointer at build-time, which means there is no runtime penalty. This method is useful if you want to store the device pointer as configuration data. But because the device may not be initialized, or may have failed to initialize, you must verify that the device is ready to be used before passing it to any API functions. (This check is done for you by *device_get_binding()*.)

In some situations the device cannot be known at build-time, e.g., if it depends on user input like in a shell application. In this case you can get the `struct device` by combining *device_get_binding()* with the device name:

```
const char *dev_name = /* TODO: insert device name from user */;
const struct device *uart_dev = device_get_binding(dev_name);
```

You can then use `uart_dev` with *Universal Asynchronous Receiver-Transmitter (UART)* API functions like *uart_configure()*. Similar code will work for other device types; just make sure you use the correct API for the device.

If you're having trouble, see *Troubleshooting devicetree*. The first thing to check is that the node has `status = "okay"`, like this:

```
#define MY_SERIAL DT_NODELABEL(my_serial)

#if DT_NODE_HAS_STATUS(MY_SERIAL, okay)
const struct device *const uart_dev = DEVICE_DT_GET(MY_SERIAL);
#else
#error "Node is disabled"
#endif
```

If you see the `#error` output, make sure to enable the node in your devicetree. In some situations your code will compile but it will fail to link with a message similar to:

```
...undefined reference to `__device_dts_ord_N'
collect2: error: ld returned 1 exit status
```

This likely means there's a Kconfig issue preventing the device driver from being built, resulting in a reference that does not exist. If your code compiles successfully, the last thing to check is if the device is ready, like this:

```
if (!device_is_ready(uart_dev)) {
     printk("Device not ready\n");
}
```

If you find that the device is not ready, it likely means that the device's initialization function failed. Enabling logging or debugging driver code may help in such situations. Note that you can also use *device_get_binding()* to obtain a reference at runtime. If it returns `NULL` it can either mean that device's driver failed to initialize or that it does not exist.

Find a devicetree binding *Devicetree bindings* are YAML files which declare what you can do with the nodes they describe, so it's critical to be able to find them for the nodes you are using.

If you don't have them already, *Get your devicetree and generated header*. To find a node's binding, open the generated header file, which starts with a list of nodes in a block comment:

```
/*
 * [...]
 * Nodes in dependency order (ordinal and path):
 *   0   /
 *   1   /aliases
 *   2   /chosen
 *   3   /flash@0
 *   4   /memory@20000000
 *          (etc.)
 * [...]
 */
```

Make note of the path to the node you want to find, like `/flash@0`. Search for the node's output in the file, which starts with something like this if the node has a matching binding:

```
/*
 * Devicetree node:
 *   /flash@0
 *
 * Binding (compatible = soc-nv-flash):
 *   $ZEPHYR_BASE/dts/bindings/mtd/soc-nv-flash.yaml
```

(continues on next page)

(continued from previous page)

```
 * [...]
 */
```

See *Check for missing bindings* for troubleshooting.

Set devicetree overlays Devicetree overlays are explained in *Introduction to devicetree*. The CMake variable **DTC_OVERLAY_FILE** contains a space- or semicolon-separated list of overlay files to use. If **DTC_OVERLAY_FILE** specifies multiple files, they are included in that order by the C preprocessor. A file in a Zephyr module can be referred to by escaping the Zephyr module dir variable like `\${ZEPHYR_<module>_MODULE_DIR}/<path-to>/dts.overlay` when setting the DTC_OVERLAY_FILE variable.

You can set **DTC_OVERLAY_FILE** to contain exactly the files you want to use. Here is an *example* using `west build`.

If you don't set **DTC_OVERLAY_FILE**, the build system will follow these steps, looking for files in your application configuration directory to use as devicetree overlays:

1. If the file `socs/<SOC>_<BOARD_QUALIFIERS>.overlay` exists, it will be used.
2. If the file `boards/<BOARD>.overlay` exists, it will be used in addition to the above.
3. If the current board has *multiple revisions* and `boards/<BOARD>_<revision>.overlay` exists, it will be used in addition to the above.
4. If one or more files have been found in the previous steps, the build system stops looking and just uses those files.
5. Otherwise, if `<BOARD>.overlay` exists, it will be used, and the build system will stop looking for more files.
6. Otherwise, if `app.overlay` exists, it will be used.

Extra devicetree overlays may be provided using `EXTRA_DTC_OVERLAY_FILE` which will still allow the build system to automatically use devicetree overlays described in the above steps.

The build system appends overlays specified in `EXTRA_DTC_OVERLAY_FILE` to the overlays in `DTC_OVERLAY_FILE` when processing devicetree overlays. This means that changes made via `EXTRA_DTC_OVERLAY_FILE` have higher precedence than those made via `DTC_OVERLAY_FILE`.

All configuration files will be taken from the application's configuration directory except for files with an absolute path that are given with the `DTC_OVERLAY_FILE` or `EXTRA_DTC_OVERLAY_FILE` argument.

See *Application Configuration Directory* on how the application configuration directory is defined.

Using *Shields* will also add devicetree overlay files.

The **DTC_OVERLAY_FILE** value is stored in the CMake cache and used in successive builds.

The *build system* prints all the devicetree overlays it finds in the configuration phase, like this:

```
-- Found devicetree overlay: .../some/file.overlay
```

Use devicetree overlays See *Set devicetree overlays* for how to add an overlay to the build.

Overlays can override node property values in multiple ways. For example, if your BOARD.dts contains this node:

```
/ {
        soc {
                serial0: serial@40002000 {
                        status = "okay";
                        current-speed = <115200>;
                        /* ... */
                };
        };
};
```

These are equivalent ways to override the `current-speed` value in an overlay:

```
/* Option 1 */
&serial0 {
     current-speed = <9600>;
};

/* Option 2 */
&{/soc/serial@40002000} {
     current-speed = <9600>;
};
```

We'll use the `&serial0` style for the rest of these examples.

You can add aliases to your devicetree using overlays: an alias is just a property of the `/aliases` node. For example:

```
/ {
     aliases {
             my-serial = &serial0;
     };
};
```

Chosen nodes work the same way. For example:

```
/ {
     chosen {
             zephyr,console = &serial0;
     };
};
```

To delete a property (in addition to deleting properties in general, this is how to set a boolean property to false if it's true in BOARD.dts):

```
&serial0 {
     /delete-property/ some-unwanted-property;
};
```

You can add subnodes using overlays. For example, to configure a SPI or I2C child device on an existing bus node, do something like this:

```
/* SPI device example */
&spi1 {
     my_spi_device: temp-sensor@0 {
             compatible = "...";
             label = "TEMP_SENSOR_0";
             /* reg is the chip select number, if needed;
              * If present, it must match the node's unit address. */
             reg = <0>;

             /* Configure other SPI device properties as needed.
              * Find your device's DT binding for details. */
```

(continues on next page)

(continued from previous page)

```
            spi-max-frequency = <4000000>;
    };
};

/* I2C device example */
&i2c2 {
    my_i2c_device: touchscreen@76 {
            compatible = "...";
            label = "TOUCHSCREEN";
            /* reg is the I2C device address.
             * It must match the node's unit address. */
            reg = <76>;

            /* Configure other I2C device properties as needed.
             * Find your device's DT binding for details. */
    };
};
```

Other bus devices can be configured similarly:

- create the device as a subnode of the parent bus
- set its properties according to its binding

Assuming you have a suitable device driver associated with the `my_spi_device` and `my_i2c_device` compatibles, you should now be able to enable the driver via Kconfig and *get the struct device* for your newly added bus node, then use it with that driver API.

Write device drivers using devicetree APIs "Devicetree-aware" *device drivers* should create a `struct device` for each `status = "okay"` devicetree node with a particular *compatible* (or related set of compatibles) supported by the driver.

Writing a devicetree-aware driver begins by defining a *devicetree binding* for the devices supported by the driver. Use existing bindings from similar drivers as a starting point. A skeletal binding to get started needs nothing more than this:

```
description: <Human-readable description of your binding>
compatible: "foo-company,bar-device"
include: base.yaml
```

See *Find a devicetree binding* for more advice on locating existing bindings.

After writing your binding, your driver C file can then use the devicetree API to find `status = "okay"` nodes with the desired compatible, and instantiate a `struct device` for each one. There are two options for instantiating each `struct device`: using instance numbers, and using node labels.

In either case:

- Each `struct device`'s name should be set to its devicetree node's `label` property. This allows the driver's users to *Get a struct device from a devicetree node* in the usual way.
- Each device's initial configuration should use values from devicetree properties whenever practical. This allows users to configure the driver using *devicetree overlays*.

Examples for how to do this follow. They assume you've already implemented the device-specific configuration and data structures and API functions, like this:

```
/* my_driver.c */
#include <zephyr/drivers/some_api.h>

/* Define data (RAM) and configuration (ROM) structures: */
```

(continues on next page)

(continued from previous page)

```
struct my_dev_data {
     /* per-device values to store in RAM */
};
struct my_dev_cfg {
     uint32_t freq; /* Just an example: initial clock frequency in Hz */
     /* other configuration to store in ROM */
};

/* Implement driver API functions (drivers/some_api.h callbacks): */
static int my_driver_api_func1(const struct device *dev, uint32_t *foo) { /* ... */ }
static int my_driver_api_func2(const struct device *dev, uint64_t bar) { /* ... */ }
static struct some_api my_api_funcs = {
     .func1 = my_driver_api_func1,
     .func2 = my_driver_api_func2,
};
```

Option 1: create devices using instance numbers Use this option, which uses *Instance-based APIs*, if possible. However, they only work when devicetree nodes for your driver's `compatible` are all equivalent, and you do not need to be able to distinguish between them.

To use instance-based APIs, begin by defining `DT_DRV_COMPAT` to the lowercase-and-underscores version of the compatible that the device driver supports. For example, if your driver's compatible is `"vnd,my-device"` in devicetree, you would define `DT_DRV_COMPAT` to `vnd_my_device` in your driver C file:

```
/*
 * Put this near the top of the file. After the includes is a good place.
 * (Note that you can therefore run "git grep DT_DRV_COMPAT drivers" in
 * the zephyr Git repository to look for example drivers using this style).
 */
#define DT_DRV_COMPAT vnd_my_device
```

Important

As shown, the DT_DRV_COMPAT macro should have neither quotes nor special characters. Remove quotes and convert special characters to underscores when creating `DT_DRV_COMPAT` from the compatible property.

Finally, define an instantiation macro, which creates each `struct device` using instance numbers. Do this after defining `my_api_funcs`.

```
/*
 * This instantiation macro is named "CREATE_MY_DEVICE".
 * Its "inst" argument is an arbitrary instance number.
 *
 * Put this near the end of the file, e.g. after defining "my_api_funcs".
 */
#define CREATE_MY_DEVICE(inst)                                          \
     static struct my_dev_data my_data_##inst = {                       \
             /* initialize RAM values as needed, e.g.: */               \
             .freq = DT_INST_PROP(inst, clock_frequency),               \
     };                                                                 \
     static const struct my_dev_cfg my_cfg_##inst = {                   \
             /* initialize ROM values as needed. */                     \
     };                                                                 \
     DEVICE_DT_INST_DEFINE(inst,                                        \
                           my_dev_init_function,                        \
```

(continues on next page)

(continued from previous page)

```
                        NULL,                                   \
                        &my_data_##inst,                        \
                        &my_cfg_##inst,                         \
                        MY_DEV_INIT_LEVEL, MY_DEV_INIT_PRIORITY,  \
                        &my_api_funcs);
```

Notice the use of APIs like *DT_INST_PROP()* and *DEVICE_DT_INST_DEFINE()* to access devicetree node data. These APIs retrieve data from the devicetree for instance number `inst` of the node with compatible determined by `DT_DRV_COMPAT`.

Finally, pass the instantiation macro to *DT_INST_FOREACH_STATUS_OKAY()*:

```
/* Call the device creation macro for each instance: */
DT_INST_FOREACH_STATUS_OKAY(CREATE_MY_DEVICE)
```

`DT_INST_FOREACH_STATUS_OKAY` expands to code which calls `CREATE_MY_DEVICE` once for each enabled node with the compatible determined by `DT_DRV_COMPAT`. It does not append a semicolon to the end of the expansion of `CREATE_MY_DEVICE`, so the macro's expansion must end in a semicolon or function definition to support multiple devices.

Option 2: create devices using node labels Some device drivers cannot use instance numbers. One example is an SoC peripheral driver which relies on vendor HAL APIs specialized for individual IP blocks to implement Zephyr driver callbacks. Cases like this should use *DT_NODELABEL()* to refer to individual nodes in the devicetree representing the supported peripherals on the SoC. The devicetree.h *Generic APIs* can then be used to access node data.

For this to work, your *SoC's dtsi file* must define node labels like `mydevice0`, `mydevice1`, etc. appropriately for the IP blocks your driver supports. The resulting devicetree usually looks something like this:

```
/ {
        soc {
                mydevice0: dev@0 {
                        compatible = "vnd,my-device";
                };
                mydevice1: dev@1 {
                        compatible = "vnd,my-device";
                };
        };
};
```

The driver can use the `mydevice0` and `mydevice1` node labels in the devicetree to operate on specific device nodes:

```
/*
 * This is a convenience macro for creating a node identifier for
 * the relevant devices. An example use is MYDEV(0) to refer to
 * the node with label "mydevice0".
 */
#define MYDEV(idx) DT_NODELABEL(mydevice ## idx)

/*
 * Define your instantiation macro; "idx" is a number like 0 for mydevice0
 * or 1 for mydevice1. It uses MYDEV() to create the node label from the
 * index.
 */
#define CREATE_MY_DEVICE(idx)                                   \
     static struct my_dev_data my_data_##idx = {                \
             /* initialize RAM values as needed, e.g.: */       \
             .freq = DT_PROP(MYDEV(idx), clock_frequency),      \
```

(continues on next page)

(continued from previous page)

```
    };                                                              \
    static const struct my_dev_cfg my_cfg_##idx = { /* ... */ };    \
    DEVICE_DT_DEFINE(MYDEV(idx),                                    \
                     my_dev_init_function,                          \
                     NULL,                                          \
                     &my_data_##idx,                                \
                     &my_cfg_##idx,                                 \
                     MY_DEV_INIT_LEVEL, MY_DEV_INIT_PRIORITY,       \
                     &my_api_funcs)
```

Notice the use of APIs like *DT_PROP()* and *DEVICE_DT_DEFINE()* to access devicetree node data.

Finally, manually detect each enabled devicetree node and use `CREATE_MY_DEVICE` to instantiate each `struct device`:

```
#if DT_NODE_HAS_STATUS(DT_NODELABEL(mydevice0), okay)
CREATE_MY_DEVICE(0)
#endif

#if DT_NODE_HAS_STATUS(DT_NODELABEL(mydevice1), okay)
CREATE_MY_DEVICE(1)
#endif
```

Since this style does not use `DT_INST_FOREACH_STATUS_OKAY()`, the driver author is responsible for calling `CREATE_MY_DEVICE()` for every possible node, e.g. using knowledge about the peripherals available on supported SoCs.

Device drivers that depend on other devices At times, one `struct device` depends on another `struct device` and requires a pointer to it. For example, a sensor device might need a pointer to its SPI bus controller device. Some advice:

- Write your devicetree binding in a way that permits use of *Hardware specific APIs* from devicetree.h if possible.
- In particular, for bus devices, your driver's binding should include a file like dts/bindings/spi/spi-device.yaml which provides common definitions for devices addressable via a specific bus. This enables use of APIs like *DT_BUS()* to obtain a node identifier for the bus node. You can then *Get a struct device from a devicetree node* for the bus in the usual way.

Search existing bindings and device drivers for examples.

Applications that depend on board-specific devices One way to allow application code to run unmodified on multiple boards is by supporting a devicetree alias to specify the hardware specific portions, as is done in the blinky sample. The application can then be configured in *BOARD.dts* files or via *devicetree overlays*.

Troubleshooting devicetree

Here are some tips for fixing misbehaving devicetree related code.

See *Devicetree HOWTOs* for other "HOWTO" style information.

Important

Try this first, before doing anything else.

Try again with a pristine build directory See *Pristine Builds* for examples, or just delete the build directory completely and retry.

This is general advice which is especially applicable to debugging devicetree issues, because the outputs are created during the CMake configuration phase, and are not always regenerated when one of their inputs changes.

Make sure <devicetree.h> is included Unlike Kconfig symbols, the `devicetree.h` header must be included explicitly.

Many Zephyr header files rely on information from devicetree, so including some other API may transitively include `devicetree.h`, but that's not guaranteed.

undefined reference to __device_dts_ord_<N> This usually happens on a line like this:

```
const struct device *dev = DEVICE_DT_GET(NODE_ID);
```

where `NODE_ID` is a valid *node identifier*, but no device driver has allocated a `struct device` for this devicetree node. You thus get a linker error, because you're asking for a pointer to a device that isn't defined.

To fix it, you need to make sure that:

1. The node is enabled: the node must have `status = "okay";`.

   (Recall that a missing `status` property means the same thing as `status = "okay";`; see *Important properties* for more information about `status`).

2. A device driver responsible for allocating the `struct device` is enabled. That is, the Kconfig option which makes the build system compile the driver sources into your application needs to be set to `y`.

   (See *Setting Kconfig configuration values* for more information on setting Kconfig options.)

Below, `<build>` means your build directory.

Making sure the node is enabled:

To find the devicetree node you need to check, use the number `<N>` from the linker error. Look for this number in the list of nodes at the top of `<build>/zephyr/include/generated/zephyr/devicetree_generated.h`. For example, if `<N>` is 15, and your `devicetree_generated.h` file looks like this, the node you are interested in is `/soc/i2c@deadbeef`:

```
/*
 * Generated by gen_defines.py
 *
 * DTS input file:
 *   <build>/zephyr/zephyr.dts.pre
 *
 * Directories with bindings:
 *   $ZEPHYR_BASE/dts/bindings
 *
 * Node dependency ordering (ordinal and path):
 *   0   /
 *   1   /aliases
[...]
 *   15  /soc/i2c@deadbeef
[...]
```

Now look for this node in `<build>/zephyr/zephyr.dts`, which is the final devicetree for your application build. (See *Get your devicetree and generated header* for information and examples.)

If the node has `status = "disabled";` in `zephyr.dts`, then you need to enable it by setting `status = "okay";`, probably by using a devicetree *overlay*. For example, if `zephyr.dts` looks like this:

```
i2c0: i2c@deadbeef {
        status = "disabled";
};
```

Then you should put this into your devicetree overlay and *Try again with a pristine build directory*:

```
&i2c0 {
        status = "okay";
};
```

Make sure that you see `status = "okay";` in `zephyr.dts` after you rebuild.

Making sure the device driver is enabled:

The first step is to figure out which device driver is responsible for handling your devicetree node and allocating devices for it. To do this, you need to start with the `compatible` property in your devicetree node, and find the driver that allocates `struct device` instances for that compatible.

If you're not familiar with how devices are allocated from devicetree nodes based on compatible properties, the ZDS 2021 talk A deep dive into the Zephyr 2.5 device model may be a useful place to start, along with the *Device Driver Model* pages. See *Important properties* and the Devicetree specification for more information about `compatible`.

There is currently no documentation for what device drivers exist and which devicetree compatibles they are associated with. You will have to figure this out by reading the source code:

- Look in drivers for the appropriate subdirectory that corresponds to the API your device implements
- Look inside that directory for relevant files until you figure out what the driver is, or realize there is no such driver.

Often, but not always, you can find the driver by looking for a file that sets the `DT_DRV_COMPAT` macro to match your node's `compatible` property, except lowercased and with special characters converted to underscores. For example, if your node's compatible is `vnd,foo-device`, look for a file with this line:

```
#define DT_DRV_COMPAT vnd_foo_device
```

Important

This **does not always work** since not all drivers use `DT_DRV_COMPAT`.

If you find a driver, you next need to make sure the Kconfig option that compiles it is enabled. (If you don't find a driver, and you are sure the compatible property is correct, then you need to write a driver. Writing drivers is outside the scope of this documentation page.)

Continuing the above example, if your devicetree node looks like this now:

```
i2c0: i2c@deadbeef {
        compatible = "nordic,nrf-twim";
        status = "okay";
};
```

Then you would look inside of drivers/i2c for the driver file that handles the compatible `nordic,nrf-twim`. In this case, that is drivers/i2c/i2c_nrfx_twim.c. Notice how even in cases where `DT_DRV_COMPAT` is not set, you can use information like driver file names as clues.

Once you know the driver you want to enable, you need to make sure its Kconfig option is set to `y`. You can figure out which Kconfig option is needed by looking for a line similar to

this one in the `CMakeLists.txt` file in the drivers subdirectory. Continuing the above example, drivers/i2c/CMakeLists.txt has a line that looks like this:

```
zephyr_library_sources_ifdef(CONFIG_NRFX_TWIM       i2c_nrfx_twim.c)
```

This means that `CONFIG_NRFX_TWIM` must be set to `y` in `<build>/zephyr/.config` file.

If your driver's Kconfig is not set to `y`, you need to figure out what you need to do to make that happen. Often, this will happen automatically as soon as you enable the devicetree node. Otherwise, it is sometimes as simple as adding a line like this to your application's `prj.conf` file and then making sure to *Try again with a pristine build directory*:

```
CONFIG_FOO=y
```

where `CONFIG_FOO` is the option that `CMakeLists.txt` uses to decide whether or not to compile the driver.

However, there may be other problems in your way, such as unmet Kconfig dependencies that you also have to enable before you can enable your driver.

Consult the Kconfig file that defines `CONFIG_FOO` (for your value of `FOO`) for more information.

Make sure you're using the right names Remember that:

- In C/C++, devicetree names must be lowercased and special characters must be converted to underscores. Zephyr's generated devicetree header has DTS names converted in this way into the C tokens used by the preprocessor-based `<devicetree.h>` API.
- In overlays, use devicetree node and property names the same way they would appear in any DTS file. Zephyr overlays are just DTS fragments.

For example, if you're trying to **get** the `clock-frequency` property of a node with path `/soc/i2c@12340000` in a C/C++ file:

```
/*
 * foo.c: lowercase-and-underscores names
 */

/* Don't do this: */
#define MY_CLOCK_FREQ DT_PROP(DT_PATH(soc, i2c@1234000), clock-frequency)
/*                                          ^             ^
 *                                    @ should be _     - should be _  */

/* Do this instead: */
#define MY_CLOCK_FREQ DT_PROP(DT_PATH(soc, i2c_1234000), clock_frequency)
/*                                          ^               ^           */
```

And if you're trying to **set** that property in a devicetree overlay:

```
/*
 * foo.overlay: DTS names with special characters, etc.
 */

/* Don't do this; you'll get devicetree errors. */
&{/soc/i2c_12340000/} {
     clock_frequency = <115200>;
};

/* Do this instead. Overlays are just DTS fragments. */
&{/soc/i2c@12340000/} {
     clock-frequency = <115200>;
};
```

Look at the preprocessor output To save preprocessor output files, enable the CONFIG_COMPILER_SAVE_TEMPS option. For example, to build hello_world with west with this option set, use:

```
west build -b BOARD samples/hello_world -- -DCONFIG_COMPILER_SAVE_TEMPS=y
```

This will create a preprocessor output file named `foo.c.i` in the build directory for each source file `foo.c`.

You can then search for the file in the build directory to see what your devicetree macros expanded to. For example, on macOS and Linux, using `find` to find `main.c.i`:

```
$ find build -name main.c.i
build/CMakeFiles/app.dir/src/main.c.i
```

It's usually easiest to run a style formatter on the results before opening them. For example, to use `clang-format` to reformat the file in place:

```
clang-format -i build/CMakeFiles/app.dir/src/main.c.i
```

You can then open the file in your favorite editor to view the final C results after preprocessing.

Do not track macro expansion Compiler messages for devicetree errors can sometimes be very long. This typically happens when the compiler prints a message for every step of a complex macro expansion that has several intermediate expansion steps.

To prevent the compiler from doing this, you can disable the CONFIG_COMPILER_TRACK_MACRO_EXPANSION option. This typically reduces the output to one message per error.

For example, to build hello_world with west and this option disabled, use:

```
west build -b BOARD samples/hello_world -- -DCONFIG_COMPILER_TRACK_MACRO_EXPANSION=n
```

Validate properties If you're getting a compile error reading a node property, check your node identifier and property. For example, if you get a build error on a line that looks like this:

```
int baud_rate = DT_PROP(DT_NODELABEL(my_serial), current_speed);
```

Try checking the node by adding this to the file and recompiling:

```
#if !DT_NODE_EXISTS(DT_NODELABEL(my_serial))
#error "whoops"
#endif
```

If you see the "whoops" error message when you rebuild, the node identifier isn't referring to a valid node. *Get your devicetree and generated header* and debug from there.

Some hints for what to check next if you don't see the "whoops" error message:

- did you *Make sure you're using the right names*?
- does the *property exist*?
- does the node have a *matching binding*?
- does the binding define the property?

Check for missing bindings See *Devicetree bindings* for information about bindings, and *Bindings index* for information on bindings built into Zephyr.

If the build fails to *Find a devicetree binding* for a node, then either the node's `compatible` property is not defined, or its value has no matching binding. If the property is set, check for typos in its name. In a devicetree source file, `compatible` should look like `"vnd,some-device"` – *Make sure you're using the right names*.

If your binding file is not under `zephyr/dts`, you may need to set *DTS_ROOT*; see *Where bindings are located*.

Errors with DT_INST_() APIs If you're using an API like *DT_INST_PROP()*, you must define `DT_DRV_COMPAT` to the lowercase-and-underscores version of the compatible you are interested in. See *Option 1: create devices using instance numbers*.

Devicetree versus Kconfig

Along with devicetree, Zephyr also uses the Kconfig language to configure the source code. Whether to use devicetree or Kconfig for a particular purpose can sometimes be confusing. This section should help you decide which one to use.

In short:

- Use devicetree to describe **hardware** and its **boot-time configuration**. Examples include peripherals on a board, boot-time clock frequencies, interrupt lines, etc.
- Use Kconfig to configure **software support** to build into the final image. Examples include whether to add networking support, which drivers are needed by the application, etc.

In other words, devicetree mainly deals with hardware, and Kconfig with software.

For example, consider a board containing a SoC with 2 UART, or serial port, instances.

- The fact that the board has this UART **hardware** is described with two UART nodes in the devicetree. These provide the UART type (via the `compatible` property) and certain settings such as the address range of the hardware peripheral registers in memory (via the `reg` property).
- Additionally, the UART **boot-time configuration** is also described with devicetree. This could include configuration such as the RX IRQ line's priority and the UART baud rate. These may be modifiable at runtime, but their boot-time configuration is described in devicetree.
- Whether or not to include **software support** for UART in the build is controlled via Kconfig. Applications which do not need to use the UARTs can remove the driver source code from the build using Kconfig, even though the board's devicetree still includes UART nodes.

As another example, consider a device with a 2.4GHz, multi-protocol radio supporting both the Bluetooth Low Energy and 802.15.4 wireless technologies.

- Devicetree should be used to describe the presence of the radio **hardware**, what driver or drivers it's compatible with, etc.
- **Boot-time configuration** for the radio, such as TX power in dBm, should also be specified using devicetree.
- Kconfig should determine which **software features** should be built for the radio, such as selecting a BLE or 802.15.4 protocol stack.

As another example, Kconfig options that formerly enabled a particular instance of a driver (that is itself enabled by Kconfig) have been removed. The devices are selected individually using devicetree's *status* keyword on the corresponding hardware instance.

There are **exceptions** to these rules:

- Because Kconfig is unable to flexibly control some instance-specific driver configuration parameters, such as the size of an internal buffer, these options may be defined in devicetree. However, to make clear that they are specific to Zephyr drivers and not hardware description or configuration these properties should be prefixed with `zephyr,`, e.g. `zephyr,random-mac-address` in the common Ethernet devicetree properties.
- Devicetree's `chosen` keyword, which allows the user to select a specific instance of a hardware device to be used for a particular purpose. An example of this is selecting a particular UART for use as the system's console.

5.2.2 Devicetree Reference

These pages contain reference material for Zephyr's devicetree APIs and built-in bindings.

For the platform-independent details, see the Devicetree specification.

Devicetree API

This is a reference page for the `<zephyr/devicetree.h>` API. The API is macro based. Use of these macros has no impact on scheduling. They can be used from any calling context and at file scope.

Some of these – the ones beginning with `DT_INST_` – require a special macro named `DT_DRV_COMPAT` to be defined before they can be used; these are discussed individually below. These macros are generally meant for use within *device drivers*, though they can be used outside of drivers with appropriate care.

Contents

- *Generic APIs*
  - *Node identifiers and helpers*
  - *Property access*
  - *`ranges` property*
  - *`reg` property*
  - *`interrupts` property*
  - *For-each macros*
  - *Existence checks*
  - *Inter-node dependencies*
  - *Bus helpers*
- *Instance-based APIs*
- *Hardware specific APIs*
  - *CAN*
  - *Clocks*
  - *DMA*
  - *Fixed flash partitions*
  - *GPIO*
  - *IO channels*

  - *MBOX*
  - *Pinctrl (pin control)*
  - *PWM*
  - *Reset Controller*
  - *SPI*
- *Chosen nodes*
- *Zephyr-specific chosen nodes*

Generic APIs The APIs in this section can be used anywhere and do not require `DT_DRV_COMPAT` to be defined.

Node identifiers and helpers A *node identifier* is a way to refer to a devicetree node at C preprocessor time. While node identifiers are not C values, you can use them to access devicetree data in C rvalue form using, for example, the *Property access* API.

The root node `/` has node identifier `DT_ROOT`. You can create node identifiers for other devicetree nodes using *`DT_PATH()`*, *`DT_NODELABEL()`*, *`DT_ALIAS()`*, and *`DT_INST()`*.

There are also *`DT_PARENT()`* and *`DT_CHILD()`* macros which can be used to create node identifiers for a given node's parent node or a particular child node, respectively.

The following macros create or operate on node identifiers.

Related code samples

GPIO with custom Devicetree binding
Use custom Devicetree binding to control a GPIO.

group `devicetree-generic-id`

Defines

`DT_INVALID_NODE`

Name for an invalid node identifier.

This supports cases where factored macros can be invoked from paths where devicetree data may or may not be available. It is a preprocessor identifier that does not match any valid devicetree node identifier.

`DT_ROOT`

Node identifier for the root node in the devicetree.

`DT_PATH(...)`

Get a node identifier for a devicetree path.

The arguments to this macro are the names of non-root nodes in the tree required to reach the desired node, starting from the root. Non-alphanumeric characters in each

name must be converted to underscores to form valid C tokens, and letters must be lowercased.

Example devicetree fragment:

```
/ {
        soc {
                serial1: serial@40001000 {
                        status = "okay";
                        current-speed = <115200>;
                        ...
                };
        };
};
```

You can use *DT_PATH(soc, serial_40001000)* to get a node identifier for the `serial@40001000` node. Node labels like `serial1` cannot be used as *DT_PATH()* arguments; use *DT_NODELABEL()* for those instead.

Example usage with *DT_PROP()* to get the `current-speed` property:

```
DT_PROP(DT_PATH(soc, serial_40001000), current_speed) // 115200
```

(The `current-speed` property is also in `lowercase-and-underscores` form when used with this API.)

When determining arguments to *DT_PATH()*:

- the first argument corresponds to a child node of the root (`soc` above)
- a second argument corresponds to a child of the first argument (`serial_40001000` above, from the node name `serial@40001000` after lowercasing and changing `@` to `_`)
- and so on for deeper nodes in the desired node's path

Note

This macro returns a node identifier from path components. To get a path string from a node identifier, use *DT_NODE_PATH()* instead.

Parameters

- ... – lowercase-and-underscores node names along the node's path, with each name given as a separate argument

Returns

node identifier for the node with that path

`DT_NODELABEL`(label)

Get a node identifier for a node label.

Convert non-alphanumeric characters in the node label to underscores to form valid C tokens, and lowercase all letters. Note that node labels are not the same thing as label properties.

Example devicetree fragment:

```
serial1: serial@40001000 {
        label = "UART_0";
        status = "okay";
```

(continues on next page)

(continued from previous page)

```
        current-speed = <115200>;
        ...
};
```

The only node label in this example is `serial1`.

The string `UART_0` is *not* a node label; it's the value of a property named label.

You can use *DT_NODELABEL(serial1)* to get a node identifier for the `serial@40001000` node. Example usage with *DT_PROP()* to get the current-speed property:

```
DT_PROP(DT_NODELABEL(serial1), current_speed) // 115200
```

Another example devicetree fragment:

```
cpu@0 {
       L2_0: l2-cache {
               cache-level = <2>;
               ...
       };
};
```

Example usage to get the cache-level property:

```
DT_PROP(DT_NODELABEL(l2_0), cache_level) // 2
```

Notice how `L2_0` in the devicetree is lowercased to `l2_0` in the *DT_NODELABEL()* argument.

Parameters

- **label** – lowercase-and-underscores node label name

Returns
node identifier for the node with that label

`DT_ALIAS`(alias)

Get a node identifier from /aliases.

This macro's argument is a property of the `/aliases` node. It returns a node identifier for the node which is aliased. Convert non-alphanumeric characters in the alias property to underscores to form valid C tokens, and lowercase all letters.

Example devicetree fragment:

```
/ {
        aliases {
                my-serial = &serial1;
        };

        soc {
                serial1: serial@40001000 {
                        status = "okay";
                        current-speed = <115200>;
                        ...
                };
        };
};
```

You can use *DT_ALIAS(my_serial)* to get a node identifier for the `serial@40001000` node. Notice how `my-serial` in the devicetree becomes `my_serial` in the *DT_ALIAS()* argument. Example usage with *DT_PROP()* to get the current-speed property:

```
DT_PROP(DT_ALIAS(my_serial), current_speed) // 115200
```

Parameters

- **alias** – lowercase-and-underscores alias name.

Returns
node identifier for the node with that alias

DT_INST(inst, compat)

Get a node identifier for an instance of a compatible.

All nodes with a particular compatible property value are assigned instance numbers, which are zero-based indexes specific to that compatible. You can get a node identifier for these nodes by passing *DT_INST()* an instance number, `inst`, along with the lowercase-and-underscores version of the compatible, `compat`.

Instance numbers have the following properties:

- for each compatible, instance numbers start at 0 and are contiguous
- exactly one instance number is assigned for each node with a compatible, **including disabled nodes**
- enabled nodes (status property is `okay` or missing) are assigned the instance numbers starting from 0, and disabled nodes have instance numbers which are greater than those of any enabled node

No other guarantees are made. In particular:

- instance numbers **in no way reflect** any numbering scheme that might exist in SoC documentation, node labels or unit addresses, or properties of the /aliases node (use *DT_NODELABEL()* or *DT_ALIAS()* for those)
- there **is no general guarantee** that the same node will have the same instance number between builds, even if you are building the same application again in the same build directory

Example devicetree fragment:

```
serial1: serial@40001000 {
        compatible = "vnd,soc-serial";
        status = "disabled";
        current-speed = <9600>;
        ...
};

serial2: serial@40002000 {
        compatible = "vnd,soc-serial";
        status = "okay";
        current-speed = <57600>;
        ...
};

serial3: serial@40003000 {
        compatible = "vnd,soc-serial";
        current-speed = <115200>;
        ...
};
```

Assuming no other nodes in the devicetree have compatible "vnd,soc-serial", that compatible has nodes with instance numbers 0, 1, and 2.

The nodes serial@40002000 and serial@40003000 are both enabled, so their instance numbers are 0 and 1, but no guarantees are made regarding which node has which instance number.

Since serial@40001000 is the only disabled node, it has instance number 2, since disabled nodes are assigned the largest instance numbers. Therefore:

```
// Could be 57600 or 115200. There is no way to be sure:
// either serial@40002000 or serial@40003000 could
// have instance number 0, so this could be the current-speed
// property of either of those nodes.
DT_PROP(DT_INST(0, vnd_soc_serial), current_speed)

// Could be 57600 or 115200, for the same reason.
// If the above expression expands to 57600, then
// this expands to 115200, and vice-versa.
DT_PROP(DT_INST(1, vnd_soc_serial), current_speed)

// 9600, because there is only one disabled node, and
// disabled nodes are "at the end" of the instance
// number "list".
DT_PROP(DT_INST(2, vnd_soc_serial), current_speed)
```

Notice how "vnd,soc-serial" in the devicetree becomes vnd_soc_serial (without quotes) in the *DT_INST()* arguments. (As usual, current-speed in the devicetree becomes current_speed as well.)

Nodes whose compatible property has multiple values are assigned independent instance numbers for each compatible.

Parameters

- **inst** – instance number for compatible compat
- **compat** – lowercase-and-underscores compatible, without quotes

Returns

node identifier for the node with that instance number and compatible

DT_PARENT(node_id)

Get a node identifier for a parent node.

Example devicetree fragment:

```
parent: parent-node {
        child: child-node {
                ...
        };
};
```

The following are equivalent ways to get the same node identifier:

```
DT_NODELABEL(parent)
DT_PARENT(DT_NODELABEL(child))
```

Parameters

- **node_id** – node identifier

Returns

a node identifier for the node's parent

DT_GPARENT(node_id)

Get a node identifier for a grandparent node.

Example devicetree fragment:

```
gparent: grandparent-node {
        parent: parent-node {
                child: child-node { ... }
        };
};
```

The following are equivalent ways to get the same node identifier:

```
DT_GPARENT(DT_NODELABEL(child))
DT_PARENT(DT_PARENT(DT_NODELABEL(child))
```

Parameters

- **node_id** – node identifier

Returns

a node identifier for the node's parent's parent

DT_CHILD(node_id, child)

Get a node identifier for a child node.

Example devicetree fragment:

```
/ {
        soc-label: soc {
                serial1: serial@40001000 {
                        status = "okay";
                        current-speed = <115200>;
                        ...
                };
        };
};
```

Example usage with *DT_PROP()* to get the status of the `serial@40001000` node:

```
#define SOC_NODE DT_NODELABEL(soc_label)
DT_PROP(DT_CHILD(SOC_NODE, serial_40001000), status) // "okay"
```

Node labels like `serial1` cannot be used as the `child` argument to this macro. Use *DT_NODELABEL()* for that instead.

You can also use *DT_FOREACH_CHILD()* to iterate over node identifiers for all of a node's children.

Parameters

- **node_id** – node identifier
- **child** – lowercase-and-underscores child node name

Returns

node identifier for the node with the name referred to by 'child'

DT_COMPAT_GET_ANY_STATUS_OKAY(compat)

Get a node identifier for a status okay node with a compatible.

Use this if you want to get an arbitrary enabled node with a given compatible, and you do not care which one you get. If any enabled nodes with the given compatible exist, a node identifier for one of them is returned. Otherwise, *DT_INVALID_NODE* is returned.

Example devicetree fragment:

```
node-a {
   compatible = "vnd,device";
   status = "okay";
};

node-b {
   compatible = "vnd,device";
   status = "okay";
};

node-c {
   compatible = "vnd,device";
   status = "disabled";
};
```

Example usage:

```
DT_COMPAT_GET_ANY_STATUS_OKAY(vnd_device)
```

This expands to a node identifier for either `node-a` or `node-b`. It will not expand to a node identifier for `node-c`, because that node does not have status `okay`.

Parameters

- **compat** – lowercase-and-underscores compatible, without quotes

Returns
node identifier for a node with that compatible, or *DT_INVALID_NODE*

`DT_NODE_PATH`(node_id)

Get a devicetree node's full path as a string literal.

This returns the path to a node from a node identifier. To get a node identifier from path components instead, use *DT_PATH()*.

Example devicetree fragment:

```
/ {
        soc {
                node: my-node@12345678 { ... };
        };
};
```

Example usage:

```
DT_NODE_PATH(DT_NODELABEL(node)) // "/soc/my-node@12345678"
DT_NODE_PATH(DT_PATH(soc))       // "/soc"
DT_NODE_PATH(DT_ROOT)            // "/"
```

Parameters

- **node_id** – node identifier

Returns
the node's full path in the devicetree

`DT_NODE_FULL_NAME`(node_id)

Get a devicetree node's name with unit-address as a string literal.

This returns the node name and unit-address from a node identifier.

Example devicetree fragment:

```
/ {
        soc {
                node: my-node@12345678 { ... };
        };
};
```

Example usage:

```
DT_NODE_FULL_NAME(DT_NODELABEL(node)) // "my-node@12345678"
```

Parameters

- **node_id** – node identifier

Returns
the node's name with unit-address as a string in the devicetree

DT_NODE_CHILD_IDX(node_id)

Get a devicetree node's index into its parent's list of children.

Indexes are zero-based.

It is an error to use this macro with the root node.

Example devicetree fragment:

```
parent {
        c1: child-1 {};
        c2: child-2 {};
};
```

Example usage:

```
DT_NODE_CHILD_IDX(DT_NODELABEL(c1)) // 0
DT_NODE_CHILD_IDX(DT_NODELABEL(c2)) // 1
```

Parameters

- **node_id** – node identifier

Returns
the node's index in its parent node's list of children

DT_CHILD_NUM(node_id)

Get the number of child nodes of a given node.

Parameters

- **node_id** – a node identifier

Returns
Number of child nodes

DT_CHILD_NUM_STATUS_OKAY(node_id)

Get the number of child nodes of a given node which child nodes' status are okay.

Parameters

- **node_id** – a node identifier

Returns
Number of child nodes which status are okay

DT_SAME_NODE(node_id1, node_id2)

Do node_id1 and node_id2 refer to the same node?

Both node_id1 and node_id2 must be node identifiers for nodes that exist in the devicetree (if unsure, you can check with *DT_NODE_EXISTS()*).

The expansion evaluates to 0 or 1, but may not be a literal integer 0 or 1.

Parameters

- **node_id1** – first node identifier
- **node_id2** – second node identifier

Returns

an expression that evaluates to 1 if the node identifiers refer to the same node, and evaluates to 0 otherwise

DT_NODELABEL_STRING_ARRAY(node_id)

Get a devicetree node's node labels as an array of strings.

Example devicetree fragment:

```
foo: bar: node@deadbeef {};
```

Example usage:

```
DT_NODELABEL_STRING_ARRAY(DT_NODELABEL(foo))
```

This expands to:

```
{ "foo", "bar", }
```

Parameters

- **node_id** – node identifier

Returns

an array initializer for an array of the node's node labels as strings

Property access The following general-purpose macros can be used to access node properties. There are special-purpose APIs for accessing the *ranges property*, *reg property* and *interrupts property*.

Property values can be read using these macros even if the node is disabled, as long as it has a matching binding.

group devicetree-generic-prop

Defines

DT_PROP(node_id, prop)

Get a devicetree property value.

For properties whose bindings have the following types, this macro expands to:

- string: a string literal
- boolean: 0 if the property is false, or 1 if it is true
- int: the property's value as an integer literal

- array, uint8-array, string-array: an initializer expression in braces, whose elements are integer or string literals (like `{0, 1, 2}`, `{"hello", "world"}`, etc.)
- phandle: a node identifier for the node with that phandle

A property's type is usually defined by its binding. In some special cases, it has an assumed type defined by the devicetree specification even when no binding is available: `compatible` has type string-array, `status` has type string, and `interrupt-controller` has type boolean.

For other properties or properties with unknown type due to a missing binding, behavior is undefined.

For usage examples, see *DT_PATH()*, *DT_ALIAS()*, *DT_NODELABEL()*, and *DT_INST()* above.

Parameters

- `node_id` – node identifier
- `prop` – lowercase-and-underscores property name

Returns
a representation of the property's value

`DT_PROP_LEN`(node_id, prop)

Get a property's logical length.

Here, "length" is a number of elements, which may differ from the property's size in bytes.

The return value depends on the property's type:

- for types array, string-array, and uint8-array, this expands to the number of elements in the array
- for type phandles, this expands to the number of phandles
- for type phandle-array, this expands to the number of phandle and specifier blocks in the property
- for type phandle, this expands to 1 (so that a phandle can be treated as a degenerate case of phandles with length 1)
- for type string, this expands to 1 (so that a string can be treated as a degenerate case of string-array with length 1)

These properties are handled as special cases:

- reg property: use *DT_NUM_REGS(node_id)* instead
- interrupts property: use *DT_NUM_IRQS(node_id)* instead

It is an error to use this macro with the `ranges`, `dma-ranges`, `reg` or `interrupts` properties.

For other properties, behavior is undefined.

Parameters

- `node_id` – node identifier
- `prop` – a lowercase-and-underscores property with a logical length

Returns
the property's length

DT_PROP_LEN_OR(node_id, prop, default_value)

Like *DT_PROP_LEN()*, but with a fallback to `default_value`.

If the property is defined (as determined by *DT_NODE_HAS_PROP()*), this expands to *DT_PROP_LEN(node_id, prop)*. The `default_value` parameter is not expanded in this case.

Otherwise, this expands to `default_value`.

Parameters

- **node_id** – node identifier
- **prop** – a lowercase-and-underscores property with a logical length
- **default_value** – a fallback value to expand to

Returns
the property's length or the given default value

DT_PROP_HAS_IDX(node_id, prop, idx)

Is index `idx` valid for an array type property?

If this returns 1, then *DT_PROP_BY_IDX(node_id, prop, idx)* or *DT_PHA_BY_IDX(node_id, prop, idx, ...)* are valid at index `idx`. If it returns 0, it is an error to use those macros with that index.

These properties are handled as special cases:

- `reg` property: use *DT_REG_HAS_IDX(node_id, idx)* instead
- `interrupts` property: use *DT_IRQ_HAS_IDX(node_id, idx)* instead

It is an error to use this macro with the `reg` or `interrupts` properties.

Parameters

- **node_id** – node identifier
- **prop** – a lowercase-and-underscores property with a logical length
- **idx** – index to check

Returns
An expression which evaluates to 1 if `idx` is a valid index into the given property, and 0 otherwise.

DT_PROP_HAS_NAME(node_id, prop, name)

Is name `name` available in a `foo-names` property?

This property is handled as special case:

- `interrupts` property: use *DT_IRQ_HAS_NAME(node_id, idx)* instead

It is an error to use this macro with the `interrupts` property.

Example devicetree fragment:

```
nx: node-x {
   foos = <&bar xx yy>, <&baz xx zz>;
   foo-names = "event", "error";
   status = "okay";
};
```

Example usage:

```
DT_PROP_HAS_NAME(DT_NODELABEL(nx), foos, event)    // 1
DT_PROP_HAS_NAME(DT_NODELABEL(nx), foos, failure)  // 0
```

Parameters

- **node_id** – node identifier
- **prop** – a lowercase-and-underscores `prop-names` type property
- **name** – a lowercase-and-underscores name to check

Returns
An expression which evaluates to 1 if "name" is an available name into the given property, and 0 otherwise.

`DT_PROP_BY_IDX`(node_id, prop, idx)

Get the value at index `idx` in an array type property.

It might help to read the argument order as being similar to `node->property[index]`.

The return value depends on the property's type:

- for types array, string-array, uint8-array, and phandles, this expands to the idx-th array element as an integer, string literal, integer, and node identifier respectively
- for type phandle, idx must be 0 and the expansion is a node identifier (this treats phandle like a phandles of length 1)
- for type string, idx must be 0 and the expansion is the entire string (this treats string like string-array of length 1)

These properties are handled as special cases:

- `reg`: use *DT_REG_ADDR_BY_IDX()* or *DT_REG_SIZE_BY_IDX()* instead
- `interrupts`: use *DT_IRQ_BY_IDX()*
- `ranges`: use *DT_NUM_RANGES()*
- `dma-ranges`: it is an error to use this property with *DT_PROP_BY_IDX()*

For properties of other types, behavior is undefined.

Parameters

- **node_id** – node identifier
- **prop** – lowercase-and-underscores property name
- **idx** – the index to get

Returns
a representation of the idx-th element of the property

`DT_PROP_OR`(node_id, prop, default_value)

Like *DT_PROP()*, but with a fallback to `default_value`.

If the value exists, this expands to *DT_PROP(node_id, prop)*. The `default_value` parameter is not expanded in this case.

Otherwise, this expands to `default_value`.

Parameters

- **node_id** – node identifier
- **prop** – lowercase-and-underscores property name

- **default_value** – a fallback value to expand to

Returns
the property's value or default_value

DT_ENUM_IDX(node_id, prop)

Get a property value's index into its enumeration values.

The return values start at zero.

Example devicetree fragment:

```
usb1: usb@12340000 {
        maximum-speed = "full-speed";
};
usb2: usb@12341000 {
        maximum-speed = "super-speed";
};
```

Example bindings fragment:

```
properties:
  maximum-speed:
    type: string
    enum:
       - "low-speed"
       - "full-speed"
       - "high-speed"
       - "super-speed"
```

Example usage:

```
DT_ENUM_IDX(DT_NODELABEL(usb1), maximum_speed) // 1
DT_ENUM_IDX(DT_NODELABEL(usb2), maximum_speed) // 3
```

Parameters

- **node_id** – node identifier
- **prop** – lowercase-and-underscores property name

Returns
zero-based index of the property's value in its enum: list

DT_ENUM_IDX_OR(node_id, prop, default_idx_value)

Like *DT_ENUM_IDX()*, but with a fallback to a default enum index.

If the value exists, this expands to its zero based index value thanks to *DT_ENUM_IDX(node_id, prop)*.

Otherwise, this expands to provided default index enum value.

Parameters

- **node_id** – node identifier
- **prop** – lowercase-and-underscores property name
- **default_idx_value** – a fallback index value to expand to

Returns
zero-based index of the property's value in its enum if present, default_idx_value otherwise

DT_ENUM_HAS_VALUE(node_id, prop, value)

Does a node enumeration property have a given value?

Parameters

- node_id – node identifier
- prop – lowercase-and-underscores property name
- value – lowercase-and-underscores enumeration value

Returns

1 if the node property has the value *value*, 0 otherwise.

DT_STRING_TOKEN(node_id, prop)

Get a string property's value as a token.

This removes "the quotes" from a string property's value, converting any non-alphanumeric characters to underscores. This can be useful, for example, when programmatically using the value to form a C variable or code.

DT_STRING_TOKEN() can only be used for properties with string type.

It is an error to use *DT_STRING_TOKEN()* in other circumstances.

Example devicetree fragment:

```
n1: node-1 {
        prop = "foo";
};
n2: node-2 {
        prop = "FOO";
}
n3: node-3 {
        prop = "123 foo";
};
```

Example bindings fragment:

```
properties:
  prop:
    type: string
```

Example usage:

```
DT_STRING_TOKEN(DT_NODELABEL(n1), prop) // foo
DT_STRING_TOKEN(DT_NODELABEL(n2), prop) // FOO
DT_STRING_TOKEN(DT_NODELABEL(n3), prop) // 123_foo
```

Notice how:

- Unlike C identifiers, the property values may begin with a number. It's the user's responsibility not to use such values as the name of a C identifier.
- The uppercased "FOO" in the DTS remains FOO as a token. It is *not* converted to foo.
- The whitespace in the DTS "123 foo" string is converted to 123_foo as a token.

Parameters

- node_id – node identifier
- prop – lowercase-and-underscores property name

Returns
the value of prop as a token, i.e. without any quotes and with special characters converted to underscores

DT_STRING_TOKEN_OR(node_id, prop, default_value)

Like *DT_STRING_TOKEN()*, but with a fallback to default_value.

If the value exists, this expands to *DT_STRING_TOKEN(node_id, prop)*. The default_value parameter is not expanded in this case.

Otherwise, this expands to default_value.

Parameters

- **node_id** – node identifier
- **prop** – lowercase-and-underscores property name
- **default_value** – a fallback value to expand to

Returns
the property's value as a token, or default_value

DT_STRING_UPPER_TOKEN(node_id, prop)

Like *DT_STRING_TOKEN()*, but uppercased.

This removes "the quotes" from a string property's value, converting any non-alphanumeric characters to underscores, and capitalizing the result. This can be useful, for example, when programmatically using the value to form a C variable or code.

DT_STRING_UPPER_TOKEN() can only be used for properties with string type.

It is an error to use *DT_STRING_UPPER_TOKEN()* in other circumstances.

Example devicetree fragment:

```
n1: node-1 {
        prop = "foo";
};
n2: node-2 {
        prop = "123 foo";
};
```

Example bindings fragment:

```
properties:
  prop:
    type: string
```

Example usage:

```
DT_STRING_UPPER_TOKEN(DT_NODELABEL(n1), prop) // FOO
DT_STRING_UPPER_TOKEN(DT_NODELABEL(n2), prop) // 123_FOO
```

Notice how:

- Unlike C identifiers, the property values may begin with a number. It's the user's responsibility not to use such values as the name of a C identifier.
- The lowercased "foo" in the DTS becomes FOO as a token, i.e. it is uppercased.
- The whitespace in the DTS "123 foo" string is converted to 123_FOO as a token, i.e. it is uppercased and whitespace becomes an underscore.

Parameters

- **node_id** – node identifier
- **prop** – lowercase-and-underscores property name

Returns
the value of prop as an uppercased token, i.e. without any quotes and with special characters converted to underscores

DT_STRING_UPPER_TOKEN_OR(node_id, prop, default_value)

Like *DT_STRING_UPPER_TOKEN()*, but with a fallback to default_value.

If the value exists, this expands to *DT_STRING_UPPER_TOKEN(node_id, prop)*. The default_value parameter is not expanded in this case.

Otherwise, this expands to default_value.

Parameters

- **node_id** – node identifier
- **prop** – lowercase-and-underscores property name
- **default_value** – a fallback value to expand to

Returns
the property's value as an uppercased token, or default_value

DT_STRING_UNQUOTED(node_id, prop)

Get a string property's value as an unquoted sequence of tokens.

This removes "the quotes" from string-valued properties. That can be useful, for example, when defining floating point values as a string in devicetree that you would like to use to initialize a float or double variable in C.

DT_STRING_UNQUOTED() can only be used for properties with string type.

It is an error to use *DT_STRING_UNQUOTED()* in other circumstances.

Example devicetree fragment:

```
n1: node-1 {
        prop = "12.7";
};
n2: node-2 {
        prop = "0.5";
}
n3: node-3 {
        prop = "A B C";
};
```

Example bindings fragment:

```
properties:
  prop:
    type: string
```

Example usage:

```
DT_STRING_UNQUOTED(DT_NODELABEL(n1), prop) // 12.7
DT_STRING_UNQUOTED(DT_NODELABEL(n2), prop) // 0.5
DT_STRING_UNQUOTED(DT_NODELABEL(n3), prop) // A B C
```

Parameters

- **node_id** – node identifier
- **prop** – lowercase-and-underscores property name

Returns
the property's value as a sequence of tokens, with no quotes

`DT_STRING_UNQUOTED_OR`(node_id, prop, default_value)

Like *DT_STRING_UNQUOTED()*, but with a fallback to `default_value`.

If the value exists, this expands to *DT_STRING_UNQUOTED(node_id, prop)*. The `default_value` parameter is not expanded in this case.

Otherwise, this expands to `default_value`.

Parameters

- **node_id** – node identifier
- **prop** – lowercase-and-underscores property name
- **default_value** – a fallback value to expand to

Returns
the property's value as a sequence of tokens, with no quotes, or `default_value`

`DT_STRING_TOKEN_BY_IDX`(node_id, prop, idx)

Get an element out of a string-array property as a token.

This removes "the quotes" from an element in the array, and converts non-alphanumeric characters to underscores. That can be useful, for example, when programmatically using the value to form a C variable or code.

DT_STRING_TOKEN_BY_IDX() can only be used for properties with string-array type.

It is an error to use *DT_STRING_TOKEN_BY_IDX()* in other circumstances.

Example devicetree fragment:

```
n1: node-1 {
        prop = "f1", "F2";
};
n2: node-2 {
        prop = "123 foo", "456 FOO";
};
```

Example bindings fragment:

```
properties:
  prop:
    type: string-array
```

Example usage:

```
DT_STRING_TOKEN_BY_IDX(DT_NODELABEL(n1), prop, 0) // f1
DT_STRING_TOKEN_BY_IDX(DT_NODELABEL(n1), prop, 1) // F2
DT_STRING_TOKEN_BY_IDX(DT_NODELABEL(n2), prop, 0) // 123_foo
DT_STRING_TOKEN_BY_IDX(DT_NODELABEL(n2), prop, 1) // 456_FOO
```

For more information, see *DT_STRING_TOKEN*.

Parameters

- **node_id** – node identifier
- **prop** – lowercase-and-underscores property name
- **idx** – the index to get

Returns
the element in `prop` at index `idx` as a token

DT_STRING_UPPER_TOKEN_BY_IDX(node_id, prop, idx)

Like *DT_STRING_TOKEN_BY_IDX()*, but uppercased.

This removes "the quotes" and capitalizes an element in the array, and converts non-alphanumeric characters to underscores. That can be useful, for example, when programmatically using the value to form a C variable or code.

DT_STRING_UPPER_TOKEN_BY_IDX() can only be used for properties with string-array type.

It is an error to use *DT_STRING_UPPER_TOKEN_BY_IDX()* in other circumstances.

Example devicetree fragment:

```
n1: node-1 {
        prop = "f1", "F2";
};
n2: node-2 {
        prop = "123 foo", "456 FOO";
};
```

Example bindings fragment:

```
properties:
  prop:
    type: string-array
```

Example usage:

```
DT_STRING_UPPER_TOKEN_BY_IDX(DT_NODELABEL(n1), prop, 0) // F1
DT_STRING_UPPER_TOKEN_BY_IDX(DT_NODELABEL(n1), prop, 1) // F2
DT_STRING_UPPER_TOKEN_BY_IDX(DT_NODELABEL(n2), prop, 0) // 123_FOO
DT_STRING_UPPER_TOKEN_BY_IDX(DT_NODELABEL(n2), prop, 1) // 456_FOO
```

For more information, see *DT_STRING_UPPER_TOKEN*.

Parameters

- **node_id** – node identifier
- **prop** – lowercase-and-underscores property name
- **idx** – the index to get

Returns

the element in `prop` at index `idx` as an uppercased token

DT_STRING_UNQUOTED_BY_IDX(node_id, prop, idx)

Get a string array item value as an unquoted sequence of tokens.

This removes "the quotes" from string-valued item. That can be useful, for example, when defining floating point values as a string in devicetree that you would like to use to initialize a float or double variable in C.

DT_STRING_UNQUOTED_BY_IDX() can only be used for properties with string-array type.

It is an error to use *DT_STRING_UNQUOTED_BY_IDX()* in other circumstances.

Example devicetree fragment:

```
n1: node-1 {
        prop = "12.7", "34.1";
};
n2: node-2 {
        prop = "A B", "C D";
}
```

Example bindings fragment:

```
properties:
  prop:
    type: string-array
```

Example usage:

```
DT_STRING_UNQUOTED_BY_IDX(DT_NODELABEL(n1), prop, 0) // 12.7
DT_STRING_UNQUOTED_BY_IDX(DT_NODELABEL(n1), prop, 1) // 34.1
DT_STRING_UNQUOTED_BY_IDX(DT_NODELABEL(n2), prop, 0) // A B
DT_STRING_UNQUOTED_BY_IDX(DT_NODELABEL(n2), prop, 1) // C D
```

Parameters

- **node_id** – node identifier
- **prop** – lowercase-and-underscores property name
- **idx** – the index to get

Returns
the property's value as a sequence of tokens, with no quotes

DT_PROP_BY_PHANDLE_IDX(node_id, phs, idx, prop)

Get a property value from a phandle in a property.

This is a shorthand for:

```
DT_PROP(DT_PHANDLE_BY_IDX(node_id, phs, idx), prop)
```

That is, `prop` is a property of the phandle's node, not a property of `node_id`.

Example devicetree fragment:

```
n1: node-1 {
        foo = <&n2 &n3>;
};

n2: node-2 {
        bar = <42>;
};

n3: node-3 {
        baz = <43>;
};
```

Example usage:

```
#define N1 DT_NODELABEL(n1)

DT_PROP_BY_PHANDLE_IDX(N1, foo, 0, bar) // 42
DT_PROP_BY_PHANDLE_IDX(N1, foo, 1, baz) // 43
```

Parameters

- **node_id** – node identifier
- **phs** – lowercase-and-underscores property with type `phandle`, `phandles`, or `phandle-array`
- **idx** – logical index into `phs`, which must be zero if `phs` has type `phandle`
- **prop** – lowercase-and-underscores property of the phandle's node

Returns
the property's value

`DT_PROP_BY_PHANDLE_IDX_OR`(node_id, phs, idx, prop, default_value)

Like *DT_PROP_BY_PHANDLE_IDX()*, but with a fallback to `default_value`.

If the value exists, this expands to *DT_PROP_BY_PHANDLE_IDX(node_id, phs,idx, prop)*. The `default_value` parameter is not expanded in this case.

Otherwise, this expands to `default_value`.

Parameters

- **node_id** – node identifier
- **phs** – lowercase-and-underscores property with type `phandle`, `phandles`, or `phandle-array`
- **idx** – logical index into `phs`, which must be zero if `phs` has type `phandle`
- **prop** – lowercase-and-underscores property of the phandle's node
- **default_value** – a fallback value to expand to

Returns
the property's value

`DT_PROP_BY_PHANDLE`(node_id, ph, prop)

Get a property value from a phandle's node.

This is equivalent to *DT_PROP_BY_PHANDLE_IDX(node_id, ph, 0, prop)*.

Parameters

- **node_id** – node identifier
- **ph** – lowercase-and-underscores property of `node_id` with type `phandle`
- **prop** – lowercase-and-underscores property of the phandle's node

Returns
the property's value

`DT_PHA_BY_IDX`(node_id, pha, idx, cell)

Get a phandle-array specifier cell value at an index.

It might help to read the argument order as being similar to `node->phandle_array[index].cell`. That is, the cell value is in the `pha` property of `node_id`, inside the specifier at index `idx`.

Example devicetree fragment:

```
gpio0: gpio@abcd1234 {
        #gpio-cells = <2>;
};

gpio1: gpio@1234abcd {
        #gpio-cells = <2>;
};

led: led_0 {
        gpios = <&gpio0 17 0x1>, <&gpio1 5 0x3>;
};
```

Bindings fragment for the `gpio0` and `gpio1` nodes:

```
gpio-cells:
  - pin
  - flags
```

Above, gpios has two elements:

- index 0 has specifier <17 0x1>, so its pin cell is 17, and its flags cell is 0x1
- index 1 has specifier <5 0x3>, so pin is 5 and flags is 0x3

Example usage:

```
#define LED DT_NODELABEL(led)

DT_PHA_BY_IDX(LED, gpios, 0, pin)   // 17
DT_PHA_BY_IDX(LED, gpios, 1, flags) // 0x3
```

Parameters

- **node_id** – node identifier
- **pha** – lowercase-and-underscores property with type phandle-array
- **idx** – logical index into pha
- **cell** – lowercase-and-underscores cell name within the specifier at pha index idx

Returns
the cell's value

DT_PHA_BY_IDX_OR(node_id, pha, idx, cell, default_value)

Like *DT_PHA_BY_IDX()*, but with a fallback to default_value.

If the value exists, this expands to *DT_PHA_BY_IDX(node_id, pha,idx, cell)*. The default_value parameter is not expanded in this case.

Otherwise, this expands to default_value.

Parameters

- **node_id** – node identifier
- **pha** – lowercase-and-underscores property with type phandle-array
- **idx** – logical index into pha
- **cell** – lowercase-and-underscores cell name within the specifier at pha index idx
- **default_value** – a fallback value to expand to

Returns
the cell's value or default_value

DT_PHA(node_id, pha, cell)

Equivalent to *DT_PHA_BY_IDX(node_id, pha, 0, cell)*

Parameters

- **node_id** – node identifier
- **pha** – lowercase-and-underscores property with type phandle-array
- **cell** – lowercase-and-underscores cell name

Returns
the cell's value

DT_PHA_OR(node_id, pha, cell, default_value)

Like *DT_PHA()*, but with a fallback to `default_value`.

If the value exists, this expands to *DT_PHA(node_id, pha, cell)*. The `default_value` parameter is not expanded in this case.

Otherwise, this expands to `default_value`.

Parameters

- **node_id** – node identifier
- **pha** – lowercase-and-underscores property with type `phandle-array`
- **cell** – lowercase-and-underscores cell name
- **default_value** – a fallback value to expand to

Returns
the cell's value or `default_value`

DT_PHA_BY_NAME(node_id, pha, name, cell)

Get a value within a phandle-array specifier by name.

This is like *DT_PHA_BY_IDX()*, except it treats `pha` as a structure where each array element has a name.

It might help to read the argument order as being similar to `node->phandle_struct.name.cell`. That is, the cell value is in the `pha` property of `node_id`, treated as a data structure where each array element has a name.

Example devicetree fragment:

```
n: node {
        io-channels = <&adc1 10>, <&adc2 20>;
        io-channel-names = "SENSOR", "BANDGAP";
};
```

Bindings fragment for the "adc1" and "adc2" nodes:

```
io-channel-cells:
  - input
```

Example usage:

```
DT_PHA_BY_NAME(DT_NODELABEL(n), io_channels, sensor, input)  // 10
DT_PHA_BY_NAME(DT_NODELABEL(n), io_channels, bandgap, input) // 20
```

Parameters

- **node_id** – node identifier
- **pha** – lowercase-and-underscores property with type `phandle-array`
- **name** – lowercase-and-underscores name of a specifier in `pha`
- **cell** – lowercase-and-underscores cell name in the named specifier

Returns
the cell's value

DT_PHA_BY_NAME_OR(node_id, pha, name, cell, default_value)

Like *DT_PHA_BY_NAME()*, but with a fallback to `default_value`.

If the value exists, this expands to *DT_PHA_BY_NAME(node_id, pha,name, cell)*. The `default_value` parameter is not expanded in this case.

Otherwise, this expands to `default_value`.

Parameters

- **node_id** – node identifier
- **pha** – lowercase-and-underscores property with type `phandle-array`
- **name** – lowercase-and-underscores name of a specifier in `pha`
- **cell** – lowercase-and-underscores cell name in the named specifier
- **default_value** – a fallback value to expand to

Returns

the cell's value or `default_value`

DT_PHANDLE_BY_NAME(node_id, pha, name)

Get a phandle's node identifier from a phandle array by `name`.

It might help to read the argument order as being similar to `node->phandle_struct.name.phandle`. That is, the phandle array is treated as a structure with named elements. The return value is the node identifier for a phandle inside the structure.

Example devicetree fragment:

```
adc1: adc@abcd1234 {
        foobar = "ADC_1";
};

adc2: adc@1234abcd {
        foobar = "ADC_2";
};

n: node {
        io-channels = <&adc1 10>, <&adc2 20>;
        io-channel-names = "SENSOR", "BANDGAP";
};
```

Above, "io-channels" has two elements:

- the element named `"SENSOR"` has phandle `&adc1`
- the element named `"BANDGAP"` has phandle `&adc2`

Example usage:

```
#define NODE DT_NODELABEL(n)

DT_PROP(DT_PHANDLE_BY_NAME(NODE, io_channels, sensor), foobar)  // "ADC_1"
DT_PROP(DT_PHANDLE_BY_NAME(NODE, io_channels, bandgap), foobar) // "ADC_2"
```

Notice how devicetree properties and names are lowercased, and non-alphanumeric characters are converted to underscores.

Parameters

- **node_id** – node identifier
- **pha** – lowercase-and-underscores property with type `phandle-array`
- **name** – lowercase-and-underscores name of an element in `pha`

Returns
a node identifier for the node with that phandle

`DT_PHANDLE_BY_IDX`(node_id, prop, idx)

Get a node identifier for a phandle in a property.

When a node's value at a logical index contains a phandle, this macro returns a node identifier for the node with that phandle.

Therefore, if `prop` has type `phandle`, `idx` must be zero. (A `phandle` type is treated as a `phandles` with a fixed length of 1).

Example devicetree fragment:

```
n1: node-1 {
        foo = <&n2 &n3>;
};

n2: node-2 { ... };
n3: node-3 { ... };
```

Above, `foo` has type phandles and has two elements:

- index 0 has phandle `&n2`, which is `node-2`'s phandle
- index 1 has phandle `&n3`, which is `node-3`'s phandle

Example usage:

```
#define N1 DT_NODELABEL(n1)

DT_PHANDLE_BY_IDX(N1, foo, 0) // node identifier for node-2
DT_PHANDLE_BY_IDX(N1, foo, 1) // node identifier for node-3
```

Behavior is analogous for phandle-arrays.

Parameters

- **node_id** – node identifier
- **prop** – lowercase-and-underscores property name in `node_id` with type `phandle`, `phandles` or `phandle-array`
- **idx** – index into `prop`

Returns
node identifier for the node with the phandle at that index

`DT_PHANDLE`(node_id, prop)

Get a node identifier for a phandle property's value.

This is equivalent to *DT_PHANDLE_BY_IDX(node_id, prop, 0)*. Its primary benefit is readability when `prop` has type `phandle`.

Parameters

- **node_id** – node identifier
- **prop** – lowercase-and-underscores property of `node_id` with type `phandle`

Returns
a node identifier for the node pointed to by "ph"

ranges property Use these APIs instead of *Property access* to access the ranges property. Because this property's semantics are defined by the devicetree specification, these macros can be used even for nodes without matching bindings. However, they take on special semantics when the node's binding indicates it is a PCIe bus node, as defined in the PCI Bus Binding to: IEEE Std 1275-1994 Standard for Boot (Initialization Configuration) Firmware

group devicetree-ranges-prop

Defines

DT_NUM_RANGES(node_id)

Get the number of range blocks in the ranges property.

Use this instead of *DT_PROP_LEN(node_id, ranges)*.

Example devicetree fragment:

```
pcie0: pcie@0 {
        compatible = "pcie-controller";
        reg = <0 1>;
        #address-cells = <3>;
        #size-cells = <2>;

        ranges = <0x1000000 0 0 0 0x3eff0000 0 0x10000>,
                 <0x2000000 0 0x10000000 0 0x10000000 0 0x2eff0000>,
                 <0x3000000 0x80 0 0x80 0 0x80 0>;
};

other: other@1 {
        reg = <1 1>;

        ranges = <0x0 0x0 0x0 0x3eff0000 0x10000>,
                 <0x0 0x10000000 0x0 0x10000000 0x2eff0000>;
};
```

Example usage:

```
DT_NUM_RANGES(DT_NODELABEL(pcie0)) // 3
DT_NUM_RANGES(DT_NODELABEL(other)) // 2
```

Parameters

- **node_id** – node identifier

DT_RANGES_HAS_IDX(node_id, idx)

Is idx a valid range block index?

If this returns 1, then *DT_RANGES_CHILD_BUS_ADDRESS_BY_IDX(node_id, idx)*, *DT_RANGES_PARENT_BUS_ADDRESS_BY_IDX(node_id, idx)* or *DT_RANGES_LENGTH_BY_IDX(node_id, idx)* are valid. For *DT_RANGES_CHILD_BUS_FLAGS_BY_IDX(node_id, idx)* the return value of *DT_RANGES_HAS_CHILD_BUS_FLAGS_AT_IDX(node_id, idx)* will indicate validity. If it returns 0, it is an error to use those macros with index idx, including *DT_RANGES_CHILD_BUS_FLAGS_BY_IDX(node_id, idx)*.

Example devicetree fragment:

```
pcie0: pcie@0 {
        compatible = "pcie-controller";
        reg = <0 1>;
```

(continues on next page)

(continued from previous page)

```
        #address-cells = <3>;
        #size-cells = <2>;

        ranges = <0x1000000 0 0 0 0x3eff0000 0 0x10000>,
                 <0x2000000 0 0x10000000 0 0x10000000 0 0x2eff0000>,
                 <0x3000000 0x80 0 0x80 0 0x80 0>;
};

other: other@1 {
        reg = <1 1>;

        ranges = <0x0 0x0 0x0 0x3eff0000 0x10000>,
                 <0x0 0x10000000 0x0 0x10000000 0x2eff0000>;
};
```

Example usage:

```
DT_RANGES_HAS_IDX(DT_NODELABEL(pcie0), 0) // 1
DT_RANGES_HAS_IDX(DT_NODELABEL(pcie0), 1) // 1
DT_RANGES_HAS_IDX(DT_NODELABEL(pcie0), 2) // 1
DT_RANGES_HAS_IDX(DT_NODELABEL(pcie0), 3) // 0
DT_RANGES_HAS_IDX(DT_NODELABEL(other), 0) // 1
DT_RANGES_HAS_IDX(DT_NODELABEL(other), 1) // 1
DT_RANGES_HAS_IDX(DT_NODELABEL(other), 2) // 0
DT_RANGES_HAS_IDX(DT_NODELABEL(other), 3) // 0
```

Parameters

- **node_id** – node identifier
- **idx** – index to check

Returns

1 if `idx` is a valid register block index, 0 otherwise.

`DT_RANGES_HAS_CHILD_BUS_FLAGS_AT_IDX`(node_id, idx)

Does a ranges property have child bus flags at index?

If this returns 1, then *DT_RANGES_CHILD_BUS_FLAGS_BY_IDX(node_id, idx)* is valid. If it returns 0, it is an error to use this macro with index `idx`. This macro only returns 1 for PCIe buses (i.e. nodes whose bindings specify they are “pcie” bus nodes.)

Example devicetree fragment:

```
parent {
        #address-cells = <2>;

        pcie0: pcie@0 {
                compatible = "pcie-controller";
                reg = <0 0 1>;
                #address-cells = <3>;
                #size-cells = <2>;

                ranges = <0x1000000 0 0 0 0x3eff0000 0 0x10000>,
                         <0x2000000 0 0x10000000 0 0x10000000 0 0x2eff0000>,
                         <0x3000000 0x80 0 0x80 0 0x80 0>;
        };

        other: other@1 {
                reg = <0 1 1>;

                ranges = <0x0 0x0 0x0 0x3eff0000 0x10000>,
```

(continues on next page)

(continued from previous page)

```
                        <0x0 0x10000000 0x0 0x10000000 0x2eff0000>;
        };
};
```

Example usage:

```
DT_RANGES_HAS_CHILD_BUS_FLAGS_AT_IDX(DT_NODELABEL(pcie0), 0) // 1
DT_RANGES_HAS_CHILD_BUS_FLAGS_AT_IDX(DT_NODELABEL(pcie0), 1) // 1
DT_RANGES_HAS_CHILD_BUS_FLAGS_AT_IDX(DT_NODELABEL(pcie0), 2) // 1
DT_RANGES_HAS_CHILD_BUS_FLAGS_AT_IDX(DT_NODELABEL(pcie0), 3) // 0
DT_RANGES_HAS_CHILD_BUS_FLAGS_AT_IDX(DT_NODELABEL(other), 0) // 0
DT_RANGES_HAS_CHILD_BUS_FLAGS_AT_IDX(DT_NODELABEL(other), 1) // 0
DT_RANGES_HAS_CHILD_BUS_FLAGS_AT_IDX(DT_NODELABEL(other), 2) // 0
DT_RANGES_HAS_CHILD_BUS_FLAGS_AT_IDX(DT_NODELABEL(other), 3) // 0
```

Parameters

- **node_id** – node identifier
- **idx** – logical index into the ranges array

Returns

1 if idx is a valid child bus flags index, 0 otherwise.

DT_RANGES_CHILD_BUS_FLAGS_BY_IDX(node_id, idx)

Get the ranges property child bus flags at index.

When the node is a PCIe bus, the Child Bus Address has an extra cell used to store some flags, thus this cell is extracted from the Child Bus Address as Child Bus Flags field.

Example devicetree fragments:

```
parent {
        #address-cells = <2>;

        pcie0: pcie@0 {
                compatible = "pcie-controller";
                reg = <0 0 1>;
                #address-cells = <3>;
                #size-cells = <2>;

                ranges = <0x1000000 0 0 0 0x3eff0000 0 0x10000>,
                         <0x2000000 0 0x10000000 0 0x10000000 0 0x2eff0000>,
                         <0x3000000 0x80 0 0x80 0 0x80 0>;
        };
};
```

Example usage:

```
DT_RANGES_CHILD_BUS_FLAGS_BY_IDX(DT_NODELABEL(pcie0), 0) // 0x1000000
DT_RANGES_CHILD_BUS_FLAGS_BY_IDX(DT_NODELABEL(pcie0), 1) // 0x2000000
DT_RANGES_CHILD_BUS_FLAGS_BY_IDX(DT_NODELABEL(pcie0), 2) // 0x3000000
```

Parameters

- **node_id** – node identifier
- **idx** – logical index into the ranges array

Returns

range child bus flags field at idx

DT_RANGES_CHILD_BUS_ADDRESS_BY_IDX(node_id, idx)

Get the ranges property child bus address at index.

When the node is a PCIe bus, the Child Bus Address has an extra cell used to store some flags, thus this cell is removed from the Child Bus Address.

Example devicetree fragments:

```
parent {
        #address-cells = <2>;

        pcie0: pcie@0 {
                compatible = "pcie-controller";
                reg = <0 0 1>;
                #address-cells = <3>;
                #size-cells = <2>;

                ranges = <0x1000000 0 0 0 0x3eff0000 0 0x10000>,
                         <0x2000000 0 0x10000000 0 0x10000000 0 0x2eff0000>,
                         <0x3000000 0x80 0 0x80 0 0x80 0>;
        };

        other: other@1 {
                reg = <0 1 1>;

                ranges = <0x0 0x0 0x0 0x3eff0000 0x10000>,
                         <0x0 0x10000000 0x0 0x10000000 0x2eff0000>;
        };
};
```

Example usage:

```
DT_RANGES_CHILD_BUS_ADDRESS_BY_IDX(DT_NODELABEL(pcie0), 0) // 0
DT_RANGES_CHILD_BUS_ADDRESS_BY_IDX(DT_NODELABEL(pcie0), 1) // 0x10000000
DT_RANGES_CHILD_BUS_ADDRESS_BY_IDX(DT_NODELABEL(pcie0), 2) // 0x8000000000
DT_RANGES_CHILD_BUS_ADDRESS_BY_IDX(DT_NODELABEL(other), 0) // 0
DT_RANGES_CHILD_BUS_ADDRESS_BY_IDX(DT_NODELABEL(other), 1) // 0x10000000
```

Parameters

- **node_id** – node identifier
- **idx** – logical index into the ranges array

Returns

range child bus address field at idx

DT_RANGES_PARENT_BUS_ADDRESS_BY_IDX(node_id, idx)

Get the ranges property parent bus address at index.

Similarly to *DT_RANGES_CHILD_BUS_ADDRESS_BY_IDX()*, this properly accounts for child bus flags cells when the node is a PCIe bus.

Example devicetree fragment:

```
parent {
        #address-cells = <2>;

        pcie0: pcie@0 {
                compatible = "pcie-controller";
                reg = <0 0 1>;
                #address-cells = <3>;
                #size-cells = <2>;
```

(continues on next page)

(continued from previous page)

```
                ranges = <0x1000000 0 0 0 0x3eff0000 0 0x10000>,
                         <0x2000000 0 0x10000000 0 0x10000000 0 0x2eff0000>,
                         <0x3000000 0x80 0 0x80 0 0x80 0>;
        };

        other: other@1 {
                reg = <0 1 1>;

                ranges = <0x0 0x0 0x0 0x3eff0000 0x10000>,
                         <0x0 0x10000000 0x0 0x10000000 0x2eff0000>;
        };
};
```

Example usage:

```
DT_RANGES_PARENT_BUS_ADDRESS_BY_IDX(DT_NODELABEL(pcie0), 0) // 0x3eff0000
DT_RANGES_PARENT_BUS_ADDRESS_BY_IDX(DT_NODELABEL(pcie0), 1) // 0x10000000
DT_RANGES_PARENT_BUS_ADDRESS_BY_IDX(DT_NODELABEL(pcie0), 2) // 0x8000000000
DT_RANGES_PARENT_BUS_ADDRESS_BY_IDX(DT_NODELABEL(other), 0) // 0x3eff0000
DT_RANGES_PARENT_BUS_ADDRESS_BY_IDX(DT_NODELABEL(other), 1) // 0x10000000
```

Parameters

- `node_id` – node identifier
- `idx` – logical index into the ranges array

Returns
range parent bus address field at idx

`DT_RANGES_LENGTH_BY_IDX`(node_id, idx)

Get the ranges property length at index.

Similarly to *DT_RANGES_CHILD_BUS_ADDRESS_BY_IDX()*, this properly accounts for child bus flags cells when the node is a PCIe bus.

Example devicetree fragment:

```
parent {
        #address-cells = <2>;

        pcie0: pcie@0 {
                compatible = "pcie-controller";
                reg = <0 0 1>;
                #address-cells = <3>;
                #size-cells = <2>;

                ranges = <0x1000000 0 0 0 0x3eff0000 0 0x10000>,
                         <0x2000000 0 0x10000000 0 0x10000000 0 0x2eff0000>,
                         <0x3000000 0x80 0 0x80 0 0x80 0>;
        };

        other: other@1 {
                reg = <0 1 1>;

                ranges = <0x0 0x0 0x0 0x3eff0000 0x10000>,
                         <0x0 0x10000000 0x0 0x10000000 0x2eff0000>;
        };
};
```

Example usage:

```
DT_RANGES_LENGTH_BY_IDX(DT_NODELABEL(pcie0), 0) // 0x10000
DT_RANGES_LENGTH_BY_IDX(DT_NODELABEL(pcie0), 1) // 0x2eff0000
DT_RANGES_LENGTH_BY_IDX(DT_NODELABEL(pcie0), 2) // 0x8000000000
DT_RANGES_LENGTH_BY_IDX(DT_NODELABEL(other), 0) // 0x10000
DT_RANGES_LENGTH_BY_IDX(DT_NODELABEL(other), 1) // 0x2eff0000
```

Parameters

- **node_id** – node identifier
- **idx** – logical index into the ranges array

Returns
range length field at idx

DT_FOREACH_RANGE(node_id, fn)

Invokes `fn` for each entry of `node_id` ranges property.

The macro `fn` must take two parameters, `node_id` which will be the node identifier of the node with the ranges property and `idx` the index of the ranges block.

Example devicetree fragment:

```
n: node@0 {
        reg = <0 0 1>;

        ranges = <0x0 0x0 0x0 0x3eff0000 0x10000>,
                 <0x0 0x10000000 0x0 0x10000000 0x2eff0000>;
};
```

Example usage:

```
#define RANGE_LENGTH(node_id, idx) DT_RANGES_LENGTH_BY_IDX(node_id, idx),

const uint64_t *ranges_length[] = {
        DT_FOREACH_RANGE(DT_NODELABEL(n), RANGE_LENGTH)
};
```

This expands to:

```
const char *ranges_length[] = {
    0x10000, 0x2eff0000,
};
```

Parameters

- **node_id** – node identifier
- **fn** – macro to invoke

reg property Use these APIs instead of *Property access* to access the `reg` property. Because this property's semantics are defined by the devicetree specification, these macros can be used even for nodes without matching bindings.

group devicetree-reg-prop

Defines

DT_NUM_REGS(node_id)

Get the number of register blocks in the reg property.

Use this instead of *DT_PROP_LEN(node_id, reg)*.

Parameters

- **node_id** – node identifier

Returns
Number of register blocks in the node's "reg" property.

DT_REG_HAS_IDX(node_id, idx)

Is idx a valid register block index?

If this returns 1, then *DT_REG_ADDR_BY_IDX(node_id, idx)* or *DT_REG_SIZE_BY_IDX(node_id, idx)* are valid. If it returns 0, it is an error to use those macros with index idx.

Parameters

- **node_id** – node identifier
- **idx** – index to check

Returns
1 if idx is a valid register block index, 0 otherwise.

DT_REG_HAS_NAME(node_id, name)

Is name a valid register block name?

If this returns 1, then *DT_REG_ADDR_BY_NAME(node_id, name)* or *DT_REG_SIZE_BY_NAME(node_id, name)* are valid. If it returns 0, it is an error to use those macros with name name.

Parameters

- **node_id** – node identifier
- **name** – name to check

Returns
1 if name is a valid register block name, 0 otherwise.

DT_REG_ADDR_BY_IDX(node_id, idx)

Get the base address of the register block at index idx.

Parameters

- **node_id** – node identifier
- **idx** – index of the register whose address to return

Returns
address of the idx-th register block

DT_REG_SIZE_BY_IDX(node_id, idx)

Get the size of the register block at index idx.

This is the size of an individual register block, not the total number of register blocks in the property; use *DT_NUM_REGS()* for that.

Parameters

- **node_id** – node identifier
- **idx** – index of the register whose size to return

Returns
size of the idx-th register block

DT_REG_ADDR(node_id)

Get a node's (only) register block address.

Equivalent to *DT_REG_ADDR_BY_IDX(node_id, 0)*.

Parameters

- **node_id** – node identifier

Returns
node's register block address

DT_REG_ADDR_U64(node_id)

64-bit version of *DT_REG_ADDR()*

This macro version adds the appropriate suffix for 64-bit unsigned integer literals. Note that this macro is equivalent to *DT_REG_ADDR()* in linker/ASM context.

Parameters

- **node_id** – node identifier

Returns
node's register block address

DT_REG_SIZE(node_id)

Get a node's (only) register block size.

Equivalent to *DT_REG_SIZE_BY_IDX(node_id, 0)*.

Parameters

- **node_id** – node identifier

Returns
node's only register block's size

DT_REG_ADDR_BY_NAME(node_id, name)

Get a register block's base address by name.

Parameters

- **node_id** – node identifier
- **name** – lowercase-and-underscores register specifier name

Returns
address of the register block specified by name

DT_REG_ADDR_BY_NAME_OR(node_id, name, default_value)

Like *DT_REG_ADDR_BY_NAME()*, but with a fallback to `default_value`.

Parameters

- **node_id** – node identifier
- **name** – lowercase-and-underscores register specifier name
- **default_value** – a fallback value to expand to

Returns
address of the register block specified by name if present, `default_value` otherwise

DT_REG_ADDR_BY_NAME_U64(node_id, name)

64-bit version of *DT_REG_ADDR_BY_NAME()*

This macro version adds the appropriate suffix for 64-bit unsigned integer literals. Note that this macro is equivalent to *DT_REG_ADDR_BY_NAME()* in linker/ASM context.

Parameters

- `node_id` – node identifier
- `name` – lowercase-and-underscores register specifier name

Returns
address of the register block specified by name

`DT_REG_SIZE_BY_NAME`(node_id, name)

Get a register block's size by name.

Parameters

- `node_id` – node identifier
- `name` – lowercase-and-underscores register specifier name

Returns
size of the register block specified by name

`DT_REG_SIZE_BY_NAME_OR`(node_id, name, default_value)

Like *DT_REG_SIZE_BY_NAME()*, but with a fallback to `default_value`.

Parameters

- `node_id` – node identifier
- `name` – lowercase-and-underscores register specifier name
- `default_value` – a fallback value to expand to

Returns
size of the register block specified by name if present, `default_value` otherwise

`interrupts` property Use these APIs instead of *Property access* to access the `interrupts` property.

Because this property's semantics are defined by the devicetree specification, some of these macros can be used even for nodes without matching bindings. This does not apply to macros which take cell names as arguments.

group `devicetree-interrupts-prop`

Defines

`DT_NUM_IRQS`(node_id)

Get the number of interrupt sources for the node.

Use this instead of *DT_PROP_LEN(node_id, interrupts)*.

Parameters

- `node_id` – node identifier

Returns
Number of interrupt specifiers in the node's "interrupts" property.

`DT_NUM_NODELABELS`(node_id)

Get the number of node labels that a node has.

Example devicetree fragment:

```
/ {
    foo {};
    bar: bar@1000 {};
    baz: baz2: baz@2000 {};
};
```

Example usage:

```
DT_NUM_NODELABELS(DT_PATH(foo))       // 0
DT_NUM_NODELABELS(DT_NODELABEL(bar)) // 1
DT_NUM_NODELABELS(DT_NODELABEL(baz)) // 2
```

Parameters

- **node_id** – node identifier

Returns
number of node labels that the node has

`DT_IRQ_LEVEL`(node_id)

Get the interrupt level for the node.

Parameters

- **node_id** – node identifier

Returns
interrupt level

`DT_IRQ_HAS_IDX`(node_id, idx)

Is `idx` a valid interrupt index?

If this returns 1, then *DT_IRQ_BY_IDX(node_id, idx)* is valid. If it returns 0, it is an error to use that macro with this index.

Parameters

- **node_id** – node identifier
- **idx** – index to check

Returns
1 if the idx is valid for the interrupt property 0 otherwise.

`DT_IRQ_HAS_CELL_AT_IDX`(node_id, idx, cell)

Does an interrupts property have a named cell specifier at an index? If this returns 1, then *DT_IRQ_BY_IDX(node_id, idx, cell)* is valid.

If it returns 0, it is an error to use that macro.

Parameters

- **node_id** – node identifier
- **idx** – index to check
- **cell** – named cell value whose existence to check

Returns
1 if the named cell exists in the interrupt specifier at index idx 0 otherwise.

`DT_IRQ_HAS_CELL`(node_id, cell)

Equivalent to *DT_IRQ_HAS_CELL_AT_IDX(node_id, 0, cell)*

Parameters

- **node_id** – node identifier

- **cell** – named cell value whose existence to check

Returns
1 if the named cell exists in the interrupt specifier at index 0 0 otherwise.

DT_IRQ_HAS_NAME(node_id, name)

Does an interrupts property have a named specifier value at an index? If this returns 1, then *DT_IRQ_BY_NAME(node_id, name, cell)* is valid.

If it returns 0, it is an error to use that macro.

Parameters

- **node_id** – node identifier
- **name** – lowercase-and-underscores interrupt specifier name

Returns
1 if "name" is a valid named specifier 0 otherwise.

DT_IRQ_BY_IDX(node_id, idx, cell)

Get a value within an interrupt specifier at an index.

It might help to read the argument order as being similar to "node->interrupts[index].cell".

This can be used to get information about an individual interrupt when a device generates more than one.

Example devicetree fragment:

```
my-serial: serial@abcd1234 {
        interrupts = < 33 0 >, < 34 1 >;
};
```

Assuming the node's interrupt domain has "#interrupt-cells = <2>;" and the individual cells in each interrupt specifier are named "irq" and "priority" by the node's binding, here are some examples:

```
#define SERIAL DT_NODELABEL(my_serial)

Example usage                        Value
-------------                        -----
DT_IRQ_BY_IDX(SERIAL, 0, irq)           33
DT_IRQ_BY_IDX(SERIAL, 0, priority)       0
DT_IRQ_BY_IDX(SERIAL, 1, irq,           34
DT_IRQ_BY_IDX(SERIAL, 1, priority)       1
```

Parameters

- **node_id** – node identifier
- **idx** – logical index into the interrupt specifier array
- **cell** – cell name specifier

Returns
the named value at the specifier given by the index

DT_IRQ_BY_NAME(node_id, name, cell)

Get a value within an interrupt specifier by name.

It might help to read the argument order as being similar to `node->interrupts.name.cell`.

This can be used to get information about an individual interrupt when a device generates more than one, if the bindings give each interrupt specifier a name.

Parameters

- **node_id** – node identifier
- **name** – lowercase-and-underscores interrupt specifier name
- **cell** – cell name specifier

Returns
the named value at the specifier given by the index

DT_IRQ(node_id, cell)

Get an interrupt specifier's value Equivalent to *DT_IRQ_BY_IDX(node_id, 0, cell)*.

Parameters

- **node_id** – node identifier
- **cell** – cell name specifier

Returns
the named value at that index

DT_IRQ_INTC_BY_IDX(node_id, idx)

Get an interrupt specifier's interrupt controller by index.

```
gpio0: gpio0 {
        interrupt-controller;
        #interrupt-cells = <2>;
};

foo: foo {
        interrupt-parent = <&gpio0>;
        interrupts = <1 1>, <2 2>;
};

bar: bar {
        interrupts-extended = <&gpio0 3 3>, <&pic0 4>;
};

pic0: pic0 {
        interrupt-controller;
        #interrupt-cells = <1>;

        qux: qux {
                interrupts = <5>, <6>;
                interrupt-names = "int1", "int2";
        };
};
```

Example usage:

```
DT_IRQ_INTC_BY_IDX(DT_NODELABEL(foo), 0) // &gpio0
DT_IRQ_INTC_BY_IDX(DT_NODELABEL(foo), 1) // &gpio0
DT_IRQ_INTC_BY_IDX(DT_NODELABEL(bar), 0) // &gpio0
DT_IRQ_INTC_BY_IDX(DT_NODELABEL(bar), 1) // &pic0
DT_IRQ_INTC_BY_IDX(DT_NODELABEL(qux), 0) // &pic0
DT_IRQ_INTC_BY_IDX(DT_NODELABEL(qux), 1) // &pic0
```

Parameters

- **node_id** – node identifier
- **idx** – interrupt specifier's index

Returns
node_id of interrupt specifier's interrupt controller

DT_IRQ_INTC_BY_NAME(node_id, name)

Get an interrupt specifier's interrupt controller by name.

```
gpio0: gpio0 {
        interrupt-controller;
        #interrupt-cells = <2>;
};

foo: foo {
        interrupt-parent = <&gpio0>;
        interrupts = <1 1>, <2 2>;
        interrupt-names = "int1", "int2";
};

bar: bar {
        interrupts-extended = <&gpio0 3 3>, <&pic0 4>;
        interrupt-names = "int1", "int2";
};

pic0: pic0 {
        interrupt-controller;
        #interrupt-cells = <1>;

        qux: qux {
                interrupts = <5>, <6>;
                interrupt-names = "int1", "int2";
        };
};
```

Example usage:

```
DT_IRQ_INTC_BY_NAME(DT_NODELABEL(foo), int1) // &gpio0
DT_IRQ_INTC_BY_NAME(DT_NODELABEL(foo), int2) // &gpio0
DT_IRQ_INTC_BY_NAME(DT_NODELABEL(bar), int1) // &gpio0
DT_IRQ_INTC_BY_NAME(DT_NODELABEL(bar), int2) // &pic0
DT_IRQ_INTC_BY_NAME(DT_NODELABEL(qux), int1) // &pic0
DT_IRQ_INTC_BY_NAME(DT_NODELABEL(qux), int2) // &pic0
```

Parameters

- **node_id** – node identifier
- **name** – interrupt specifier's name

Returns

node_id of interrupt specifier's interrupt controller

DT_IRQ_INTC(node_id)

Get an interrupt specifier's interrupt controller.

```
gpio0: gpio0 {
        interrupt-controller;
        #interrupt-cells = <2>;
};

foo: foo {
        interrupt-parent = <&gpio0>;
        interrupts = <1 1>;
};

bar: bar {
```

(continues on next page)

(continued from previous page)

```
        interrupts-extended = <&gpio0 3 3>;
};

pic0: pic0 {
        interrupt-controller;
        #interrupt-cells = <1>;

        qux: qux {
                interrupts = <5>;
        };
};
```

Example usage:

```
DT_IRQ_INTC(DT_NODELABEL(foo)) // &gpio0
DT_IRQ_INTC(DT_NODELABEL(bar)) // &gpio0
DT_IRQ_INTC(DT_NODELABEL(qux)) // &pic0
```

See also

DT_IRQ_INTC_BY_IDX()

Note

Equivalent to *DT_IRQ_INTC_BY_IDX(node_id, 0)*

Parameters

- **node_id** – node identifier

Returns
node_id of interrupt specifier's interrupt controller

`DT_IRQN_BY_IDX`(node_id, idx)

Get the node's Zephyr interrupt number at index If `CONFIG_MULTI_LEVEL_INTERRUPTS` is enabled, the interrupt number at index will be multi-level encoded.

Parameters

- **node_id** – node identifier
- **idx** – logical index into the interrupt specifier array

Returns
the Zephyr interrupt number

`DT_IRQN`(node_id)

Get a node's (only) irq number.

Equivalent to *DT_IRQ(node_id, irq)*. This is provided as a convenience for the common case where a node generates exactly one interrupt, and the IRQ number is in a cell named `irq`.

Parameters

- **node_id** – node identifier

Returns
the interrupt number for the node's only interrupt

For-each macros There is currently only one "generic" for-each macro, *DT_FOREACH_CHILD()*, which allows iterating over the children of a devicetree node.

There are special-purpose for-each macros, like *DT_INST_FOREACH_STATUS_OKAY()*, but these require DT_DRV_COMPAT to be defined before use.

group devicetree-generic-foreach

Defines

DT_FOREACH_NODE(fn)

Invokes fn for every node in the tree.

The macro fn must take one parameter, which will be a node identifier. The macro is expanded once for each node in the tree. The order that nodes are visited in is not specified.

Parameters

- fn – macro to invoke

DT_FOREACH_NODE_VARGS(fn, ...)

Invokes fn for every node in the tree with multiple arguments.

The macro fn takes multiple arguments. The first should be the node identifier for the node. The remaining are passed-in by the caller.

The macro is expanded once for each node in the tree. The order that nodes are visited in is not specified.

Parameters

- fn – macro to invoke
- ... – variable number of arguments to pass to fn

DT_FOREACH_STATUS_OKAY_NODE(fn)

Invokes fn for every status okay node in the tree.

The macro fn must take one parameter, which will be a node identifier. The macro is expanded once for each node in the tree with status okay (as usual, a missing status property is treated as status okay). The order that nodes are visited in is not specified.

Parameters

- fn – macro to invoke

DT_FOREACH_STATUS_OKAY_NODE_VARGS(fn, ...)

Invokes fn for every status okay node in the tree with multiple arguments.

The macro fn takes multiple arguments. The first should be the node identifier for the node. The remaining are passed-in by the caller.

The macro is expanded once for each node in the tree with status okay (as usual, a missing status property is treated as status okay). The order that nodes are visited in is not specified.

Parameters

- fn – macro to invoke
- ... – variable number of arguments to pass to fn

DT_FOREACH_CHILD(node_id, fn)

Invokes fn for each child of node_id.

The macro fn must take one parameter, which will be the node identifier of a child node of node_id.

The children will be iterated over in the same order as they appear in the final devicetree.

Example devicetree fragment:

```
n: node {
        child-1 {
                foobar = "foo";
        };
        child-2 {
                foobar = "bar";
        };
};
```

Example usage:

```
#define FOOBAR_AND_COMMA(node_id) DT_PROP(node_id, foobar),

const char *child_foobars[] = {
    DT_FOREACH_CHILD(DT_NODELABEL(n), FOOBAR_AND_COMMA)
};
```

This expands to:

```
const char *child_foobars[] = {
    "foo", "bar",
};
```

Parameters

- **node_id** – node identifier
- **fn** – macro to invoke

DT_FOREACH_CHILD_SEP(node_id, fn, sep)

Invokes fn for each child of node_id with a separator.

The macro fn must take one parameter, which will be the node identifier of a child node of node_id.

Example devicetree fragment:

```
n: node {
        child-1 {
                ...
        };
        child-2 {
                ...
        };
};
```

Example usage:

```
const char *child_names[] = {
    DT_FOREACH_CHILD_SEP(DT_NODELABEL(n), DT_NODE_FULL_NAME, (,))
};
```

This expands to:

```
const char *child_names[] = {
    "child-1", "child-2"
};
```

Parameters

- `node_id` – node identifier
- `fn` – macro to invoke
- `sep` – Separator (e.g. comma or semicolon). Must be in parentheses; this is required to enable providing a comma as separator.

`DT_FOREACH_CHILD_VARGS`(node_id, fn, ...)

Invokes `fn` for each child of `node_id` with multiple arguments.

The macro `fn` takes multiple arguments. The first should be the node identifier for the child node. The remaining are passed-in by the caller.

The children will be iterated over in the same order as they appear in the final devicetree.

See also

DT_FOREACH_CHILD

Parameters

- `node_id` – node identifier
- `fn` – macro to invoke
- `...` – variable number of arguments to pass to `fn`

`DT_FOREACH_CHILD_SEP_VARGS`(node_id, fn, sep, ...)

Invokes `fn` for each child of `node_id` with separator and multiple arguments.

The macro `fn` takes multiple arguments. The first should be the node identifier for the child node. The remaining are passed-in by the caller.

See also

DT_FOREACH_CHILD_VARGS

Parameters

- `node_id` – node identifier
- `fn` – macro to invoke
- `sep` – Separator (e.g. comma or semicolon). Must be in parentheses; this is required to enable providing a comma as separator.
- `...` – variable number of arguments to pass to `fn`

DT_FOREACH_CHILD_STATUS_OKAY(node_id, fn)

Call fn on the child nodes with status okay

The macro fn should take one argument, which is the node identifier for the child node.

As usual, both a missing status and an ok status are treated as okay.

The children will be iterated over in the same order as they appear in the final devicetree.

Parameters

- node_id – node identifier
- fn – macro to invoke

DT_FOREACH_CHILD_STATUS_OKAY_SEP(node_id, fn, sep)

Call fn on the child nodes with status okay with separator.

The macro fn should take one argument, which is the node identifier for the child node.

As usual, both a missing status and an ok status are treated as okay.

See also

DT_FOREACH_CHILD_STATUS_OKAY

Parameters

- node_id – node identifier
- fn – macro to invoke
- sep – Separator (e.g. comma or semicolon). Must be in parentheses; this is required to enable providing a comma as separator.

DT_FOREACH_CHILD_STATUS_OKAY_VARGS(node_id, fn, ...)

Call fn on the child nodes with status okay with multiple arguments.

The macro fn takes multiple arguments. The first should be the node identifier for the child node. The remaining are passed-in by the caller.

As usual, both a missing status and an ok status are treated as okay.

The children will be iterated over in the same order as they appear in the final devicetree.

See also

DT_FOREACH_CHILD_STATUS_OKAY

Parameters

- node_id – node identifier
- fn – macro to invoke
- ... – variable number of arguments to pass to fn

DT_FOREACH_CHILD_STATUS_OKAY_SEP_VARGS(node_id, fn, sep, ...)

Call fn on the child nodes with status okay with separator and multiple arguments.

The macro fn takes multiple arguments. The first should be the node identifier for the child node. The remaining are passed-in by the caller.

As usual, both a missing status and an ok status are treated as okay.

See also

DT_FOREACH_CHILD_SEP_STATUS_OKAY

Parameters

- **node_id** – node identifier
- **fn** – macro to invoke
- **sep** – Separator (e.g. comma or semicolon). Must be in parentheses; this is required to enable providing a comma as separator.
- **...** – variable number of arguments to pass to fn

DT_FOREACH_PROP_ELEM(node_id, prop, fn)

Invokes fn for each element in the value of property prop.

The macro fn must take three parameters: fn(node_id, prop, idx). node_id and prop are the same as what is passed to *DT_FOREACH_PROP_ELEM()*, and idx is the current index into the array. The idx values are integer literals starting from 0.

The prop argument must refer to a property that can be passed to *DT_PROP_LEN()*.

Example devicetree fragment:

```
n: node {
        my-ints = <1 2 3>;
};
```

Example usage:

```
#define TIMES_TWO(node_id, prop, idx) \
        (2 * DT_PROP_BY_IDX(node_id, prop, idx)),

int array[] = {
        DT_FOREACH_PROP_ELEM(DT_NODELABEL(n), my_ints, TIMES_TWO)
};
```

This expands to:

```
int array[] = {
        (2 * 1), (2 * 2), (2 * 3),
};
```

In general, this macro expands to:

```
fn(node_id, prop, 0) fn(node_id, prop, 1) [...] fn(node_id, prop, n-1)
```

where n is the number of elements in prop, as it would be returned by *DT_PROP_LEN(node_id, prop)*.

See also

DT_PROP_LEN

Parameters

- **node_id** – node identifier
- **prop** – lowercase-and-underscores property name
- **fn** – macro to invoke

DT_FOREACH_PROP_ELEM_SEP(node_id, prop, fn, sep)

Invokes fn for each element in the value of property prop with separator.

Example devicetree fragment:

```
n: node {
        my-gpios = <&gpioa 0 GPIO_ACTICE_HIGH>,
                   <&gpiob 1 GPIO_ACTIVE_HIGH>;
};
```

Example usage:

```
struct gpio_dt_spec specs[] = {
        DT_FOREACH_PROP_ELEM_SEP(DT_NODELABEL(n), my_gpios,
                                 GPIO_DT_SPEC_GET_BY_IDX, (,))
};
```

This expands as a first step to:

```
struct gpio_dt_spec specs[] = {
        GPIO_DT_SPEC_GET_BY_IDX(DT_NODELABEL(n), my_gpios, 0),
        GPIO_DT_SPEC_GET_BY_IDX(DT_NODELABEL(n), my_gpios, 1)
};
```

The prop parameter has the same restrictions as the same parameter given to *DT_FOREACH_PROP_ELEM()*.

See also

DT_FOREACH_PROP_ELEM

Parameters

- **node_id** – node identifier
- **prop** – lowercase-and-underscores property name
- **fn** – macro to invoke
- **sep** – Separator (e.g. comma or semicolon). Must be in parentheses; this is required to enable providing a comma as separator.

DT_FOREACH_PROP_ELEM_VARGS(node_id, prop, fn, ...)

Invokes fn for each element in the value of property prop with multiple arguments.

The macro fn must take multiple parameters: fn(node_id, prop, idx, ...). node_id and prop are the same as what is passed to *DT_FOREACH_PROP_ELEM()*, and idx is the

current index into the array. The `idx` values are integer literals starting from 0. The remaining arguments are passed-in by the caller.

The `prop` parameter has the same restrictions as the same parameter given to *DT_FOREACH_PROP_ELEM()*.

See also

DT_FOREACH_PROP_ELEM

Parameters

- **node_id** – node identifier
- **prop** – lowercase-and-underscores property name
- **fn** – macro to invoke
- **...** – variable number of arguments to pass to `fn`

`DT_FOREACH_PROP_ELEM_SEP_VARGS`(node_id, prop, fn, sep, ...)

Invokes `fn` for each element in the value of property `prop` with multiple arguments and a separator.

The `prop` parameter has the same restrictions as the same parameter given to *DT_FOREACH_PROP_ELEM()*.

See also

DT_FOREACH_PROP_ELEM_VARGS

Parameters

- **node_id** – node identifier
- **prop** – lowercase-and-underscores property name
- **fn** – macro to invoke
- **sep** – Separator (e.g. comma or semicolon). Must be in parentheses; this is required to enable providing a comma as separator.
- **...** – variable number of arguments to pass to fn

`DT_FOREACH_STATUS_OKAY`(compat, fn)

Invokes `fn` for each status `okay` node of a compatible.

This macro expands to:

```
fn(node_id_1) fn(node_id_2) ... fn(node_id_n)
```

where each `node_id_<i>` is a node identifier for some node with compatible `compat` and status `okay`. Whitespace is added between expansions as shown above.

Example devicetree fragment:

```
/ {
        a {
                compatible = "foo";
                status = "okay";
        };
        b {
                compatible = "foo";
                status = "disabled";
        };
        c {
                compatible = "foo";
        };
};
```

Example usage:

```
DT_FOREACH_STATUS_OKAY(foo, DT_NODE_PATH)
```

This expands to one of the following:

```
"/a" "/c"
"/c" "/a"
```

“One of the following” is because no guarantees are made about the order that node identifiers are passed to `fn` in the expansion.

(The `/c` string literal is present because a missing status property is always treated as if the status were set to `okay`.)

Note also that `fn` is responsible for adding commas, semicolons, or other terminators as needed.

Parameters

- **compat** – lowercase-and-underscores devicetree compatible
- **fn** – Macro to call for each enabled node. Must accept a node_id as its only parameter.

DT_FOREACH_STATUS_OKAY_VARGS(compat, fn, ...)

Invokes `fn` for each status `okay` node of a compatible with multiple arguments.

This is like *DT_FOREACH_STATUS_OKAY()* except you can also pass additional arguments to `fn`.

Example devicetree fragment:

```
/ {
        a {
                compatible = "foo";
                val = <3>;
        };
        b {
                compatible = "foo";
                val = <4>;
        };
};
```

Example usage:

```
#define MY_FN(node_id, operator) DT_PROP(node_id, val) operator
x = DT_FOREACH_STATUS_OKAY_VARGS(foo, MY_FN, +) 0;
```

This expands to one of the following:

```
x = 3 + 4 + 0;
x = 4 + 3 + 0;
```

i.e. it sets x to 7. As with *DT_FOREACH_STATUS_OKAY()*, there are no guarantees about the order nodes appear in the expansion.

Parameters

- **compat** – lowercase-and-underscores devicetree compatible
- **fn** – Macro to call for each enabled node. Must accept a node_id as its only parameter.
- **...** – Additional arguments to pass to fn

DT_FOREACH_NODELABEL(node_id, fn)

Invokes fn for each node label of a given node.

The order of the node labels in this macro's expansion matches the order in the final devicetree, with duplicates removed.

Node labels are passed to fn as tokens. Note that devicetree node labels are always valid C tokens (see "6.2 Labels" in Devicetree Specification v0.4 for details). The node labels are passed as tokens to fn as-is, without any lowercasing or conversion of special characters to underscores.

Example devicetree fragment:

```
foo: bar: FOO: node@deadbeef {};
```

Example usage:

```
int foo = 1;
int bar = 2;
int FOO = 3;

#define FN(nodelabel) + nodelabel
int sum = 0 DT_FOREACH_NODELABEL(DT_NODELABEL(foo), FN)
```

This expands to:

```
int sum = 0 + 1 + 2 + 3;
```

Parameters

- **node_id** – node identifier whose node labels to use
- **fn** – macro which will be passed each node label in order

DT_FOREACH_NODELABEL_VARGS(node_id, fn, ...)

Invokes fn for each node label of a given node with multiple arguments.

This is like *DT_FOREACH_NODELABEL()* except you can also pass additional arguments to fn.

Example devicetree fragment:

```
foo: bar: node@deadbeef {};
```

Example usage:

```
int foo = 0;
int bar = 1;
```

(continues on next page)

(continued from previous page)

```
#define VAR_PLUS(nodelabel, to_add) int nodelabel ## _added = nodelabel + to_add;

DT_FOREACH_NODELABEL_VARGS(DT_NODELABEL(foo), VAR_PLUS, 1)
```

This expands to:

```
int foo = 0;
int bar = 1;
int foo_added = foo + 1;
int bar_added = bar + 1;
```

Parameters

- `node_id` – node identifier whose node labels to use
- `fn` – macro which will be passed each node label in order
- `...` – additional arguments to pass to `fn`

Existence checks This section documents miscellaneous macros that can be used to test if a node exists, how many nodes of a certain type exist, whether a node has certain properties, etc. Some macros used for special purposes (such as *DT_IRQ_HAS_IDX()* and all macros which require `DT_DRV_COMPAT`) are documented elsewhere on this page.

Related code samples

GPIO with custom Devicetree binding
Use custom Devicetree binding to control a GPIO.

group `devicetree-generic-exist`

Defines

`DT_NODE_EXISTS`(node_id)

Does a node identifier refer to a node?

Tests whether a node identifier refers to a node which exists, i.e. is defined in the devicetree.

It doesn't matter whether or not the node has a matching binding, or what the node's status value is. This is purely a check of whether the node exists at all.

Parameters

- `node_id` – a node identifier

Returns
1 if the node identifier refers to a node, 0 otherwise.

`DT_NODE_HAS_STATUS`(node_id, status)

Does a node identifier refer to a node with a status?

Example uses:

```
DT_NODE_HAS_STATUS(DT_PATH(soc, i2c_12340000), okay)
DT_NODE_HAS_STATUS(DT_PATH(soc, i2c_12340000), disabled)
```

Tests whether a node identifier refers to a node which:

- exists in the devicetree, and
- has a status property matching the second argument (except that either a missing status or an ok status in the devicetree is treated as if it were okay instead)

Parameters

- **node_id** – a node identifier
- **status** – a status as one of the tokens okay or disabled, not a string

Returns
1 if the node has the given status, 0 otherwise.

DT_HAS_COMPAT_STATUS_OKAY(compat)

Does the devicetree have a status okay node with a compatible?

Test for whether the devicetree has any nodes with status okay and the given compatible. That is, this returns 1 if and only if there is at least one node_id for which both of these expressions return 1:

```
DT_NODE_HAS_STATUS(node_id, okay)
DT_NODE_HAS_COMPAT(node_id, compat)
```

As usual, both a missing status and an ok status are treated as okay.

Parameters

- **compat** – lowercase-and-underscores compatible, without quotes

Returns
1 if both of the above conditions are met, 0 otherwise

DT_NUM_INST_STATUS_OKAY(compat)

Get the number of instances of a given compatible with status okay

Parameters

- **compat** – lowercase-and-underscores compatible, without quotes

Returns
Number of instances with status okay

DT_NODE_HAS_COMPAT(node_id, compat)

Does a devicetree node match a compatible?

Example devicetree fragment:

```
n: node {
        compatible = "vnd,specific-device", "generic-device";
}
```

Example usages which evaluate to 1:

```
DT_NODE_HAS_COMPAT(DT_NODELABEL(n), vnd_specific_device)
DT_NODE_HAS_COMPAT(DT_NODELABEL(n), generic_device)
```

This macro only uses the value of the compatible property. Whether or not a particular compatible has a matching binding has no effect on its value, nor does the node's status.

Parameters

- **node_id** – node identifier

- **compat** – lowercase-and-underscores compatible, without quotes

Returns
1 if the node's compatible property contains `compat`, 0 otherwise.

`DT_NODE_HAS_COMPAT_STATUS`(node_id, compat, status)

Does a devicetree node have a compatible and status?

This is equivalent to:

```
(DT_NODE_HAS_COMPAT(node_id, compat) &&
 DT_NODE_HAS_STATUS(node_id, status))
```

Parameters

- **node_id** – node identifier
- **compat** – lowercase-and-underscores compatible, without quotes
- **status** – okay or disabled as a token, not a string

`DT_NODE_HAS_PROP`(node_id, prop)

Does a devicetree node have a property?

Tests whether a devicetree node has a property defined.

This tests whether the property is defined at all, not whether a boolean property is true or false. To get a boolean property's truth value, use *DT_PROP(node_id, prop)* instead.

Parameters

- **node_id** – node identifier
- **prop** – lowercase-and-underscores property name

Returns
1 if the node has the property, 0 otherwise.

`DT_PHA_HAS_CELL_AT_IDX`(node_id, pha, idx, cell)

Does a phandle array have a named cell specifier at an index?

If this returns 1, then the phandle-array property `pha` has a cell named `cell` at index `idx`, and therefore *DT_PHA_BY_IDX(node_id,pha, idx, cell)* is valid. If it returns 0, it's an error to use *DT_PHA_BY_IDX()* with the same arguments.

Parameters

- **node_id** – node identifier
- **pha** – lowercase-and-underscores property with type `phandle-array`
- **idx** – index to check within `pha`
- **cell** – lowercase-and-underscores cell name whose existence to check at index `idx`

Returns
1 if the named cell exists in the specifier at index idx, 0 otherwise.

`DT_PHA_HAS_CELL`(node_id, pha, cell)

Equivalent to *DT_PHA_HAS_CELL_AT_IDX(node_id, pha, 0, cell)*

Parameters

- **node_id** – node identifier
- **pha** – lowercase-and-underscores property with type `phandle-array`
- **cell** – lowercase-and-underscores cell name whose existence to check at index `idx`

Returns
1 if the named cell exists in the specifier at index 0, 0 otherwise.

Inter-node dependencies The `devicetree.h` API has some support for tracking dependencies between nodes. Dependency tracking relies on a binary "depends on" relation between devicetree nodes, which is defined as the transitive closure of the following "directly depends on" relation:

- every non-root node directly depends on its parent node
- a node directly depends on any nodes its properties refer to by phandle
- a node directly depends on its `interrupt-parent` if it has an `interrupts` property
- a parent node inherits all dependencies from its child nodes

A *dependency ordering* of a devicetree is a list of its nodes, where each node `n` appears earlier in the list than any nodes that depend on `n`. A node's *dependency ordinal* is then its zero-based index in that list. Thus, for two distinct devicetree nodes `n1` and `n2` with dependency ordinals `d1` and `d2`, we have:

- `d1 != d2`
- if `n1` depends on `n2`, then `d1 > d2`
- `d1 > d2` does **not** necessarily imply that `n1` depends on `n2`

The Zephyr build system chooses a dependency ordering of the final devicetree and assigns a dependency ordinal to each node. Dependency related information can be accessed using the following macros. The exact dependency ordering chosen is an implementation detail, but cyclic dependencies are detected and cause errors, so it's safe to assume there are none when using these macros.

There are instance number-based conveniences as well; see *DT_INST_DEP_ORD()* and subsequent documentation.

group `devicetree-dep-ord`

Defines

`DT_DEP_ORD`(node_id)
Get a node's dependency ordinal.

Parameters
- **node_id** – Node identifier

Returns
the node's dependency ordinal as an integer literal

`DT_DEP_ORD_STR_SORTABLE`(node_id)
Get a node's dependency ordinal in string sortable form.

Parameters
- **node_id** – Node identifier

Returns
the node's dependency ordinal as a zero-padded integer literal

`DT_REQUIRES_DEP_ORDS`(node_id)
Get a list of dependency ordinals of a node's direct dependencies.

There is a comma after each ordinal in the expansion, **including** the last one:

```
DT_REQUIRES_DEP_ORDS(my_node) // required_ord_1, ..., required_ord_n,
```

The one case *DT_REQUIRES_DEP_ORDS()* expands to nothing is when given the root node identifier `DT_ROOT` as argument. The root has no direct dependencies; every other node at least depends on its parent.

Parameters

- **node_id** – Node identifier

Returns
a list of dependency ordinals, with each ordinal followed by a comma (`,`), or an empty expansion

`DT_SUPPORTS_DEP_ORDS`(node_id)

Get a list of dependency ordinals of what depends directly on a node.

There is a comma after each ordinal in the expansion, **including** the last one:

```
DT_SUPPORTS_DEP_ORDS(my_node) // supported_ord_1, ..., supported_ord_n,
```

DT_SUPPORTS_DEP_ORDS() may expand to nothing. This happens when `node_id` refers to a leaf node that nothing else depends on.

Parameters

- **node_id** – Node identifier

Returns
a list of dependency ordinals, with each ordinal followed by a comma (`,`), or an empty expansion

`DT_INST_DEP_ORD`(inst)

Get a DT_DRV_COMPAT instance's dependency ordinal.

Equivalent to *DT_DEP_ORD(DT_DRV_INST(inst))*.

Parameters

- **inst** – instance number

Returns
The instance's dependency ordinal

`DT_INST_REQUIRES_DEP_ORDS`(inst)

Get a list of dependency ordinals of a DT_DRV_COMPAT instance's direct dependencies.

Equivalent to *DT_REQUIRES_DEP_ORDS(DT_DRV_INST(inst))*.

Parameters

- **inst** – instance number

Returns
a list of dependency ordinals for the nodes the instance depends on directly

`DT_INST_SUPPORTS_DEP_ORDS`(inst)

Get a list of dependency ordinals of what depends directly on a DT_DRV_COMPAT instance.

Equivalent to *DT_SUPPORTS_DEP_ORDS(DT_DRV_INST(inst))*.

Parameters

- **inst** – instance number

Returns
a list of node identifiers for the nodes that depend directly on the instance

Bus helpers Zephyr's devicetree bindings language supports a `bus:` key which allows bindings to declare that nodes with a given compatible describe system buses. In this case, child nodes are considered to be on a bus of the given type, and the following APIs may be used.

group `devicetree-generic-bus`

Defines

`DT_BUS`(node_id)

Node's bus controller.

Get the node identifier of the node's bus controller. This can be used with *DT_PROP()* to get properties of the bus controller.

It is an error to use this with nodes which do not have bus controllers.

Example devicetree fragment:

```
i2c@deadbeef {
        status = "okay";
        clock-frequency = < 100000 >;

        i2c_device: accelerometer@12 {
                ...
        };
};
```

Example usage:

```
DT_PROP(DT_BUS(DT_NODELABEL(i2c_device)), clock_frequency) // 100000
```

Parameters

- **node_id** – node identifier

Returns

a node identifier for the node's bus controller

`DT_ON_BUS`(node_id, bus)

Is a node on a bus of a given type?

Example devicetree overlay:

```
&i2c0 {
       temp: temperature-sensor@76 {
                compatible = "vnd,some-sensor";
                reg = <0x76>;
       };
};
```

Example usage, assuming `i2c0` is an I2C bus controller node, and therefore `temp` is on an I2C bus:

```
DT_ON_BUS(DT_NODELABEL(temp), i2c) // 1
DT_ON_BUS(DT_NODELABEL(temp), spi) // 0
```

Parameters

- **node_id** – node identifier
- **bus** – lowercase-and-underscores bus type as a C token (i.e. without quotes)

Returns
1 if the node is on a bus of the given type, 0 otherwise

Instance-based APIs These are recommended for use within device drivers. To use them, define DT_DRV_COMPAT to the lowercase-and-underscores compatible the device driver implements support for. Here is an example devicetree fragment:

```
serial@40001000 {
        compatible = "vnd,serial";
        status = "okay";
        current-speed = <115200>;
};
```

Example usage, assuming serial@40001000 is the only enabled node with compatible vnd, serial:

```
#define DT_DRV_COMPAT vnd_serial
DT_DRV_INST(0)                  // node identifier for serial@40001000
DT_INST_PROP(0, current_speed)  // 115200
```

Warning

Be careful making assumptions about instance numbers. See *DT_INST()* for the API guarantees.

As shown above, the DT_INST_* APIs are conveniences for addressing nodes by instance number. They are almost all defined in terms of one of the *Generic APIs*. The equivalent generic API can be found by removing INST_ from the macro name. For example, DT_INST_PROP(inst, prop) is equivalent to DT_PROP(DT_DRV_INST(inst), prop). Similarly, DT_INST_REG_ADDR(inst) is equivalent to DT_REG_ADDR(DT_DRV_INST(inst)), and so on. There are some exceptions: *DT_ANY_INST_ON_BUS_STATUS_OKAY()* and *DT_INST_FOREACH_STATUS_OKAY()* are special-purpose helpers without straightforward generic equivalents.

Since DT_DRV_INST() requires DT_DRV_COMPAT to be defined, it's an error to use any of these without that macro defined.

Note that there are also helpers available for specific hardware; these are documented in *Hardware specific APIs*.

group **devicetree-inst**

Defines

DT_DRV_INST(inst)
Node identifier for an instance of a DT_DRV_COMPAT compatible.

Parameters
- **inst** – instance number

Returns
a node identifier for the node with DT_DRV_COMPAT compatible and instance number inst

DT_INST_PARENT(inst)
Get a DT_DRV_COMPAT parent's node identifier.

See also

DT_PARENT

Parameters

- **inst** – instance number

Returns
a node identifier for the instance's parent

DT_INST_GPARENT(inst)
Get a DT_DRV_COMPAT grandparent's node identifier.

See also

DT_GPARENT

Parameters

- **inst** – instance number

Returns
a node identifier for the instance's grandparent

DT_INST_CHILD(inst, child)
Get a node identifier for a child node of *DT_DRV_INST(inst)*

See also

DT_CHILD

Parameters

- **inst** – instance number
- **child** – lowercase-and-underscores child node name

Returns
node identifier for the node with the name referred to by 'child'

DT_INST_CHILD_NUM(inst)
Get the number of child nodes of a given node.

This is equivalent to

See also

DT_CHILD_NUM(DT_DRV_INST(inst)).

Parameters

- **inst** – Devicetree instance number

Returns
Number of child nodes

`DT_INST_CHILD_NUM_STATUS_OKAY`(inst)

Get the number of child nodes of a given node.

This is equivalent to

> **See also**
> *DT_CHILD_NUM_STATUS_OKAY(DT_DRV_INST(inst))*.

Parameters

- `inst` – Devicetree instance number

Returns
Number of child nodes which status are okay

`DT_INST_NODELABEL_STRING_ARRAY`(inst)

Get a string array of *DT_DRV_INST(inst)*'s node labels.

Equivalent to *DT_NODELABEL_STRING_ARRAY(DT_DRV_INST(inst))*.

Parameters

- `inst` – instance number

Returns
an array initializer for an array of the instance's node labels as strings

`DT_INST_NUM_NODELABELS`(inst)

Get the number of node labels by instance number.

Equivalent to *DT_NUM_NODELABELS(DT_DRV_INST(inst))*.

Parameters

- `inst` – instance number

Returns
the number of node labels that the node with that instance number has

`DT_INST_FOREACH_CHILD`(inst, fn)

Call `fn` on all child nodes of *DT_DRV_INST(inst)*.

The macro `fn` should take one argument, which is the node identifier for the child node.

The children will be iterated over in the same order as they appear in the final devicetree.

> **See also**
> *DT_FOREACH_CHILD*

Parameters

- `inst` – instance number
- `fn` – macro to invoke on each child node identifier

DT_INST_FOREACH_CHILD_SEP(inst, fn, sep)

Call fn on all child nodes of *DT_DRV_INST(inst)* with a separator.

The macro fn should take one argument, which is the node identifier for the child node.

See also

DT_FOREACH_CHILD_SEP

Parameters

- **inst** – instance number
- **fn** – macro to invoke on each child node identifier
- **sep** – Separator (e.g. comma or semicolon). Must be in parentheses; this is required to enable providing a comma as separator.

DT_INST_FOREACH_CHILD_VARGS(inst, fn, ...)

Call fn on all child nodes of *DT_DRV_INST(inst)*.

The macro fn takes multiple arguments. The first should be the node identifier for the child node. The remaining are passed-in by the caller.

The children will be iterated over in the same order as they appear in the final devicetree.

See also

DT_FOREACH_CHILD

Parameters

- **inst** – instance number
- **fn** – macro to invoke on each child node identifier
- **...** – variable number of arguments to pass to fn

DT_INST_FOREACH_CHILD_SEP_VARGS(inst, fn, sep, ...)

Call fn on all child nodes of *DT_DRV_INST(inst)* with separator.

The macro fn takes multiple arguments. The first should be the node identifier for the child node. The remaining are passed-in by the caller.

See also

DT_FOREACH_CHILD_SEP_VARGS

Parameters

- **inst** – instance number
- **fn** – macro to invoke on each child node identifier

- **sep** – Separator (e.g. comma or semicolon). Must be in parentheses; this is required to enable providing a comma as separator.
- **...** – variable number of arguments to pass to fn

DT_INST_FOREACH_CHILD_STATUS_OKAY(inst, fn)

Call fn on all child nodes of *DT_DRV_INST(inst)* with status okay.

The macro fn should take one argument, which is the node identifier for the child node.

See also

DT_FOREACH_CHILD_STATUS_OKAY

Parameters

- **inst** – instance number
- **fn** – macro to invoke on each child node identifier

DT_INST_FOREACH_CHILD_STATUS_OKAY_SEP(inst, fn, sep)

Call fn on all child nodes of *DT_DRV_INST(inst)* with status okay and with separator.

The macro fn should take one argument, which is the node identifier for the child node.

See also

DT_FOREACH_CHILD_STATUS_OKAY_SEP

Parameters

- **inst** – instance number
- **fn** – macro to invoke on each child node identifier
- **sep** – Separator (e.g. comma or semicolon). Must be in parentheses; this is required to enable providing a comma as separator.

DT_INST_FOREACH_CHILD_STATUS_OKAY_VARGS(inst, fn, ...)

Call fn on all child nodes of *DT_DRV_INST(inst)* with status okay and multiple arguments.

The macro fn takes multiple arguments. The first should be the node identifier for the child node. The remaining are passed-in by the caller.

See also

DT_FOREACH_CHILD_STATUS_OKAY_VARGS

Parameters

- **inst** – instance number
- **fn** – macro to invoke on each child node identifier
- **...** – variable number of arguments to pass to fn

DT_INST_FOREACH_CHILD_STATUS_OKAY_SEP_VARGS(inst, fn, sep, ...)

Call fn on all child nodes of *DT_DRV_INST(inst)* with status okay and with separator and multiple arguments.

The macro fn takes multiple arguments. The first should be the node identifier for the child node. The remaining are passed-in by the caller.

See also

DT_FOREACH_CHILD_STATUS_OKAY_SEP_VARGS

Parameters

- **inst** – instance number
- **fn** – macro to invoke on each child node identifier
- **sep** – Separator (e.g. comma or semicolon). Must be in parentheses; this is required to enable providing a comma as separator.
- ... – variable number of arguments to pass to fn

DT_INST_ENUM_IDX(inst, prop)

Get a DT_DRV_COMPAT value's index into its enumeration values.

Parameters

- **inst** – instance number
- **prop** – lowercase-and-underscores property name

Returns

zero-based index of the property's value in its enum: list

DT_INST_ENUM_IDX_OR(inst, prop, default_idx_value)

Like *DT_INST_ENUM_IDX()*, but with a fallback to a default enum index.

Parameters

- **inst** – instance number
- **prop** – lowercase-and-underscores property name
- **default_idx_value** – a fallback index value to expand to

Returns

zero-based index of the property's value in its enum if present, default_idx_value otherwise

DT_INST_ENUM_HAS_VALUE(inst, prop, value)

Does a DT_DRV_COMPAT enumeration property have a given value?

Parameters

- **inst** – instance number
- **prop** – lowercase-and-underscores property name
- **value** – lowercase-and-underscores enumeration value

Returns

1 if the node property has the value *value*, 0 otherwise.

DT_INST_PROP(inst, prop)

Get a DT_DRV_COMPAT instance property.

Parameters

- **inst** – instance number
- **prop** – lowercase-and-underscores property name

Returns
a representation of the property's value

DT_INST_PROP_LEN(inst, prop)

Get a DT_DRV_COMPAT property length.

Parameters

- **inst** – instance number
- **prop** – lowercase-and-underscores property name

Returns
logical length of the property

DT_INST_PROP_HAS_IDX(inst, prop, idx)

Is index idx valid for an array type property on a DT_DRV_COMPAT instance?

Parameters

- **inst** – instance number
- **prop** – lowercase-and-underscores property name
- **idx** – index to check

Returns
1 if idx is a valid index into the given property, 0 otherwise.

DT_INST_PROP_HAS_NAME(inst, prop, name)

Is name name available in a foo-names property?

Parameters

- **inst** – instance number
- **prop** – a lowercase-and-underscores prop-names type property
- **name** – a lowercase-and-underscores name to check

Returns
An expression which evaluates to 1 if name is an available name into the given property, and 0 otherwise.

DT_INST_PROP_BY_IDX(inst, prop, idx)

Get a DT_DRV_COMPAT element value in an array property.

Parameters

- **inst** – instance number
- **prop** – lowercase-and-underscores property name
- **idx** – the index to get

Returns
a representation of the idx-th element of the property

DT_INST_PROP_OR(inst, prop, default_value)

Like *DT_INST_PROP()*, but with a fallback to default_value.

Parameters

- **inst** – instance number
- **prop** – lowercase-and-underscores property name
- **default_value** – a fallback value to expand to

Returns
DT_INST_PROP(inst, prop) or `default_value`

`DT_INST_PROP_LEN_OR`(inst, prop, default_value)

Like *DT_INST_PROP_LEN()*, but with a fallback to `default_value`.

Parameters

- **inst** – instance number
- **prop** – lowercase-and-underscores property name
- **default_value** – a fallback value to expand to

Returns
DT_INST_PROP_LEN(inst, prop) or `default_value`

`DT_INST_STRING_TOKEN`(inst, prop)

Get a `DT_DRV_COMPAT` instance's string property's value as a token.

Parameters

- **inst** – instance number
- **prop** – lowercase-and-underscores property name

Returns
the value of `prop` as a token, i.e. without any quotes and with special characters converted to underscores

`DT_INST_STRING_UPPER_TOKEN`(inst, prop)

Like *DT_INST_STRING_TOKEN()*, but uppercased.

Parameters

- **inst** – instance number
- **prop** – lowercase-and-underscores property name

Returns
the value of `prop` as an uppercased token, i.e. without any quotes and with special characters converted to underscores

`DT_INST_STRING_UNQUOTED`(inst, prop)

Get a `DT_DRV_COMPAT` instance's string property's value as an unquoted sequence of tokens.

Parameters

- **inst** – instance number
- **prop** – lowercase-and-underscores property name

Returns
the value of `prop` as a sequence of tokens, with no quotes

`DT_INST_STRING_TOKEN_BY_IDX`(inst, prop, idx)

Get an element out of string-array property as a token.

Parameters

- **inst** – instance number
- **prop** – lowercase-and-underscores property name

- **idx** – the index to get

Returns
the element in prop at index idx as a token

DT_INST_STRING_UPPER_TOKEN_BY_IDX(inst, prop, idx)

Like *DT_INST_STRING_TOKEN_BY_IDX()*, but uppercased.

Parameters

- **inst** – instance number
- **prop** – lowercase-and-underscores property name
- **idx** – the index to get

Returns
the element in prop at index idx as an uppercased token

DT_INST_STRING_UNQUOTED_BY_IDX(inst, prop, idx)

Get an element out of string-array property as an unquoted sequence of tokens.

Parameters

- **inst** – instance number
- **prop** – lowercase-and-underscores property name
- **idx** – the index to get

Returns
the value of prop at index idx as a sequence of tokens, with no quotes

DT_INST_PROP_BY_PHANDLE(inst, ph, prop)

Get a DT_DRV_COMPAT instance's property value from a phandle's node.

Parameters

- **inst** – instance number
- **ph** – lowercase-and-underscores property of inst with type phandle
- **prop** – lowercase-and-underscores property of the phandle's node

Returns
the value of prop as described in the *DT_PROP()* documentation

DT_INST_PROP_BY_PHANDLE_IDX(inst, phs, idx, prop)

Get a DT_DRV_COMPAT instance's property value from a phandle in a property.

Parameters

- **inst** – instance number
- **phs** – lowercase-and-underscores property with type phandle, phandles, or phandle-array
- **idx** – logical index into "phs", which must be zero if "phs" has type phandle
- **prop** – lowercase-and-underscores property of the phandle's node

Returns
the value of prop as described in the *DT_PROP()* documentation

DT_INST_PHA_BY_IDX(inst, pha, idx, cell)

Get a DT_DRV_COMPAT instance's phandle-array specifier value at an index.

Parameters

- **inst** – instance number

- **pha** – lowercase-and-underscores property with type phandle-array
- **idx** – logical index into the property pha
- **cell** – binding's cell name within the specifier at index idx

Returns
the value of the cell inside the specifier at index idx

DT_INST_PHA_BY_IDX_OR(inst, pha, idx, cell, default_value)

Like *DT_INST_PHA_BY_IDX()*, but with a fallback to default_value.

Parameters

- **inst** – instance number
- **pha** – lowercase-and-underscores property with type phandle-array
- **idx** – logical index into the property pha
- **cell** – binding's cell name within the specifier at index idx
- **default_value** – a fallback value to expand to

Returns
DT_INST_PHA_BY_IDX(inst, pha, idx, cell) or default_value

DT_INST_PHA(inst, pha, cell)

Get a DT_DRV_COMPAT instance's phandle-array specifier value Equivalent to *DT_INST_PHA_BY_IDX(inst, pha, 0, cell)*

Parameters

- **inst** – instance number
- **pha** – lowercase-and-underscores property with type phandle-array
- **cell** – binding's cell name for the specifier at pha index 0

Returns
the cell value

DT_INST_PHA_OR(inst, pha, cell, default_value)

Like *DT_INST_PHA()*, but with a fallback to default_value.

Parameters

- **inst** – instance number
- **pha** – lowercase-and-underscores property with type phandle-array
- **cell** – binding's cell name for the specifier at pha index 0
- **default_value** – a fallback value to expand to

Returns
DT_INST_PHA(inst, pha, cell) or default_value

DT_INST_PHA_BY_NAME(inst, pha, name, cell)

Get a DT_DRV_COMPAT instance's value within a phandle-array specifier by name.

Parameters

- **inst** – instance number
- **pha** – lowercase-and-underscores property with type phandle-array
- **name** – lowercase-and-underscores name of a specifier in pha
- **cell** – binding's cell name for the named specifier

Returns
the cell value

DT_INST_PHA_BY_NAME_OR(inst, pha, name, cell, default_value)

Like *DT_INST_PHA_BY_NAME()*, but with a fallback to default_value.

Parameters

- **inst** – instance number
- **pha** – lowercase-and-underscores property with type phandle-array
- **name** – lowercase-and-underscores name of a specifier in pha
- **cell** – binding's cell name for the named specifier
- **default_value** – a fallback value to expand to

Returns

DT_INST_PHA_BY_NAME(inst, pha, name, cell) or default_value

DT_INST_PHANDLE_BY_NAME(inst, pha, name)

Get a DT_DRV_COMPAT instance's phandle node identifier from a phandle array by name.

Parameters

- **inst** – instance number
- **pha** – lowercase-and-underscores property with type phandle-array
- **name** – lowercase-and-underscores name of an element in pha

Returns

node identifier for the phandle at the element named "name"

DT_INST_PHANDLE_BY_IDX(inst, prop, idx)

Get a DT_DRV_COMPAT instance's node identifier for a phandle in a property.

Parameters

- **inst** – instance number
- **prop** – lowercase-and-underscores property name in inst with type phandle, phandles or phandle-array
- **idx** – index into prop

Returns

a node identifier for the phandle at index idx in prop

DT_INST_PHANDLE(inst, prop)

Get a DT_DRV_COMPAT instance's node identifier for a phandle property's value.

Parameters

- **inst** – instance number
- **prop** – lowercase-and-underscores property of inst with type phandle

Returns

a node identifier for the node pointed to by "ph"

DT_INST_REG_HAS_IDX(inst, idx)

is idx a valid register block index on a DT_DRV_COMPAT instance?

Parameters

- **inst** – instance number
- **idx** – index to check

Returns

1 if idx is a valid register block index, 0 otherwise.

DT_INST_REG_HAS_NAME(inst, name)

is name a valid register block name on a DT_DRV_COMPAT instance?

Parameters

- **inst** – instance number
- **name** – name to check

Returns

1 if name is a valid register block name, 0 otherwise.

DT_INST_REG_ADDR_BY_IDX(inst, idx)

Get a DT_DRV_COMPAT instance's idx-th register block's address.

Parameters

- **inst** – instance number
- **idx** – index of the register whose address to return

Returns

address of the instance's idx-th register block

DT_INST_REG_SIZE_BY_IDX(inst, idx)

Get a DT_DRV_COMPAT instance's idx-th register block's size.

Parameters

- **inst** – instance number
- **idx** – index of the register whose size to return

Returns

size of the instance's idx-th register block

DT_INST_REG_ADDR_BY_NAME(inst, name)

Get a DT_DRV_COMPAT's register block address by name.

Parameters

- **inst** – instance number
- **name** – lowercase-and-underscores register specifier name

Returns

address of the register block with the given name

DT_INST_REG_ADDR_BY_NAME_OR(inst, name, default_value)

Like *DT_INST_REG_ADDR_BY_NAME()*, but with a fallback to default_value.

Parameters

- **inst** – instance number
- **name** – lowercase-and-underscores register specifier name
- **default_value** – a fallback value to expand to

Returns

address of the register block specified by name if present, default_value otherwise

DT_INST_REG_ADDR_BY_NAME_U64(inst, name)

64-bit version of *DT_INST_REG_ADDR_BY_NAME()*

This macro version adds the appropriate suffix for 64-bit unsigned integer literals. Note that this macro is equivalent to *DT_INST_REG_ADDR_BY_NAME()* in linker/ASM context.

Parameters

- **inst** – instance number
- **name** – lowercase-and-underscores register specifier name

Returns
address of the register block with the given `name`

`DT_INST_REG_SIZE_BY_NAME`(inst, name)

Get a `DT_DRV_COMPAT`'s register block size by name.

Parameters

- **inst** – instance number
- **name** – lowercase-and-underscores register specifier name

Returns
size of the register block with the given `name`

`DT_INST_REG_SIZE_BY_NAME_OR`(inst, name, default_value)

Like *DT_INST_REG_SIZE_BY_NAME()*, but with a fallback to `default_value`.

Parameters

- **inst** – instance number
- **name** – lowercase-and-underscores register specifier name
- **default_value** – a fallback value to expand to

Returns
size of the register block specified by name if present, `default_value` otherwise

`DT_INST_REG_ADDR`(inst)

Get a `DT_DRV_COMPAT`'s (only) register block address.

Parameters

- **inst** – instance number

Returns
instance's register block address

`DT_INST_REG_ADDR_U64`(inst)

64-bit version of *DT_INST_REG_ADDR()*

This macro version adds the appropriate suffix for 64-bit unsigned integer literals. Note that this macro is equivalent to *DT_INST_REG_ADDR()* in linker/ASM context.

Parameters

- **inst** – instance number

Returns
instance's register block address

`DT_INST_REG_SIZE`(inst)

Get a `DT_DRV_COMPAT`'s (only) register block size.

Parameters

- **inst** – instance number

Returns
instance's register block size

DT_INST_IRQ_LEVEL(inst)

Get a DT_DRV_COMPAT interrupt level.

Parameters

- **inst** – instance number

Returns
interrupt level

DT_INST_IRQ_BY_IDX(inst, idx, cell)

Get a DT_DRV_COMPAT interrupt specifier value at an index.

Parameters

- **inst** – instance number
- **idx** – logical index into the interrupt specifier array
- **cell** – cell name specifier

Returns
the named value at the specifier given by the index

DT_INST_IRQ_INTC_BY_IDX(inst, idx)

Get a DT_DRV_COMPAT interrupt specifier's interrupt controller by index.

Parameters

- **inst** – instance number
- **idx** – interrupt specifier's index

Returns
node_id of interrupt specifier's interrupt controller

DT_INST_IRQ_INTC_BY_NAME(inst, name)

Get a DT_DRV_COMPAT interrupt specifier's interrupt controller by name.

Parameters

- **inst** – instance number
- **name** – interrupt specifier's name

Returns
node_id of interrupt specifier's interrupt controller

DT_INST_IRQ_INTC(inst)

Get a DT_DRV_COMPAT interrupt specifier's interrupt controller.

See also

DT_INST_IRQ_INTC_BY_IDX()

Note

Equivalent to *DT_INST_IRQ_INTC_BY_IDX(node_id, 0)*

Parameters

- **inst** – instance number

Returns
node_id of interrupt specifier's interrupt controller

`DT_INST_IRQ_BY_NAME`(inst, name, cell)
Get a `DT_DRV_COMPAT` interrupt specifier value by name.

Parameters
- **inst** – instance number
- **name** – lowercase-and-underscores interrupt specifier name
- **cell** – cell name specifier

Returns
the named value at the specifier given by the index

`DT_INST_IRQ`(inst, cell)
Get a `DT_DRV_COMPAT` interrupt specifier's value.

Parameters
- **inst** – instance number
- **cell** – cell name specifier

Returns
the named value at that index

`DT_INST_IRQN`(inst)
Get a `DT_DRV_COMPAT`'s (only) irq number.

Parameters
- **inst** – instance number

Returns
the interrupt number for the node's only interrupt

`DT_INST_IRQN_BY_IDX`(inst, idx)
Get a `DT_DRV_COMPAT`'s irq number at index.

Parameters
- **inst** – instance number
- **idx** – logical index into the interrupt specifier array

Returns
the interrupt number for the node's idx-th interrupt

`DT_INST_BUS`(inst)
Get a `DT_DRV_COMPAT`'s bus node identifier.

Parameters
- **inst** – instance number

Returns
node identifier for the instance's bus node

`DT_INST_ON_BUS`(inst, bus)
Test if a `DT_DRV_COMPAT`'s bus type is a given type.

Parameters
- **inst** – instance number
- **bus** – a binding's bus type as a C token, lowercased and without quotes

Returns
1 if the given instance is on a bus of the given type, 0 otherwise

`DT_INST_STRING_TOKEN_OR`(inst, name, default_value)

Like *DT_INST_STRING_TOKEN()*, but with a fallback to `default_value`.

Parameters

- **inst** – instance number
- **name** – lowercase-and-underscores property name
- **default_value** – a fallback value to expand to

Returns
if `prop` exists, its value as a token, i.e. without any quotes and with special characters converted to underscores. Otherwise `default_value`

`DT_INST_STRING_UPPER_TOKEN_OR`(inst, name, default_value)

Like *DT_INST_STRING_UPPER_TOKEN()*, but with a fallback to `default_value`.

Parameters

- **inst** – instance number
- **name** – lowercase-and-underscores property name
- **default_value** – a fallback value to expand to

Returns
the property's value as an uppercased token, or `default_value`

`DT_INST_STRING_UNQUOTED_OR`(inst, name, default_value)

Like *DT_INST_STRING_UNQUOTED()*, but with a fallback to `default_value`.

Parameters

- **inst** – instance number
- **name** – lowercase-and-underscores property name
- **default_value** – a fallback value to expand to

Returns
the property's value as a sequence of tokens, with no quotes, or `default_value`

`DT_HAS_COMPAT_ON_BUS_STATUS_OKAY`(compat, bus)

`DT_ANY_INST_ON_BUS_STATUS_OKAY`(bus)

Test if any `DT_DRV_COMPAT` node is on a bus of a given type and has status okay.

This is a special-purpose macro which can be useful when writing drivers for devices which can appear on multiple buses. One example is a sensor device which may be wired on an I2C or SPI bus.

Example devicetree overlay:

```
&i2c0 {
        temp: temperature-sensor@76 {
                compatible = "vnd,some-sensor";
                reg = <0x76>;
        };
};
```

Example usage, assuming `i2c0` is an I2C bus controller node, and therefore `temp` is on an I2C bus:

```
#define DT_DRV_COMPAT vnd_some_sensor

DT_ANY_INST_ON_BUS_STATUS_OKAY(i2c) // 1
```

Parameters

- **bus** – a binding's bus type as a C token, lowercased and without quotes

Returns

1 if any enabled node with that compatible is on that bus type, 0 otherwise

DT_ANY_INST_HAS_PROP_STATUS_OKAY(prop)

Check if any DT_DRV_COMPAT node with status okay has a given property.

Example devicetree overlay:

```
&i2c0 {
    sensor0: sensor@0 {
        compatible = "vnd,some-sensor";
        status = "okay";
        reg = <0>;
        foo = <1>;
        bar = <2>;
    };

    sensor1: sensor@1 {
        compatible = "vnd,some-sensor";
        status = "okay";
        reg = <1>;
        foo = <2>;
    };

    sensor2: sensor@2 {
        compatible = "vnd,some-sensor";
        status = "disabled";
        reg = <2>;
        baz = <1>;
    };
};
```

Example usage:

```
#define DT_DRV_COMPAT vnd_some_sensor

DT_ANY_INST_HAS_PROP_STATUS_OKAY(foo) // 1
DT_ANY_INST_HAS_PROP_STATUS_OKAY(bar) // 1
DT_ANY_INST_HAS_PROP_STATUS_OKAY(baz) // 0
```

Parameters

- **prop** – lowercase-and-underscores property name

DT_INST_FOREACH_STATUS_OKAY(fn)

Call fn on all nodes with compatible DT_DRV_COMPAT and status okay

This macro calls fn(inst) on each inst number that refers to a node with status okay. Whitespace is added between invocations.

Example devicetree fragment:

```
a {
        compatible = "vnd,device";
        status = "okay";
        foobar = "DEV_A";
};

b {
        compatible = "vnd,device";
        status = "okay";
        foobar = "DEV_B";
};

c {
        compatible = "vnd,device";
        status = "disabled";
        foobar = "DEV_C";
};
```

Example usage:

```
#define DT_DRV_COMPAT vnd_device
#define MY_FN(inst) DT_INST_PROP(inst, foobar),

DT_INST_FOREACH_STATUS_OKAY(MY_FN)
```

This expands to:

```
MY_FN(0) MY_FN(1)
```

and from there, to either this:

```
"DEV_A", "DEV_B",
```

or this:

```
"DEV_B", "DEV_A",
```

No guarantees are made about the order that a and b appear in the expansion.

Note that `fn` is responsible for adding commas, semicolons, or other separators or terminators.

Device drivers should use this macro whenever possible to instantiate a struct device for each enabled node in the devicetree of the driver's compatible `DT_DRV_COMPAT`.

Parameters

- **fn** – Macro to call for each enabled node. Must accept an instance number as its only parameter.

`DT_INST_FOREACH_STATUS_OKAY_VARGS`(fn, ...)

Call `fn` on all nodes with compatible `DT_DRV_COMPAT` and status `okay` with multiple arguments.

See also

DT_INST_FOREACH_STATUS_OKAY

Parameters

- **fn** – Macro to call for each enabled node. Must accept an instance number as its only parameter.
- **...** – variable number of arguments to pass to fn

DT_INST_FOREACH_NODELABEL(inst, fn)

Call fn on all node labels for a given DT_DRV_COMPAT instance.

Equivalent to *DT_FOREACH_NODELABEL(DT_DRV_INST(inst), fn)*.

Parameters

- **inst** – instance number
- **fn** – macro which will be passed each node label for the node with that instance number

DT_INST_FOREACH_NODELABEL_VARGS(inst, fn, ...)

Call fn on all node labels for a given DT_DRV_COMPAT instance with multiple arguments.

Equivalent to *DT_FOREACH_NODELABEL_VARGS(DT_DRV_INST(inst), fn, ...)*.

Parameters

- **inst** – instance number
- **fn** – macro which will be passed each node label for the node with that instance number
- **...** – additional arguments to pass to fn

DT_INST_FOREACH_PROP_ELEM(inst, prop, fn)

Invokes fn for each element of property prop for a DT_DRV_COMPAT instance.

Equivalent to *DT_FOREACH_PROP_ELEM(DT_DRV_INST(inst), prop, fn)*.

Parameters

- **inst** – instance number
- **prop** – lowercase-and-underscores property name
- **fn** – macro to invoke

DT_INST_FOREACH_PROP_ELEM_SEP(inst, prop, fn, sep)

Invokes fn for each element of property prop for a DT_DRV_COMPAT instance with a separator.

Equivalent to *DT_FOREACH_PROP_ELEM_SEP(DT_DRV_INST(inst), prop, fn, sep)*.

Parameters

- **inst** – instance number
- **prop** – lowercase-and-underscores property name
- **fn** – macro to invoke
- **sep** – Separator (e.g. comma or semicolon). Must be in parentheses; this is required to enable providing a comma as separator.

DT_INST_FOREACH_PROP_ELEM_VARGS(inst, prop, fn, ...)

Invokes fn for each element of property prop for a DT_DRV_COMPAT instance with multiple arguments.

Equivalent to DT_FOREACH_PROP_ELEM_VARGS(*DT_DRV_INST(inst)*, prop, fn, **VA_ARGS**)

See also

DT_INST_FOREACH_PROP_ELEM

Parameters

- `inst` – instance number
- `prop` – lowercase-and-underscores property name
- `fn` – macro to invoke
- `...` – variable number of arguments to pass to `fn`

`DT_INST_FOREACH_PROP_ELEM_SEP_VARGS`(inst, prop, fn, sep, ...)

Invokes `fn` for each element of property `prop` for a `DT_DRV_COMPAT` instance with multiple arguments and a separator.

Equivalent to DT_FOREACH_PROP_ELEM_SEP_VARGS(*DT_DRV_INST(inst)*, prop, fn, sep, **VA_ARGS**)

See also

DT_INST_FOREACH_PROP_ELEM

Parameters

- `inst` – instance number
- `prop` – lowercase-and-underscores property name
- `fn` – macro to invoke
- `sep` – Separator (e.g. comma or semicolon). Must be in parentheses; this is required to enable providing a comma as separator.
- `...` – variable number of arguments to pass to fn

`DT_INST_NODE_HAS_PROP`(inst, prop)

Does a DT_DRV_COMPAT instance have a property?

Parameters

- `inst` – instance number
- `prop` – lowercase-and-underscores property name

Returns

1 if the instance has the property, 0 otherwise.

`DT_INST_NODE_HAS_COMPAT`(inst, compat)

Does a DT_DRV_COMPAT instance have the compatible?

Parameters

- `inst` – instance number
- `compat` – lowercase-and-underscores compatible, without quotes

Returns

1 if the instance matches the compatible, 0 otherwise.

DT_INST_PHA_HAS_CELL_AT_IDX(inst, pha, idx, cell)

Does a phandle array have a named cell specifier at an index for a DT_DRV_COMPAT instance?

Parameters

- **inst** – instance number
- **pha** – lowercase-and-underscores property with type phandle-array
- **idx** – index to check
- **cell** – named cell value whose existence to check

Returns

1 if the named cell exists in the specifier at index idx, 0 otherwise.

DT_INST_PHA_HAS_CELL(inst, pha, cell)

Does a phandle array have a named cell specifier at index 0 for a DT_DRV_COMPAT instance?

Parameters

- **inst** – instance number
- **pha** – lowercase-and-underscores property with type phandle-array
- **cell** – named cell value whose existence to check

Returns

1 if the named cell exists in the specifier at index 0, 0 otherwise.

DT_INST_IRQ_HAS_IDX(inst, idx)

is index valid for interrupt property on a DT_DRV_COMPAT instance?

Parameters

- **inst** – instance number
- **idx** – logical index into the interrupt specifier array

Returns

1 if the idx is valid for the interrupt property 0 otherwise.

DT_INST_IRQ_HAS_CELL_AT_IDX(inst, idx, cell)

Does a DT_DRV_COMPAT instance have an interrupt named cell specifier?

Parameters

- **inst** – instance number
- **idx** – index to check
- **cell** – named cell value whose existence to check

Returns

1 if the named cell exists in the interrupt specifier at index idx 0 otherwise.

DT_INST_IRQ_HAS_CELL(inst, cell)

Does a DT_DRV_COMPAT instance have an interrupt value?

Parameters

- **inst** – instance number
- **cell** – named cell value whose existence to check

Returns

1 if the named cell exists in the interrupt specifier at index 0 0 otherwise.

DT_INST_IRQ_HAS_NAME(inst, name)

Does a DT_DRV_COMPAT instance have an interrupt value?

Parameters

- **inst** – instance number
- **name** – lowercase-and-underscores interrupt specifier name

Returns

1 if name is a valid named specifier

Hardware specific APIs The following APIs can also be used by including <devicetree.h>; no additional include is needed.

CAN These conveniences may be used for nodes which describe CAN controllers/transceivers, and properties related to them.

group devicetree-can

Defines

DT_CAN_TRANSCEIVER_MIN_BITRATE(node_id, min)

Get the minimum transceiver bitrate for a CAN controller.

The bitrate will be limited to the minimum bitrate supported by the CAN controller. If no CAN transceiver is present in the devicetree, the minimum bitrate will be that of the CAN controller.

Example devicetree fragment:

```
transceiver0: can-phy0 {
        compatible = "vnd,can-transceiver";
        min-bitrate = <15000>;
        max-bitrate = <1000000>;
        #phy-cells = <0>;
};

can0: can@... {
        compatible = "vnd,can-controller";
        phys = <&transceiver0>;
};

can1: can@... {
        compatible = "vnd,can-controller";

        can-transceiver {
                min-bitrate = <25000>;
                max-bitrate = <2000000>;
        };
};

can2: can@... {
        compatible = "vnd,can-controller";

        can-transceiver {
                max-bitrate = <2000000>;
        };
};
```

Example usage:

```
DT_CAN_TRANSCEIVER_MIN_BITRATE(DT_NODELABEL(can0), 10000) // 15000
DT_CAN_TRANSCEIVER_MIN_BITRATE(DT_NODELABEL(can1), 0)     // 250000
DT_CAN_TRANSCEIVER_MIN_BITRATE(DT_NODELABEL(can1), 50000) // 500000
DT_CAN_TRANSCEIVER_MIN_BITRATE(DT_NODELABEL(can2), 0)     // 0
```

Parameters

- `node_id` – node identifier
- `min` – minimum bitrate supported by the CAN controller

Returns
the minimum bitrate supported by the CAN controller/transceiver combination

`DT_CAN_TRANSCEIVER_MAX_BITRATE`(node_id, max)

Get the maximum transceiver bitrate for a CAN controller.

The bitrate will be limited to the maximum bitrate supported by the CAN controller. If no CAN transceiver is present in the devicetree, the maximum bitrate will be that of the CAN controller.

Example devicetree fragment:

```
transceiver0: can-phy0 {
        compatible = "vnd,can-transceiver";
        max-bitrate = <1000000>;
        #phy-cells = <0>;
};

can0: can@... {
        compatible = "vnd,can-controller";
        phys = <&transceiver0>;
};

can1: can@... {
        compatible = "vnd,can-controller";

        can-transceiver {
                max-bitrate = <2000000>;
        };
};
```

Example usage:

```
DT_CAN_TRANSCEIVER_MAX_BITRATE(DT_NODELABEL(can0), 5000000) // 1000000
DT_CAN_TRANSCEIVER_MAX_BITRATE(DT_NODELABEL(can1), 5000000) // 2000000
DT_CAN_TRANSCEIVER_MAX_BITRATE(DT_NODELABEL(can1), 1000000) // 1000000
```

Parameters

- `node_id` – node identifier
- `max` – maximum bitrate supported by the CAN controller

Returns
the maximum bitrate supported by the CAN controller/transceiver combination

DT_INST_CAN_TRANSCEIVER_MIN_BITRATE(inst, min)

Get the minimum transceiver bitrate for a DT_DRV_COMPAT CAN controller.

See also

DT_CAN_TRANSCEIVER_MIN_BITRATE()

Parameters

- **inst** – DT_DRV_COMPAT instance number
- **min** – minimum bitrate supported by the CAN controller

Returns

the minimum bitrate supported by the CAN controller/transceiver combination

DT_INST_CAN_TRANSCEIVER_MAX_BITRATE(inst, max)

Get the maximum transceiver bitrate for a DT_DRV_COMPAT CAN controller.

See also

DT_CAN_TRANSCEIVER_MAX_BITRATE()

Parameters

- **inst** – DT_DRV_COMPAT instance number
- **max** – maximum bitrate supported by the CAN controller

Returns

the maximum bitrate supported by the CAN controller/transceiver combination

Clocks These conveniences may be used for nodes which describe clock sources, and properties related to them.

group devicetree-clocks

Defines

DT_CLOCKS_HAS_IDX(node_id, idx)

Test if a node has a clocks phandle-array property at a given index.

This expands to 1 if the given index is valid clocks property phandle-array index. Otherwise, it expands to 0.

Example devicetree fragment:

```
n1: node-1 {
        clocks = <...>, <...>;
};
```

(continues on next page)

(continued from previous page)

```
n2: node-2 {
        clocks = <...>;
};
```

Example usage:

```
DT_CLOCKS_HAS_IDX(DT_NODELABEL(n1), 0) // 1
DT_CLOCKS_HAS_IDX(DT_NODELABEL(n1), 1) // 1
DT_CLOCKS_HAS_IDX(DT_NODELABEL(n1), 2) // 0
DT_CLOCKS_HAS_IDX(DT_NODELABEL(n2), 1) // 0
```

Parameters

- `node_id` – node identifier; may or may not have any clocks property
- `idx` – index of a clocks property phandle-array whose existence to check

Returns
1 if the index exists, 0 otherwise

`DT_CLOCKS_HAS_NAME`(node_id, name)

Test if a node has a clock-names array property holds a given name.

This expands to 1 if the name is available as clocks-name array property cell. Otherwise, it expands to 0.

Example devicetree fragment:

```
n1: node-1 {
        clocks = <...>, <...>;
        clock-names = "alpha", "beta";
};

n2: node-2 {
        clocks = <...>;
        clock-names = "alpha";
};
```

Example usage:

```
DT_CLOCKS_HAS_NAME(DT_NODELABEL(n1), alpha) // 1
DT_CLOCKS_HAS_NAME(DT_NODELABEL(n1), beta)  // 1
DT_CLOCKS_HAS_NAME(DT_NODELABEL(n2), beta)  // 0
```

Parameters

- `node_id` – node identifier; may or may not have any clock-names property.
- `name` – lowercase-and-underscores clock-names cell value name to check

Returns
1 if the clock name exists, 0 otherwise

`DT_NUM_CLOCKS`(node_id)

Get the number of elements in a clocks property.

Example devicetree fragment:

```
n1: node-1 {
        clocks = <&foo>, <&bar>;
};
```

(continues on next page)

(continued from previous page)

```
n2: node-2 {
        clocks = <&foo>;
};
```

Example usage:

```
DT_NUM_CLOCKS(DT_NODELABEL(n1)) // 2
DT_NUM_CLOCKS(DT_NODELABEL(n2)) // 1
```

Parameters

- `node_id` – node identifier with a clocks property

Returns
number of elements in the property

`DT_CLOCKS_CTLR_BY_IDX`(node_id, idx)

Get the node identifier for the controller phandle from a “clocks” phandle-array property at an index.

Example devicetree fragment:

```
clk1: clock-controller@... { ... };

clk2: clock-controller@... { ... };

n: node {
        clocks = <&clk1 10 20>, <&clk2 30 40>;
};
```

Example usage:

```
DT_CLOCKS_CTLR_BY_IDX(DT_NODELABEL(n), 0)) // DT_NODELABEL(clk1)
DT_CLOCKS_CTLR_BY_IDX(DT_NODELABEL(n), 1)) // DT_NODELABEL(clk2)
```

See also

DT_PHANDLE_BY_IDX()

Parameters

- `node_id` – node identifier
- `idx` – logical index into “clocks”

Returns
the node identifier for the clock controller referenced at index “idx”

`DT_CLOCKS_CTLR`(node_id)

Equivalent to *DT_CLOCKS_CTLR_BY_IDX(node_id, 0)*

See also

DT_CLOCKS_CTLR_BY_IDX()

Parameters

- **node_id** – node identifier

Returns
a node identifier for the clocks controller at index 0 in “clocks”

DT_CLOCKS_CTLR_BY_NAME(node_id, name)

Get the node identifier for the controller phandle from a clocks phandle-array property by name.

Example devicetree fragment:

```
clk1: clock-controller@... { ... };

clk2: clock-controller@... { ... };

n: node {
        clocks = <&clk1 10 20>, <&clk2 30 40>;
        clock-names = "alpha", "beta";
};
```

Example usage:

```
DT_CLOCKS_CTLR_BY_NAME(DT_NODELABEL(n), beta) // DT_NODELABEL(clk2)
```

See also

DT_PHANDLE_BY_NAME()

Parameters

- **node_id** – node identifier
- **name** – lowercase-and-underscores name of a clocks element as defined by the node’s clock-names property

Returns
the node identifier for the clock controller referenced by name

DT_CLOCKS_CELL_BY_IDX(node_id, idx, cell)

Get a clock specifier’s cell value at an index.

Example devicetree fragment:

```
clk1: clock-controller@... {
        compatible = "vnd,clock";
        #clock-cells = < 2 >;
};

n: node {
        clocks = < &clk1 10 20 >, < &clk1 30 40 >;
};
```

Bindings fragment for the vnd,clock compatible:

```
clock-cells:
  - bus
  - bits
```

Example usage:

```
DT_CLOCKS_CELL_BY_IDX(DT_NODELABEL(n), 0, bus) // 10
DT_CLOCKS_CELL_BY_IDX(DT_NODELABEL(n), 1, bits) // 40
```

See also

DT_PHA_BY_IDX()

Parameters

- **node_id** – node identifier for a node with a clocks property
- **idx** – logical index into clocks property
- **cell** – lowercase-and-underscores cell name

Returns
the cell value at index “idx”

DT_CLOCKS_CELL_BY_NAME(node_id, name, cell)

Get a clock specifier’s cell value by name.

Example devicetree fragment:

```
clk1: clock-controller@... {
        compatible = "vnd,clock";
        #clock-cells = < 2 >;
};

n: node {
        clocks = < &clk1 10 20 >, < &clk1 30 40 >;
        clock-names = "alpha", "beta";
};
```

Bindings fragment for the vnd,clock compatible:

```
clock-cells:
  - bus
  - bits
```

Example usage:

```
DT_CLOCKS_CELL_BY_NAME(DT_NODELABEL(n), alpha, bus) // 10
DT_CLOCKS_CELL_BY_NAME(DT_NODELABEL(n), beta, bits) // 40
```

See also

DT_PHA_BY_NAME()

Parameters

- **node_id** – node identifier for a node with a clocks property
- **name** – lowercase-and-underscores name of a clocks element as defined by the node’s clock-names property
- **cell** – lowercase-and-underscores cell name

Returns
the cell value in the specifier at the named element

DT_CLOCKS_CELL(node_id, cell)

Equivalent to *DT_CLOCKS_CELL_BY_IDX(node_id, 0, cell)*

See also

DT_CLOCKS_CELL_BY_IDX()

Parameters

- `node_id` – node identifier for a node with a clocks property
- `cell` – lowercase-and-underscores cell name

Returns
the cell value at index 0

`DT_INST_CLOCKS_HAS_IDX`(inst, idx)

Equivalent to *DT_CLOCKS_HAS_IDX(DT_DRV_INST(inst), idx)*

Parameters

- `inst` – DT_DRV_COMPAT instance number; may or may not have any clocks property
- `idx` – index of a clocks property phandle-array whose existence to check

Returns
1 if the index exists, 0 otherwise

`DT_INST_CLOCKS_HAS_NAME`(inst, name)

Equivalent to DT_CLOCK_HAS_NAME(DT_DRV_INST(inst), name)

Parameters

- `inst` – DT_DRV_COMPAT instance number; may or may not have any clock-names property.
- `name` – lowercase-and-underscores clock-names cell value name to check

Returns
1 if the clock name exists, 0 otherwise

`DT_INST_NUM_CLOCKS`(inst)

Equivalent to *DT_NUM_CLOCKS(DT_DRV_INST(inst))*

Parameters

- `inst` – instance number

Returns
number of elements in the clocks property

`DT_INST_CLOCKS_CTLR_BY_IDX`(inst, idx)

Get the node identifier for the controller phandle from a “clocks” phandle-array property at an index.

See also

DT_CLOCKS_CTLR_BY_IDX()

Parameters

- `inst` – instance number
- `idx` – logical index into “clocks”

Returns
the node identifier for the clock controller referenced at index “idx”

`DT_INST_CLOCKS_CTLR`(inst)
Equivalent to *DT_INST_CLOCKS_CTLR_BY_IDX(inst, 0)*

See also
DT_CLOCKS_CTLR()

Parameters
- `inst` – instance number

Returns
a node identifier for the clocks controller at index 0 in “clocks”

`DT_INST_CLOCKS_CTLR_BY_NAME`(inst, name)
Get the node identifier for the controller phandle from a clocks phandle-array property by name.

See also
DT_CLOCKS_CTLR_BY_NAME()

Parameters
- `inst` – instance number
- `name` – lowercase-and-underscores name of a clocks element as defined by the node’s clock-names property

Returns
the node identifier for the clock controller referenced by the named element

`DT_INST_CLOCKS_CELL_BY_IDX`(inst, idx, cell)
Get a DT_DRV_COMPAT instance’s clock specifier’s cell value at an index.

See also
DT_CLOCKS_CELL_BY_IDX()

Parameters
- `inst` – DT_DRV_COMPAT instance number
- `idx` – logical index into clocks property
- `cell` – lowercase-and-underscores cell name

Returns
the cell value at index “idx”

DT_INST_CLOCKS_CELL_BY_NAME(inst, name, cell)

Get a DT_DRV_COMPAT instance's clock specifier's cell value by name.

See also

DT_CLOCKS_CELL_BY_NAME()

Parameters

- **inst** – DT_DRV_COMPAT instance number
- **name** – lowercase-and-underscores name of a clocks element as defined by the node's clock-names property
- **cell** – lowercase-and-underscores cell name

Returns

the cell value in the specifier at the named element

DT_INST_CLOCKS_CELL(inst, cell)

Equivalent to *DT_INST_CLOCKS_CELL_BY_IDX(inst, 0, cell)*

Parameters

- **inst** – DT_DRV_COMPAT instance number
- **cell** – lowercase-and-underscores cell name

Returns

the value of the cell inside the specifier at index 0

DMA These conveniences may be used for nodes which describe direct memory access controllers or channels, and properties related to them.

group devicetree-dmas

Defines

DT_DMAS_CTLR_BY_IDX(node_id, idx)

Get the node identifier for the DMA controller from a dmas property at an index.

Example devicetree fragment:

```
dma1: dma@... { ... };

dma2: dma@... { ... };

n: node {
    dmas = <&dma1 1 2 0x400 0x3>,
           <&dma2 6 3 0x404 0x5>;
};
```

Example usage:

```
DT_DMAS_CTLR_BY_IDX(DT_NODELABEL(n), 0) // DT_NODELABEL(dma1)
DT_DMAS_CTLR_BY_IDX(DT_NODELABEL(n), 1) // DT_NODELABEL(dma2)
```

See also

DT_PROP_BY_PHANDLE_IDX()

Parameters

- **node_id** – node identifier for a node with a dmas property
- **idx** – logical index into dmas property

Returns
the node identifier for the DMA controller referenced at index “idx”

DT_DMAS_CTLR_BY_NAME(node_id, name)

Get the node identifier for the DMA controller from a dmas property by name.

Example devicetree fragment:

```
dma1: dma@... { ... };

dma2: dma@... { ... };

n: node {
    dmas = <&dma1 1 2 0x400 0x3>,
           <&dma2 6 3 0x404 0x5>;
    dma-names = "tx", "rx";
};
```

Example usage:

```
DT_DMAS_CTLR_BY_NAME(DT_NODELABEL(n), tx) // DT_NODELABEL(dma1)
DT_DMAS_CTLR_BY_NAME(DT_NODELABEL(n), rx) // DT_NODELABEL(dma2)
```

See also

DT_PHANDLE_BY_NAME()

Parameters

- **node_id** – node identifier for a node with a dmas property
- **name** – lowercase-and-underscores name of a dmas element as defined by the node’s dma-names property

Returns
the node identifier for the DMA controller in the named element

DT_DMAS_CTLR(node_id)

Equivalent to *DT_DMAS_CTLR_BY_IDX(node_id, 0)*

See also

DT_DMAS_CTLR_BY_IDX()

Parameters

- **node_id** – node identifier for a node with a dmas property

Returns
the node identifier for the DMA controller at index 0 in the node's "dmas" property

`DT_INST_DMAS_CTLR_BY_IDX`(inst, idx)
Get the node identifier for the DMA controller from a DT_DRV_COMPAT instance's dmas property at an index.

See also
DT_DMAS_CTLR_BY_IDX()

Parameters
- `inst` – DT_DRV_COMPAT instance number
- `idx` – logical index into dmas property

Returns
the node identifier for the DMA controller referenced at index "idx"

`DT_INST_DMAS_CTLR_BY_NAME`(inst, name)
Get the node identifier for the DMA controller from a DT_DRV_COMPAT instance's dmas property by name.

See also
DT_DMAS_CTLR_BY_NAME()

Parameters
- `inst` – DT_DRV_COMPAT instance number
- `name` – lowercase-and-underscores name of a dmas element as defined by the node's dma-names property

Returns
the node identifier for the DMA controller in the named element

`DT_INST_DMAS_CTLR`(inst)
Equivalent to *DT_INST_DMAS_CTLR_BY_IDX(inst, 0)*

See also
DT_DMAS_CTLR_BY_IDX()

Parameters
- `inst` – DT_DRV_COMPAT instance number

Returns
the node identifier for the DMA controller at index 0 in the instance's "dmas" property

DT_DMAS_CELL_BY_IDX(node_id, idx, cell)

Get a DMA specifier's cell value at an index.

Example devicetree fragment:

```
dma1: dma@... {
        compatible = "vnd,dma";
        #dma-cells = <2>;
};

dma2: dma@... {
        compatible = "vnd,dma";
        #dma-cells = <2>;
};

n: node {
    dmas = <&dma1 1 0x400>,
           <&dma2 6 0x404>;
};
```

Bindings fragment for the vnd,dma compatible:

```
dma-cells:
  - channel
  - config
```

Example usage:

```
DT_DMAS_CELL_BY_IDX(DT_NODELABEL(n), 0, channel) // 1
DT_DMAS_CELL_BY_IDX(DT_NODELABEL(n), 1, channel) // 6
DT_DMAS_CELL_BY_IDX(DT_NODELABEL(n), 0, config) // 0x400
DT_DMAS_CELL_BY_IDX(DT_NODELABEL(n), 1, config) // 0x404
```

See also

DT_PHA_BY_IDX()

Parameters

- **node_id** – node identifier for a node with a dmas property
- **idx** – logical index into dmas property
- **cell** – lowercase-and-underscores cell name

Returns

the cell value at index "idx"

DT_INST_DMAS_CELL_BY_IDX(inst, idx, cell)

Get a DT_DRV_COMPAT instance's DMA specifier's cell value at an index.

See also

DT_DMAS_CELL_BY_IDX()

Parameters

- **inst** – DT_DRV_COMPAT instance number
- **idx** – logical index into dmas property

- **cell** – lowercase-and-underscores cell name

Returns
the cell value at index “idx”

DT_DMAS_CELL_BY_NAME(node_id, name, cell)

Get a DMA specifier’s cell value by name.

Example devicetree fragment:

```
dma1: dma@... {
        compatible = "vnd,dma";
        #dma-cells = <2>;
};

dma2: dma@... {
        compatible = "vnd,dma";
        #dma-cells = <2>;
};

n: node {
    dmas = <&dma1 1 0x400>,
           <&dma2 6 0x404>;
    dma-names = "tx", "rx";
};
```

Bindings fragment for the vnd,dma compatible:

```
dma-cells:
  - channel
  - config
```

Example usage:

```
DT_DMAS_CELL_BY_NAME(DT_NODELABEL(n), tx, channel) // 1
DT_DMAS_CELL_BY_NAME(DT_NODELABEL(n), rx, channel) // 6
DT_DMAS_CELL_BY_NAME(DT_NODELABEL(n), tx, config) // 0x400
DT_DMAS_CELL_BY_NAME(DT_NODELABEL(n), rx, config) // 0x404
```

See also

DT_PHA_BY_NAME()

Parameters

- **node_id** – node identifier for a node with a dmas property
- **name** – lowercase-and-underscores name of a dmas element as defined by the node’s dma-names property
- **cell** – lowercase-and-underscores cell name

Returns
the cell value in the specifier at the named element

DT_INST_DMAS_CELL_BY_NAME(inst, name, cell)

Get a DT_DRV_COMPAT instance’s DMA specifier’s cell value by name.

See also

DT_DMAS_CELL_BY_NAME()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `name` – lowercase-and-underscores name of a dmas element as defined by the node's dma-names property
- `cell` – lowercase-and-underscores cell name

Returns
the cell value in the specifier at the named element

`DT_DMAS_HAS_IDX`(node_id, idx)

Is index "idx" valid for a dmas property?

Parameters

- `node_id` – node identifier for a node with a dmas property
- `idx` – logical index into dmas property

Returns
1 if the "dmas" property has index "idx", 0 otherwise

`DT_INST_DMAS_HAS_IDX`(inst, idx)

Is index "idx" valid for a DT_DRV_COMPAT instance's dmas property?

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `idx` – logical index into dmas property

Returns
1 if the "dmas" property has a specifier at index "idx", 0 otherwise

`DT_DMAS_HAS_NAME`(node_id, name)

Does a dmas property have a named element?

Parameters

- `node_id` – node identifier for a node with a dmas property
- `name` – lowercase-and-underscores name of a dmas element as defined by the node's dma-names property

Returns
1 if the dmas property has the named element, 0 otherwise

`DT_INST_DMAS_HAS_NAME`(inst, name)

Does a DT_DRV_COMPAT instance's dmas property have a named element?

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `name` – lowercase-and-underscores name of a dmas element as defined by the node's dma-names property

Returns
1 if the dmas property has the named element, 0 otherwise

Fixed flash partitions These conveniences may be used for the special-purpose `fixed-partitions` compatible used to encode information about flash memory partitions in the device tree. See See `fixed-partition` for more details.

group `devicetree-fixed-partition`

Defines

`DT_NODE_BY_FIXED_PARTITION_LABEL`(label)

Get a node identifier for a fixed partition with a given label property.

Example devicetree fragment:

```
flash@... {
         partitions {
                 compatible = "fixed-partitions";
                 boot_partition: partition@0 {
                         label = "mcuboot";
                 };
                 slot0_partition: partition@c000 {
                         label = "image-0";
                 };
                 ...
         };
};
```

Example usage:

```
DT_NODE_BY_FIXED_PARTITION_LABEL(mcuboot) // node identifier for boot_partition
DT_NODE_BY_FIXED_PARTITION_LABEL(image_0) // node identifier for slot0_partition
```

Parameters

- `label` – lowercase-and-underscores label property value

Returns
a node identifier for the partition with that label property

`DT_HAS_FIXED_PARTITION_LABEL`(label)

Test if a fixed partition with a given label property exists.

Parameters

- `label` – lowercase-and-underscores label property value

Returns
1 if any “fixed-partitions” child node has the given label, 0 otherwise.

`DT_FIXED_PARTITION_EXISTS`(node_id)

Test if fixed-partition compatible node exists.

Parameters

- `node_id` – DTS node to test

Returns
1 if node exists and is fixed-partition compatible, 0 otherwise.

`DT_FIXED_PARTITION_ID`(node_id)

Get a numeric identifier for a fixed partition.

Parameters

- **node_id** – node identifier for a fixed-partitions child node

Returns
the partition's ID, a unique zero-based index number

DT_MEM_FROM_FIXED_PARTITION(node_id)

Get the node identifier of the flash memory for a partition.

Parameters

- **node_id** – node identifier for a fixed-partitions child node

Returns
the node identifier of the internal memory that contains the fixed-partitions node, or *DT_INVALID_NODE* if it doesn't exist.

DT_MTD_FROM_FIXED_PARTITION(node_id)

Get the node identifier of the flash controller for a partition.

Parameters

- **node_id** – node identifier for a fixed-partitions child node

Returns
the node identifier of the memory technology device that contains the fixed-partitions node.

DT_FIXED_PARTITION_ADDR(node_id)

Get the absolute address of a fixed partition.

Example devicetree fragment:

```
&flash_controller {
        flash@1000000 {
                compatible = "soc-nv-flash";
                partitions {
                        compatible = "fixed-partitions";
                        storage_partition: partition@3a000 {
                                label = "storage";
                        };
                };
        };
};
```

Here, the "storage" partition is seen to belong to flash memory starting at address 0x1000000. The partition's unit address of 0x3a000 represents an offset inside that flash memory.

Example usage:

```
DT_FIXED_PARTITION_ADDR(DT_NODELABEL(storage_partition)) // 0x103a000
```

This macro can only be used with partitions of internal memory addressable by the CPU. Otherwise, it may produce a compile-time error, such as: __REG_IDX_0_VAL_ADDRESS' undeclared'.

Parameters

- **node_id** – node identifier for a fixed-partitions child node

Returns
the partition's offset plus the base address of the flash node containing it.

GPIO These conveniences may be used for nodes which describe GPIO controllers/pins, and properties related to them.

group devicetree-gpio

Defines

DT_GPIO_CTLR_BY_IDX(node_id, gpio_pha, idx)

Get the node identifier for the controller phandle from a gpio phandle-array property at an index.

Example devicetree fragment:

```
gpio1: gpio@... { };

gpio2: gpio@... { };

n: node {
        gpios = <&gpio1 10 GPIO_ACTIVE_LOW>,
                <&gpio2 30 GPIO_ACTIVE_HIGH>;
};
```

Example usage:

```
DT_GPIO_CTLR_BY_IDX(DT_NODELABEL(n), gpios, 1) // DT_NODELABEL(gpio2)
```

See also

DT_PHANDLE_BY_IDX()

Parameters

- node_id – node identifier
- gpio_pha – lowercase-and-underscores GPIO property with type "phandle-array"
- idx – logical index into "gpio_pha"

Returns

the node identifier for the gpio controller referenced at index "idx"

DT_GPIO_CTLR(node_id, gpio_pha)

Equivalent to *DT_GPIO_CTLR_BY_IDX(node_id, gpio_pha, 0)*

See also

DT_GPIO_CTLR_BY_IDX()

Parameters

- node_id – node identifier
- gpio_pha – lowercase-and-underscores GPIO property with type "phandle-array"

Returns

a node identifier for the gpio controller at index 0 in "gpio_pha"

DT_GPIO_PIN_BY_IDX(node_id, gpio_pha, idx)

Get a GPIO specifier's pin cell at an index.

This macro only works for GPIO specifiers with cells named "pin". Refer to the node's binding to check if necessary.

Example devicetree fragment:

```
gpio1: gpio@... {
        compatible = "vnd,gpio";
        #gpio-cells = <2>;
};

gpio2: gpio@... {
        compatible = "vnd,gpio";
        #gpio-cells = <2>;
};

n: node {
        gpios = <&gpio1 10 GPIO_ACTIVE_LOW>,
                <&gpio2 30 GPIO_ACTIVE_HIGH>;
};
```

Bindings fragment for the vnd,gpio compatible:

```
gpio-cells:
  - pin
  - flags
```

Example usage:

```
DT_GPIO_PIN_BY_IDX(DT_NODELABEL(n), gpios, 0) // 10
DT_GPIO_PIN_BY_IDX(DT_NODELABEL(n), gpios, 1) // 30
```

See also

DT_PHA_BY_IDX()

Parameters

- **node_id** – node identifier
- **gpio_pha** – lowercase-and-underscores GPIO property with type "phandle-array"
- **idx** – logical index into "gpio_pha"

Returns

the pin cell value at index "idx"

DT_GPIO_PIN(node_id, gpio_pha)

Equivalent to *DT_GPIO_PIN_BY_IDX(node_id, gpio_pha, 0)*

See also

DT_GPIO_PIN_BY_IDX()

Parameters

- **node_id** – node identifier
- **gpio_pha** – lowercase-and-underscores GPIO property with type "phandle-array"

Returns
the pin cell value at index 0

DT_GPIO_FLAGS_BY_IDX(node_id, gpio_pha, idx)

Get a GPIO specifier's flags cell at an index.

This macro expects GPIO specifiers with cells named "flags". If there is no "flags" cell in the GPIO specifier, zero is returned. Refer to the node's binding to check specifier cell names if necessary.

Example devicetree fragment:

```
gpio1: gpio@... {
        compatible = "vnd,gpio";
        #gpio-cells = <2>;
};

gpio2: gpio@... {
        compatible = "vnd,gpio";
        #gpio-cells = <2>;
};

n: node {
        gpios = <&gpio1 10 GPIO_ACTIVE_LOW>,
                <&gpio2 30 GPIO_ACTIVE_HIGH>;
};
```

Bindings fragment for the vnd,gpio compatible:

```
gpio-cells:
  - pin
  - flags
```

Example usage:

```
DT_GPIO_FLAGS_BY_IDX(DT_NODELABEL(n), gpios, 0) // GPIO_ACTIVE_LOW
DT_GPIO_FLAGS_BY_IDX(DT_NODELABEL(n), gpios, 1) // GPIO_ACTIVE_HIGH
```

See also

DT_PHA_BY_IDX()

Parameters

- **node_id** – node identifier
- **gpio_pha** – lowercase-and-underscores GPIO property with type "phandle-array"
- **idx** – logical index into "gpio_pha"

Returns
the flags cell value at index "idx", or zero if there is none

DT_GPIO_FLAGS(node_id, gpio_pha)

Equivalent to *DT_GPIO_FLAGS_BY_IDX(node_id, gpio_pha, 0)*

See also

DT_GPIO_FLAGS_BY_IDX()

Parameters

- **node_id** – node identifier
- **gpio_pha** – lowercase-and-underscores GPIO property with type "phandle-array"

Returns
the flags cell value at index 0, or zero if there is none

DT_NUM_GPIO_HOGS(node_id)

Get the number of GPIO hogs in a node.

This expands to the number of hogged GPIOs, or zero if there are none.

Example devicetree fragment:

```
gpio1: gpio@... {
  compatible = "vnd,gpio";
  #gpio-cells = <2>;

  n1: node-1 {
          gpio-hog;
          gpios = <0 GPIO_ACTIVE_HIGH>, <1 GPIO_ACTIVE_LOW>;
          output-high;
  };

  n2: node-2 {
          gpio-hog;
          gpios = <3 GPIO_ACTIVE_HIGH>;
          output-low;
  };
};
```

Bindings fragment for the vnd,gpio compatible:

```
gpio-cells:
  - pin
  - flags
```

Example usage:

```
DT_NUM_GPIO_HOGS(DT_NODELABEL(n1)) // 2
DT_NUM_GPIO_HOGS(DT_NODELABEL(n2)) // 1
```

Parameters

- **node_id** – node identifier; may or may not be a GPIO hog node.

Returns
number of hogged GPIOs in the node

DT_GPIO_HOG_PIN_BY_IDX(node_id, idx)

Get a GPIO hog specifier's pin cell at an index.

This macro only works for GPIO specifiers with cells named "pin". Refer to the node's binding to check if necessary.

Example devicetree fragment:

```
gpio1: gpio@... {
  compatible = "vnd,gpio";
  #gpio-cells = <2>;

  n1: node-1 {
          gpio-hog;
          gpios = <0 GPIO_ACTIVE_HIGH>, <1 GPIO_ACTIVE_LOW>;
          output-high;
  };

  n2: node-2 {
          gpio-hog;
          gpios = <3 GPIO_ACTIVE_HIGH>;
          output-low;
  };
};
```

Bindings fragment for the vnd,gpio compatible:

```
gpio-cells:
  - pin
  - flags
```

Example usage:

```
DT_GPIO_HOG_PIN_BY_IDX(DT_NODELABEL(n1), 0) // 0
DT_GPIO_HOG_PIN_BY_IDX(DT_NODELABEL(n1), 1) // 1
DT_GPIO_HOG_PIN_BY_IDX(DT_NODELABEL(n2), 0) // 3
```

Parameters

- `node_id` – node identifier
- `idx` – logical index into “gpios”

Returns
the pin cell value at index “idx”

`DT_GPIO_HOG_FLAGS_BY_IDX`(node_id, idx)

Get a GPIO hog specifier’s flags cell at an index.

This macro expects GPIO specifiers with cells named “flags”. If there is no “flags” cell in the GPIO specifier, zero is returned. Refer to the node’s binding to check specifier cell names if necessary.

Example devicetree fragment:

```
gpio1: gpio@... {
  compatible = "vnd,gpio";
  #gpio-cells = <2>;

  n1: node-1 {
          gpio-hog;
          gpios = <0 GPIO_ACTIVE_HIGH>, <1 GPIO_ACTIVE_LOW>;
          output-high;
  };

  n2: node-2 {
          gpio-hog;
          gpios = <3 GPIO_ACTIVE_HIGH>;
          output-low;
  };
};
```

Bindings fragment for the vnd,gpio compatible:

```
gpio-cells:
  - pin
  - flags
```

Example usage:

```
DT_GPIO_HOG_FLAGS_BY_IDX(DT_NODELABEL(n1), 0) // GPIO_ACTIVE_HIGH
DT_GPIO_HOG_FLAGS_BY_IDX(DT_NODELABEL(n1), 1) // GPIO_ACTIVE_LOW
DT_GPIO_HOG_FLAGS_BY_IDX(DT_NODELABEL(n2), 0) // GPIO_ACTIVE_HIGH
```

Parameters

- `node_id` – node identifier
- `idx` – logical index into “gpios”

Returns
the flags cell value at index “idx”, or zero if there is none

`DT_INST_GPIO_PIN_BY_IDX`(inst, gpio_pha, idx)

Get a DT_DRV_COMPAT instance’s GPIO specifier’s pin cell value at an index.

See also

DT_GPIO_PIN_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `gpio_pha` – lowercase-and-underscores GPIO property with type “phandle-array”
- `idx` – logical index into “gpio_pha”

Returns
the pin cell value at index “idx”

`DT_INST_GPIO_PIN`(inst, gpio_pha)

Equivalent to *DT_INST_GPIO_PIN_BY_IDX(inst, gpio_pha, 0)*

See also

DT_INST_GPIO_PIN_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `gpio_pha` – lowercase-and-underscores GPIO property with type “phandle-array”

Returns
the pin cell value at index 0

DT_INST_GPIO_FLAGS_BY_IDX(inst, gpio_pha, idx)

Get a DT_DRV_COMPAT instance's GPIO specifier's flags cell at an index.

See also

DT_GPIO_FLAGS_BY_IDX()

Parameters

- **inst** – DT_DRV_COMPAT instance number
- **gpio_pha** – lowercase-and-underscores GPIO property with type "phandle-array"
- **idx** – logical index into "gpio_pha"

Returns

the flags cell value at index "idx", or zero if there is none

DT_INST_GPIO_FLAGS(inst, gpio_pha)

Equivalent to *DT_INST_GPIO_FLAGS_BY_IDX(inst, gpio_pha, 0)*

See also

DT_INST_GPIO_FLAGS_BY_IDX()

Parameters

- **inst** – DT_DRV_COMPAT instance number
- **gpio_pha** – lowercase-and-underscores GPIO property with type "phandle-array"

Returns

the flags cell value at index 0, or zero if there is none

IO channels These are commonly used by device drivers which need to use IO channels (e.g. ADC or DAC channels) for conversion.

group devicetree-io-channels

Defines

DT_IO_CHANNELS_CTLR_BY_IDX(node_id, idx)

Get the node identifier for the node referenced by an io-channels property at an index.

Example devicetree fragment:

```
adc1: adc@... { ... };

adc2: adc@... { ... };

n: node {
```

(continues on next page)

(continued from previous page)

```
        io-channels = <&adc1 10>, <&adc2 20>;
};
```

Example usage:

```
DT_IO_CHANNELS_CTLR_BY_IDX(DT_NODELABEL(n), 0) // DT_NODELABEL(adc1)
DT_IO_CHANNELS_CTLR_BY_IDX(DT_NODELABEL(n), 1) // DT_NODELABEL(adc2)
```

See also

DT_PROP_BY_PHANDLE_IDX()

Parameters

- **node_id** – node identifier for a node with an io-channels property
- **idx** – logical index into io-channels property

Returns
the node identifier for the node referenced at index “idx”

`DT_IO_CHANNELS_CTLR_BY_NAME`(node_id, name)

Get the node identifier for the node referenced by an io-channels property by name.

Example devicetree fragment:

```
adc1: adc@... { ... };

adc2: adc@... { ... };

n: node {
        io-channels = <&adc1 10>, <&adc2 20>;
        io-channel-names = "SENSOR", "BANDGAP";
};
```

Example usage:

DT_IO_CHANNELS_CTLR_BY_NAME(DT_NODELABEL(n), sensor) // *DT_NODELABEL(adc1)* *DT_IO_CHANNELS_CTLR_BY_NAME(DT_NODELABEL(n), bandgap)* // *DT_NODELABEL(adc2)*

See also

DT_PHANDLE_BY_NAME()

Parameters

- **node_id** – node identifier for a node with an io-channels property
- **name** – lowercase-and-underscores name of an io-channels element as defined by the node’s io-channel-names property

Returns
the node identifier for the node referenced at the named element

DT_IO_CHANNELS_CTLR(node_id)

Equivalent to *DT_IO_CHANNELS_CTLR_BY_IDX(node_id, 0)*

See also

DT_IO_CHANNELS_CTLR_BY_IDX()

Parameters

- **node_id** – node identifier for a node with an io-channels property

Returns

the node identifier for the node referenced at index 0 in the node's "io-channels" property

DT_INST_IO_CHANNELS_CTLR_BY_IDX(inst, idx)

Get the node identifier from a DT_DRV_COMPAT instance's io-channels property at an index.

See also

DT_IO_CHANNELS_CTLR_BY_IDX()

Parameters

- **inst** – DT_DRV_COMPAT instance number
- **idx** – logical index into io-channels property

Returns

the node identifier for the node referenced at index "idx"

DT_INST_IO_CHANNELS_CTLR_BY_NAME(inst, name)

Get the node identifier from a DT_DRV_COMPAT instance's io-channels property by name.

See also

DT_IO_CHANNELS_CTLR_BY_NAME()

Parameters

- **inst** – DT_DRV_COMPAT instance number
- **name** – lowercase-and-underscores name of an io-channels element as defined by the node's io-channel-names property

Returns

the node identifier for the node referenced at the named element

DT_INST_IO_CHANNELS_CTLR(inst)

Equivalent to *DT_INST_IO_CHANNELS_CTLR_BY_IDX(inst, 0)*

See also

DT_IO_CHANNELS_CTLR_BY_IDX()

Parameters

- **inst** – DT_DRV_COMPAT instance number

Returns

the node identifier for the node referenced at index 0 in the node's "io-channels" property

DT_IO_CHANNELS_INPUT_BY_IDX(node_id, idx)

Get an io-channels specifier input cell at an index.

This macro only works for io-channels specifiers with cells named "input". Refer to the node's binding to check if necessary.

Example devicetree fragment:

```
adc1: adc@... {
        compatible = "vnd,adc";
        #io-channel-cells = <1>;
};

adc2: adc@... {
        compatible = "vnd,adc";
        #io-channel-cells = <1>;
};

n: node {
        io-channels = <&adc1 10>, <&adc2 20>;
};
```

Bindings fragment for the vnd,adc compatible:

io-channel-cells:

- input

Example usage:

```
DT_IO_CHANNELS_INPUT_BY_IDX(DT_NODELABEL(n), 0) // 10
DT_IO_CHANNELS_INPUT_BY_IDX(DT_NODELABEL(n), 1) // 20
```

See also

DT_PHA_BY_IDX()

Parameters

- **node_id** – node identifier for a node with an io-channels property
- **idx** – logical index into io-channels property

Returns

the input cell in the specifier at index "idx"

DT_IO_CHANNELS_INPUT_BY_NAME(node_id, name)

Get an io-channels specifier input cell by name.

This macro only works for io-channels specifiers with cells named "input". Refer to the node's binding to check if necessary.

Example devicetree fragment:

```
adc1: adc@... {
        compatible = "vnd,adc";
        #io-channel-cells = <1>;
};

adc2: adc@... {
        compatible = "vnd,adc";
        #io-channel-cells = <1>;
};

n: node {
        io-channels = <&adc1 10>, <&adc2 20>;
        io-channel-names = "SENSOR", "BANDGAP";
};
```

Bindings fragment for the vnd,adc compatible:

io-channel-cells:

- input

Example usage:

```
DT_IO_CHANNELS_INPUT_BY_NAME(DT_NODELABEL(n), sensor) // 10
DT_IO_CHANNELS_INPUT_BY_NAME(DT_NODELABEL(n), bandgap) // 20
```

See also

DT_PHA_BY_NAME()

Parameters

- **node_id** – node identifier for a node with an io-channels property
- **name** – lowercase-and-underscores name of an io-channels element as defined by the node's io-channel-names property

Returns

the input cell in the specifier at the named element

DT_IO_CHANNELS_INPUT(node_id)

Equivalent to *DT_IO_CHANNELS_INPUT_BY_IDX(node_id, 0)*

See also

DT_IO_CHANNELS_INPUT_BY_IDX()

Parameters

- **node_id** – node identifier for a node with an io-channels property

Returns
the input cell in the specifier at index 0

`DT_INST_IO_CHANNELS_INPUT_BY_IDX`(inst, idx)
Get an input cell from the “DT_DRV_INST(inst)” io-channels property at an index.

See also
DT_IO_CHANNELS_INPUT_BY_IDX()

Parameters
- `inst` – DT_DRV_COMPAT instance number
- `idx` – logical index into io-channels property

Returns
the input cell in the specifier at index “idx”

`DT_INST_IO_CHANNELS_INPUT_BY_NAME`(inst, name)
Get an input cell from the “DT_DRV_INST(inst)” io-channels property by name.

See also
DT_IO_CHANNELS_INPUT_BY_NAME()

Parameters
- `inst` – DT_DRV_COMPAT instance number
- `name` – lowercase-and-underscores name of an io-channels element as defined by the instance’s io-channel-names property

Returns
the input cell in the specifier at the named element

`DT_INST_IO_CHANNELS_INPUT`(inst)
Equivalent to *DT_INST_IO_CHANNELS_INPUT_BY_IDX(inst, 0)*

Parameters
- `inst` – DT_DRV_COMPAT instance number

Returns
the input cell in the specifier at index 0

MBOX These conveniences may be used for nodes which describe MBOX controllers/users, and properties related to them.

group `devicetree-mbox`

Defines

DT_MBOX_CTLR_BY_NAME(node_id, name)

Get the node identifier for the MBOX controller from a mboxes property by name.

Example devicetree fragment:

```
mbox1: mbox-controller@... { ... };

n: node {
        mboxes = <&mbox1 8>,
                 <&mbox1 9>;
        mbox-names = "tx", "rx";
};
```

Example usage:

```
DT_MBOX_CTLR_BY_NAME(DT_NODELABEL(n), tx) // DT_NODELABEL(mbox1)
DT_MBOX_CTLR_BY_NAME(DT_NODELABEL(n), rx) // DT_NODELABEL(mbox1)
```

See also

DT_PHANDLE_BY_NAME()

Parameters

- **node_id** – node identifier for a node with a mboxes property
- **name** – lowercase-and-underscores name of a mboxes element as defined by the node's mbox-names property

Returns

the node identifier for the MBOX controller in the named element

DT_MBOX_CHANNEL_BY_NAME(node_id, name)

Get a MBOX channel value by name.

Example devicetree fragment:

```
mbox1: mbox@... {
        #mbox-cells = <1>;
};

n: node {
    mboxes = <&mbox1 1>,
             <&mbox1 6>;
    mbox-names = "tx", "rx";
};
```

Bindings fragment for the mbox compatible:

```
mbox-cells:
  - channel
```

Example usage:

```
DT_MBOX_CHANNEL_BY_NAME(DT_NODELABEL(n), tx) // 1
DT_MBOX_CHANNEL_BY_NAME(DT_NODELABEL(n), rx) // 6
```

See also

DT_PHA_BY_NAME_OR()

Parameters

- **node_id** – node identifier for a node with a mboxes property
- **name** – lowercase-and-underscores name of a mboxes element as defined by the node's mbox-names property

Returns

the channel value in the specifier at the named element or 0 if no channels are supported

Pinctrl (pin control) These are used to access pin control properties by name or index.

Devicetree nodes may have properties which specify pin control (sometimes known as pin mux) settings. These are expressed using `pinctrl-<index>` properties within the node, where the `<index>` values are contiguous integers starting from 0. These may also be named using the `pinctrl-names` property.

Here is an example:

```
node {
    ...
    pinctrl-0 = <&foo &bar ...>;
    pinctrl-1 = <&baz ...>;
    pinctrl-names = "default", "sleep";
};
```

Above, `pinctrl-0` has name `"default"`, and `pinctrl-1` has name `"sleep"`. The `pinctrl-<index>` property values contain phandles. The `&foo`, `&bar`, etc. phandles within the properties point to nodes whose contents vary by platform, and which describe a pin configuration for the node.

group **devicetree-pinctrl**

Defines

`DT_PINCTRL_BY_IDX`(node_id, pc_idx, idx)

Get a node identifier for a phandle in a pinctrl property by index.

Example devicetree fragment:

```
n: node {
        pinctrl-0 = <&foo &bar>;
        pinctrl-1 = <&baz &blub>;
}
```

Example usage:

```
DT_PINCTRL_BY_IDX(DT_NODELABEL(n), 0, 1) // DT_NODELABEL(bar)
DT_PINCTRL_BY_IDX(DT_NODELABEL(n), 1, 0) // DT_NODELABEL(baz)
```

Parameters

- **node_id** – node with a pinctrl-'pc_idx' property
- **pc_idx** – index of the pinctrl property itself

- **idx** – index into the value of the pinctrl property

Returns
node identifier for the phandle at index 'idx' in 'pinctrl-'pc_idx''

DT_PINCTRL_0(node_id, idx)

Get a node identifier from a pinctrl-0 property.

This is equivalent to:

```
DT_PINCTRL_BY_IDX(node_id, 0, idx)
```

It is provided for convenience since pinctrl-0 is commonly used.

Parameters

- **node_id** – node with a pinctrl-0 property
- **idx** – index into the pinctrl-0 property

Returns
node identifier for the phandle at index idx in the pinctrl-0 property of that node

DT_PINCTRL_BY_NAME(node_id, name, idx)

Get a node identifier for a phandle inside a pinctrl node by name.

Example devicetree fragment:

```
n: node {
        pinctrl-0 = <&foo &bar>;
        pinctrl-1 = <&baz &blub>;
        pinctrl-names = "default", "sleep";
};
```

Example usage:

```
DT_PINCTRL_BY_NAME(DT_NODELABEL(n), default, 1) // DT_NODELABEL(bar)
DT_PINCTRL_BY_NAME(DT_NODELABEL(n), sleep, 0) // DT_NODELABEL(baz)
```

Parameters

- **node_id** – node with a named pinctrl property
- **name** – lowercase-and-underscores pinctrl property name
- **idx** – index into the value of the named pinctrl property

Returns
node identifier for the phandle at that index in the pinctrl property

DT_PINCTRL_NAME_TO_IDX(node_id, name)

Convert a pinctrl name to its corresponding index.

Example devicetree fragment:

```
n: node {
        pinctrl-0 = <&foo &bar>;
        pinctrl-1 = <&baz &blub>;
        pinctrl-names = "default", "sleep";
};
```

Example usage:

```
DT_PINCTRL_NAME_TO_IDX(DT_NODELABEL(n), default) // 0
DT_PINCTRL_NAME_TO_IDX(DT_NODELABEL(n), sleep)   // 1
```

Parameters

- **node_id** – node identifier with a named pinctrl property
- **name** – lowercase-and-underscores name name of the pinctrl whose index to get

Returns
integer literal for the index of the pinctrl property with that name

DT_PINCTRL_IDX_TO_NAME_TOKEN(node_id, pc_idx)

Convert a pinctrl property index to its name as a token.

This allows you to get a pinctrl property's name, and "remove the

quotes" from it.

DT_PINCTRL_IDX_TO_NAME_TOKEN() can only be used if the node has a pinctrl-'pc_idx' property and a pinctrl-names property element for that index. It is an error to use it in other circumstances.

Example devicetree fragment:

```
n: node {
        pinctrl-0 = <...>;
        pinctrl-1 = <...>;
        pinctrl-names = "default", "f.o.o2";
};
```

Example usage:

```
DT_PINCTRL_IDX_TO_NAME_TOKEN(DT_NODELABEL(n), 0) // default
DT_PINCTRL_IDX_TO_NAME_TOKEN(DT_NODELABEL(n), 1) // f_o_o2
```

The same caveats and restrictions that apply to *DT_STRING_TOKEN()*'s return value also apply here.

Parameters

- **node_id** – node identifier
- **pc_idx** – index of a pinctrl property in that node

Returns
name of the pinctrl property, as a token, without any quotes and with non-alphanumeric characters converted to underscores

DT_PINCTRL_IDX_TO_NAME_UPPER_TOKEN(node_id, pc_idx)

Like *DT_PINCTRL_IDX_TO_NAME_TOKEN()*, but with an uppercased result.

This does the a similar conversion as *DT_PINCTRL_IDX_TO_NAME_TOKEN(node_id, pc_idx)*. The only difference is that alphabetical characters in the result are uppercased.

Example devicetree fragment:

```
n: node {
        pinctrl-0 = <...>;
        pinctrl-1 = <...>;
        pinctrl-names = "default", "f.o.o2";
};
```

Example usage:

```
DT_PINCTRL_IDX_TO_NAME_TOKEN(DT_NODELABEL(n), 0) // DEFAULT
DT_PINCTRL_IDX_TO_NAME_TOKEN(DT_NODELABEL(n), 1) // F_0_02
```

The same caveats and restrictions that apply to *DT_STRING_UPPER_TOKEN()*'s return value also apply here.

`DT_NUM_PINCTRLS_BY_IDX`(node_id, pc_idx)

Get the number of phandles in a pinctrl property.

Example devicetree fragment:

```
n1: node-1 {
        pinctrl-0 = <&foo &bar>;
};

n2: node-2 {
        pinctrl-0 = <&baz>;
};
```

Example usage:

```
DT_NUM_PINCTRLS_BY_IDX(DT_NODELABEL(n1), 0) // 2
DT_NUM_PINCTRLS_BY_IDX(DT_NODELABEL(n2), 0) // 1
```

Parameters

- `node_id` – node identifier with a pinctrl property
- `pc_idx` – index of the pinctrl property itself

Returns
number of phandles in the property with that index

`DT_NUM_PINCTRLS_BY_NAME`(node_id, name)

Like *DT_NUM_PINCTRLS_BY_IDX()*, but by name instead.

Example devicetree fragment:

```
n: node {
        pinctrl-0 = <&foo &bar>;
        pinctrl-1 = <&baz>
        pinctrl-names = "default", "sleep";
};
```

Example usage:

```
DT_NUM_PINCTRLS_BY_NAME(DT_NODELABEL(n), default) // 2
DT_NUM_PINCTRLS_BY_NAME(DT_NODELABEL(n), sleep)   // 1
```

Parameters

- `node_id` – node identifier with a pinctrl property
- `name` – lowercase-and-underscores name name of the pinctrl property

Returns
number of phandles in the property with that name

`DT_NUM_PINCTRL_STATES`(node_id)

Get the number of pinctrl properties in a node.

This expands to 0 if there are no pinctrl-i properties. Otherwise, it expands to the number of such properties.

Example devicetree fragment:

```
n1: node-1 {
        pinctrl-0 = <...>;
        pinctrl-1 = <...>;
};

n2: node-2 {
};
```

Example usage:

```
DT_NUM_PINCTRL_STATES(DT_NODELABEL(n1)) // 2
DT_NUM_PINCTRL_STATES(DT_NODELABEL(n2)) // 0
```

Parameters

- **node_id** – node identifier; may or may not have any pinctrl properties

Returns
number of pinctrl properties in the node

DT_PINCTRL_HAS_IDX(node_id, pc_idx)

Test if a node has a pinctrl property with an index.

This expands to 1 if the pinctrl-'idx' property exists. Otherwise, it expands to 0.

Example devicetree fragment:

```
n1: node-1 {
        pinctrl-0 = <...>;
        pinctrl-1 = <...>;
};

n2: node-2 {
};
```

Example usage:

```
DT_PINCTRL_HAS_IDX(DT_NODELABEL(n1), 0) // 1
DT_PINCTRL_HAS_IDX(DT_NODELABEL(n1), 1) // 1
DT_PINCTRL_HAS_IDX(DT_NODELABEL(n1), 2) // 0
DT_PINCTRL_HAS_IDX(DT_NODELABEL(n2), 0) // 0
```

Parameters

- **node_id** – node identifier; may or may not have any pinctrl properties
- **pc_idx** – index of a pinctrl property whose existence to check

Returns
1 if the property exists, 0 otherwise

DT_PINCTRL_HAS_NAME(node_id, name)

Test if a node has a pinctrl property with a name.

This expands to 1 if the named pinctrl property exists. Otherwise, it expands to 0.

Example devicetree fragment:

```
n1: node-1 {
        pinctrl-0 = <...>;
        pinctrl-names = "default";
};
```

(continues on next page)

(continued from previous page)

```
n2: node-2 {
};
```

Example usage:

```
DT_PINCTRL_HAS_NAME(DT_NODELABEL(n1), default) // 1
DT_PINCTRL_HAS_NAME(DT_NODELABEL(n1), sleep)   // 0
DT_PINCTRL_HAS_NAME(DT_NODELABEL(n2), default) // 0
```

Parameters

- **node_id** – node identifier; may or may not have any pinctrl properties
- **name** – lowercase-and-underscores pinctrl property name to check

Returns
1 if the property exists, 0 otherwise

`DT_INST_PINCTRL_BY_IDX`(inst, pc_idx, idx)

Get a node identifier for a phandle in a pinctrl property by index for a DT_DRV_COMPAT instance.

This is equivalent to *DT_PINCTRL_BY_IDX(DT_DRV_INST(inst), pc_idx, idx)*.

Parameters

- **inst** – instance number
- **pc_idx** – index of the pinctrl property itself
- **idx** – index into the value of the pinctrl property

Returns
node identifier for the phandle at index ‘idx’ in ‘pinctrl-‘pc_idx”

`DT_INST_PINCTRL_0`(inst, idx)

Get a node identifier from a pinctrl-0 property for a DT_DRV_COMPAT instance.

This is equivalent to:

```
DT_PINCTRL_BY_IDX(DT_DRV_INST(inst), 0, idx)
```

It is provided for convenience since pinctrl-0 is commonly used.

Parameters

- **inst** – instance number
- **idx** – index into the pinctrl-0 property

Returns
node identifier for the phandle at index idx in the pinctrl-0 property of that instance

`DT_INST_PINCTRL_BY_NAME`(inst, name, idx)

Get a node identifier for a phandle inside a pinctrl node for a DT_DRV_COMPAT instance.

This is equivalent to *DT_PINCTRL_BY_NAME(DT_DRV_INST(inst), name, idx)*.

Parameters

- **inst** – instance number
- **name** – lowercase-and-underscores pinctrl property name

- `idx` – index into the value of the named pinctrl property

Returns
node identifier for the phandle at that index in the pinctrl property

`DT_INST_PINCTRL_NAME_TO_IDX`(inst, name)

Convert a pinctrl name to its corresponding index for a DT_DRV_COMPAT instance.

This is equivalent to *DT_PINCTRL_NAME_TO_IDX(DT_DRV_INST(inst),name)*.

Parameters

- `inst` – instance number
- `name` – lowercase-and-underscores name of the pinctrl whose index to get

Returns
integer literal for the index of the pinctrl property with that name

`DT_INST_PINCTRL_IDX_TO_NAME_TOKEN`(inst, pc_idx)

Convert a pinctrl index to its name as an uppercased token.

This is equivalent to *DT_PINCTRL_IDX_TO_NAME_TOKEN(DT_DRV_INST(inst), pc_idx)*.

Parameters

- `inst` – instance number
- `pc_idx` – index of the pinctrl property itself

Returns
name of the pin control property as a token

`DT_INST_PINCTRL_IDX_TO_NAME_UPPER_TOKEN`(inst, pc_idx)

Convert a pinctrl index to its name as an uppercased token.

This is equivalent to *DT_PINCTRL_IDX_TO_NAME_UPPER_TOKEN(DT_DRV_INST(inst), idx)*.

Parameters

- `inst` – instance number
- `pc_idx` – index of the pinctrl property itself

Returns
name of the pin control property as an uppercase token

`DT_INST_NUM_PINCTRLS_BY_IDX`(inst, pc_idx)

Get the number of phandles in a pinctrl property for a DT_DRV_COMPAT instance.

This is equivalent to *DT_NUM_PINCTRLS_BY_IDX(DT_DRV_INST(inst),pc_idx)*.

Parameters

- `inst` – instance number
- `pc_idx` – index of the pinctrl property itself

Returns
number of phandles in the property with that index

`DT_INST_NUM_PINCTRLS_BY_NAME`(inst, name)

Like *DT_INST_NUM_PINCTRLS_BY_IDX()*, but by name instead.

This is equivalent to *DT_NUM_PINCTRLS_BY_NAME(DT_DRV_INST(inst),name)*.

Parameters

- `inst` – instance number
- `name` – lowercase-and-underscores name of the pinctrl property

Returns
number of phandles in the property with that name

`DT_INST_NUM_PINCTRL_STATES`(inst)
Get the number of pinctrl properties in a DT_DRV_COMPAT instance.

This is equivalent to *DT_NUM_PINCTRL_STATES(DT_DRV_INST(inst))*.

Parameters
- `inst` – instance number

Returns
number of pinctrl properties in the instance

`DT_INST_PINCTRL_HAS_IDX`(inst, pc_idx)
Test if a DT_DRV_COMPAT instance has a pinctrl property with an index.

This is equivalent to *DT_PINCTRL_HAS_IDX(DT_DRV_INST(inst), pc_idx)*.

Parameters
- `inst` – instance number
- `pc_idx` – index of a pinctrl property whose existence to check

Returns
1 if the property exists, 0 otherwise

`DT_INST_PINCTRL_HAS_NAME`(inst, name)
Test if a DT_DRV_COMPAT instance has a pinctrl property with a name.

This is equivalent to *DT_PINCTRL_HAS_NAME(DT_DRV_INST(inst), name)*.

Parameters
- `inst` – instance number
- `name` – lowercase-and-underscores pinctrl property name to check

Returns
1 if the property exists, 0 otherwise

PWM These conveniences may be used for nodes which describe PWM controllers and properties related to them.

group `devicetree-pwms`

Defines

`DT_PWMS_CTLR_BY_IDX`(node_id, idx)
Get the node identifier for the PWM controller from a pwms property at an index.

Example devicetree fragment:

```
pwm1: pwm-controller@... { ... };

pwm2: pwm-controller@... { ... };

n: node {
        pwms = <&pwm1 1 PWM_POLARITY_NORMAL>,
               <&pwm2 3 PWM_POLARITY_INVERTED>;
};
```

Example usage:

```
DT_PWMS_CTLR_BY_IDX(DT_NODELABEL(n), 0) // DT_NODELABEL(pwm1)
DT_PWMS_CTLR_BY_IDX(DT_NODELABEL(n), 1) // DT_NODELABEL(pwm2)
```

See also

DT_PROP_BY_PHANDLE_IDX()

Parameters

- **node_id** – node identifier for a node with a pwms property
- **idx** – logical index into pwms property

Returns
the node identifier for the PWM controller referenced at index “idx”

DT_PWMS_CTLR_BY_NAME(node_id, name)

Get the node identifier for the PWM controller from a pwms property by name.

Example devicetree fragment:

```
pwm1: pwm-controller@... { ... };
```

pwm2: pwm-controller... { ... };

n: node { pwms = <&pwm1 1 PWM_POLARITY_NORMAL>, <&pwm2 3 PWM_POLARITY_INVERTED>; pwm-names = “alpha”, “beta”; };

Example usage:

```
DT_PWMS_CTLR_BY_NAME(DT_NODELABEL(n), alpha) // DT_NODELABEL(pwm1)
DT_PWMS_CTLR_BY_NAME(DT_NODELABEL(n), beta)  // DT_NODELABEL(pwm2)
```

See also

DT_PHANDLE_BY_NAME()

Parameters

- **node_id** – node identifier for a node with a pwms property
- **name** – lowercase-and-underscores name of a pwms element as defined by the node’s pwm-names property

Returns
the node identifier for the PWM controller in the named element

DT_PWMS_CTLR(node_id)

Equivalent to *DT_PWMS_CTLR_BY_IDX(node_id, 0)*

See also

DT_PWMS_CTLR_BY_IDX()

Parameters

- **node_id** – node identifier for a node with a pwms property

Returns
the node identifier for the PWM controller at index 0 in the node's "pwms" property

`DT_PWMS_CELL_BY_IDX`(node_id, idx, cell)

Get PWM specifier's cell value at an index.

Example devicetree fragment:

```
pwm1: pwm-controller@... {
        compatible = "vnd,pwm";
        #pwm-cells = <2>;
};

pwm2: pwm-controller@... {
        compatible = "vnd,pwm";
        #pwm-cells = <2>;
};

n: node {
        pwms = <&pwm1 1 200000 PWM_POLARITY_NORMAL>,
               <&pwm2 3 100000 PWM_POLARITY_INVERTED>;
};
```

Bindings fragment for the "vnd,pwm" compatible:

```
pwm-cells:
  - channel
  - period
  - flags
```

Example usage:

```
DT_PWMS_CELL_BY_IDX(DT_NODELABEL(n), 0, channel) // 1
DT_PWMS_CELL_BY_IDX(DT_NODELABEL(n), 1, channel) // 3
DT_PWMS_CELL_BY_IDX(DT_NODELABEL(n), 0, period)  // 200000
DT_PWMS_CELL_BY_IDX(DT_NODELABEL(n), 1, period)  // 100000
DT_PWMS_CELL_BY_IDX(DT_NODELABEL(n), 0, flags)   // PWM_POLARITY_NORMAL
DT_PWMS_CELL_BY_IDX(DT_NODELABEL(n), 1, flags)   // PWM_POLARITY_INVERTED
```

See also

DT_PHA_BY_IDX()

Parameters

- `node_id` – node identifier for a node with a pwms property
- `idx` – logical index into pwms property
- `cell` – lowercase-and-underscores cell name

Returns
the cell value at index "idx"

`DT_PWMS_CELL_BY_NAME`(node_id, name, cell)

Get a PWM specifier's cell value by name.

Example devicetree fragment:

```
pwm1: pwm-controller@... {
        compatible = "vnd,pwm";
        #pwm-cells = <2>;
};

pwm2: pwm-controller@... {
        compatible = "vnd,pwm";
        #pwm-cells = <2>;
};

n: node {
        pwms = <&pwm1 1 200000 PWM_POLARITY_NORMAL>,
               <&pwm2 3 100000 PWM_POLARITY_INVERTED>;
        pwm-names = "alpha", "beta";
};
```

Bindings fragment for the “vnd,pwm” compatible:

```
pwm-cells:
  - channel
  - period
  - flags
```

Example usage:

```
DT_PWMS_CELL_BY_NAME(DT_NODELABEL(n), alpha, channel) // 1
DT_PWMS_CELL_BY_NAME(DT_NODELABEL(n), beta, channel)  // 3
DT_PWMS_CELL_BY_NAME(DT_NODELABEL(n), alpha, period)  // 200000
DT_PWMS_CELL_BY_NAME(DT_NODELABEL(n), beta, period)   // 100000
DT_PWMS_CELL_BY_NAME(DT_NODELABEL(n), alpha, flags)   // PWM_POLARITY_NORMAL
DT_PWMS_CELL_BY_NAME(DT_NODELABEL(n), beta, flags)    // PWM_POLARITY_INVERTED
```

See also

DT_PHA_BY_NAME()

Parameters

- **node_id** – node identifier for a node with a pwms property
- **name** – lowercase-and-underscores name of a pwms element as defined by the node’s pwm-names property
- **cell** – lowercase-and-underscores cell name

Returns
the cell value in the specifier at the named element

DT_PWMS_CELL(node_id, cell)

Equivalent to *DT_PWMS_CELL_BY_IDX(node_id, 0, cell)*

See also

DT_PWMS_CELL_BY_IDX()

Parameters

- **node_id** – node identifier for a node with a pwms property

- `cell` – lowercase-and-underscores cell name

Returns
the cell value at index 0

`DT_PWMS_CHANNEL_BY_IDX`(node_id, idx)

Get a PWM specifier's channel cell value at an index.

This macro only works for PWM specifiers with cells named "channel". Refer to the node's binding to check if necessary.

This is equivalent to *DT_PWMS_CELL_BY_IDX(node_id, idx, channel)*.

See also

DT_PWMS_CELL_BY_IDX()

Parameters

- `node_id` – node identifier for a node with a pwms property
- `idx` – logical index into pwms property

Returns
the channel cell value at index "idx"

`DT_PWMS_CHANNEL_BY_NAME`(node_id, name)

Get a PWM specifier's channel cell value by name.

This macro only works for PWM specifiers with cells named "channel". Refer to the node's binding to check if necessary.

This is equivalent to *DT_PWMS_CELL_BY_NAME(node_id, name, channel)*.

See also

DT_PWMS_CELL_BY_NAME()

Parameters

- `node_id` – node identifier for a node with a pwms property
- `name` – lowercase-and-underscores name of a pwms element as defined by the node's pwm-names property

Returns
the channel cell value in the specifier at the named element

`DT_PWMS_CHANNEL`(node_id)

Equivalent to *DT_PWMS_CHANNEL_BY_IDX(node_id, 0)*

See also

DT_PWMS_CHANNEL_BY_IDX()

Parameters

- `node_id` – node identifier for a node with a pwms property

Returns
the channel cell value at index 0

`DT_PWMS_PERIOD_BY_IDX`(node_id, idx)

Get PWM specifier's period cell value at an index.

This macro only works for PWM specifiers with cells named "period". Refer to the node's binding to check if necessary.

This is equivalent to *DT_PWMS_CELL_BY_IDX(node_id, idx, period)*.

See also

DT_PWMS_CELL_BY_IDX()

Parameters

- `node_id` – node identifier for a node with a pwms property
- `idx` – logical index into pwms property

Returns
the period cell value at index "idx"

`DT_PWMS_PERIOD_BY_NAME`(node_id, name)

Get a PWM specifier's period cell value by name.

This macro only works for PWM specifiers with cells named "period". Refer to the node's binding to check if necessary.

This is equivalent to *DT_PWMS_CELL_BY_NAME(node_id, name, period)*.

See also

DT_PWMS_CELL_BY_NAME()

Parameters

- `node_id` – node identifier for a node with a pwms property
- `name` – lowercase-and-underscores name of a pwms element as defined by the node's pwm-names property

Returns
the period cell value in the specifier at the named element

`DT_PWMS_PERIOD`(node_id)

Equivalent to *DT_PWMS_PERIOD_BY_IDX(node_id, 0)*

See also

DT_PWMS_PERIOD_BY_IDX()

Parameters

- `node_id` – node identifier for a node with a pwms property

Returns
the period cell value at index 0

`DT_PWMS_FLAGS_BY_IDX`(node_id, idx)

Get a PWM specifier's flags cell value at an index.

This macro expects PWM specifiers with cells named "flags". If there is no "flags" cell in the PWM specifier, zero is returned. Refer to the node's binding to check specifier cell names if necessary.

This is equivalent to *DT_PWMS_CELL_BY_IDX(node_id, idx, flags)*.

See also

DT_PWMS_CELL_BY_IDX()

Parameters

- `node_id` – node identifier for a node with a pwms property
- `idx` – logical index into pwms property

Returns
the flags cell value at index "idx", or zero if there is none

`DT_PWMS_FLAGS_BY_NAME`(node_id, name)

Get a PWM specifier's flags cell value by name.

This macro expects PWM specifiers with cells named "flags". If there is no "flags" cell in the PWM specifier, zero is returned. Refer to the node's binding to check specifier cell names if necessary.

This is equivalent to *DT_PWMS_CELL_BY_NAME(node_id, name, flags)* if there is a flags cell, but expands to zero if there is none.

See also

DT_PWMS_CELL_BY_NAME()

Parameters

- `node_id` – node identifier for a node with a pwms property
- `name` – lowercase-and-underscores name of a pwms element as defined by the node's pwm-names property

Returns
the flags cell value in the specifier at the named element, or zero if there is none

`DT_PWMS_FLAGS`(node_id)

Equivalent to *DT_PWMS_FLAGS_BY_IDX(node_id, 0)*

See also

DT_PWMS_FLAGS_BY_IDX()

Parameters

- `node_id` – node identifier for a node with a pwms property

Returns
the flags cell value at index 0, or zero if there is none

`DT_INST_PWMS_CTLR_BY_IDX`(inst, idx)

Get the node identifier for the PWM controller from a DT_DRV_COMPAT instance's pwms property at an index.

See also

DT_PWMS_CTLR_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `idx` – logical index into pwms property

Returns
the node identifier for the PWM controller referenced at index "idx"

`DT_INST_PWMS_CTLR_BY_NAME`(inst, name)

Get the node identifier for the PWM controller from a DT_DRV_COMPAT instance's pwms property by name.

See also

DT_PWMS_CTLR_BY_NAME()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `name` – lowercase-and-underscores name of a pwms element as defined by the node's pwm-names property

Returns
the node identifier for the PWM controller in the named element

`DT_INST_PWMS_CTLR`(inst)

Equivalent to *DT_INST_PWMS_CTLR_BY_IDX(inst, 0)*

See also

DT_PWMS_CTLR_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number

Returns
the node identifier for the PWM controller at index 0 in the instance's "pwms" property

`DT_INST_PWMS_CELL_BY_IDX`(inst, idx, cell)

Get a DT_DRV_COMPAT instance's PWM specifier's cell value at an index.

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `idx` – logical index into pwms property
- `cell` – lowercase-and-underscores cell name

Returns
the cell value at index "idx"

`DT_INST_PWMS_CELL_BY_NAME`(inst, name, cell)

Get a DT_DRV_COMPAT instance's PWM specifier's cell value by name.

See also

DT_PWMS_CELL_BY_NAME()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `name` – lowercase-and-underscores name of a pwms element as defined by the node's pwm-names property
- `cell` – lowercase-and-underscores cell name

Returns
the cell value in the specifier at the named element

`DT_INST_PWMS_CELL`(inst, cell)

Equivalent to *DT_INST_PWMS_CELL_BY_IDX(inst, 0, cell)*

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `cell` – lowercase-and-underscores cell name

Returns
the cell value at index 0

`DT_INST_PWMS_CHANNEL_BY_IDX`(inst, idx)

Equivalent to *DT_INST_PWMS_CELL_BY_IDX(inst, idx, channel)*

See also

DT_INST_PWMS_CELL_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `idx` – logical index into pwms property

Returns
the channel cell value at index “idx”

`DT_INST_PWMS_CHANNEL_BY_NAME`(inst, name)
Equivalent to *DT_INST_PWMS_CELL_BY_NAME(inst, name, channel)*

See also

DT_INST_PWMS_CELL_BY_NAME()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `name` – lowercase-and-underscores name of a pwms element as defined by the node’s pwm-names property

Returns
the channel cell value in the specifier at the named element

`DT_INST_PWMS_CHANNEL`(inst)
Equivalent to *DT_INST_PWMS_CHANNEL_BY_IDX(inst, 0)*

See also

DT_INST_PWMS_CHANNEL_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number

Returns
the channel cell value at index 0

`DT_INST_PWMS_PERIOD_BY_IDX`(inst, idx)
Equivalent to *DT_INST_PWMS_CELL_BY_IDX(inst, idx, period)*

See also

DT_INST_PWMS_CELL_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `idx` – logical index into pwms property

Returns
the period cell value at index “idx”

DT_INST_PWMS_PERIOD_BY_NAME(inst, name)

Equivalent to *DT_INST_PWMS_CELL_BY_NAME(inst, name, period)*

See also

DT_INST_PWMS_CELL_BY_NAME()

Parameters

- **inst** – DT_DRV_COMPAT instance number
- **name** – lowercase-and-underscores name of a pwms element as defined by the node's pwm-names property

Returns

the period cell value in the specifier at the named element

DT_INST_PWMS_PERIOD(inst)

Equivalent to *DT_INST_PWMS_PERIOD_BY_IDX(inst, 0)*

See also

DT_INST_PWMS_PERIOD_BY_IDX()

Parameters

- **inst** – DT_DRV_COMPAT instance number

Returns

the period cell value at index 0

DT_INST_PWMS_FLAGS_BY_IDX(inst, idx)

Equivalent to *DT_INST_PWMS_CELL_BY_IDX(inst, idx, flags)*

See also

DT_INST_PWMS_CELL_BY_IDX()

Parameters

- **inst** – DT_DRV_COMPAT instance number
- **idx** – logical index into pwms property

Returns

the flags cell value at index "idx", or zero if there is none

DT_INST_PWMS_FLAGS_BY_NAME(inst, name)

Equivalent to *DT_INST_PWMS_CELL_BY_NAME(inst, name, flags)*

See also

DT_INST_PWMS_CELL_BY_NAME()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `name` – lowercase-and-underscores name of a pwms element as defined by the node's pwm-names property

Returns
the flags cell value in the specifier at the named element, or zero if there is none

`DT_INST_PWMS_FLAGS`(inst)
Equivalent to *DT_INST_PWMS_FLAGS_BY_IDX(inst, 0)*

See also

DT_INST_PWMS_FLAGS_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number

Returns
the flags cell value at index 0, or zero if there is none

Reset Controller These conveniences may be used for nodes which describe reset controllers and properties related to them.

group `devicetree-reset-controller`

Defines

`DT_RESET_CTLR_BY_IDX`(node_id, idx)
Get the node identifier for the controller phandle from a "resets" phandle-array property at an index.

Example devicetree fragment:

```
reset1: reset-controller@... { ... };

reset2: reset-controller@... { ... };

n: node {
        resets = <&reset1 10>, <&reset2 20>;
};
```

Example usage:

```
DT_RESET_CTLR_BY_IDX(DT_NODELABEL(n), 0)) // DT_NODELABEL(reset1)
DT_RESET_CTLR_BY_IDX(DT_NODELABEL(n), 1)) // DT_NODELABEL(reset2)
```

See also

DT_PHANDLE_BY_IDX()

Parameters

- `node_id` – node identifier
- `idx` – logical index into “resets”

Returns
the node identifier for the reset controller referenced at index “idx”

`DT_RESET_CTLR`(node_id)

Equivalent to *DT_RESET_CTLR_BY_IDX(node_id, 0)*

See also

DT_RESET_CTLR_BY_IDX()

Parameters

- `node_id` – node identifier

Returns
a node identifier for the reset controller at index 0 in “resets”

`DT_RESET_CTLR_BY_NAME`(node_id, name)

Get the node identifier for the controller phandle from a resets phandle-array property by name.

Example devicetree fragment:

```
reset1: reset-controller@... { ... };

reset2: reset-controller@... { ... };

n: node {
        resets = <&reset1 10>, <&reset2 20>;
        reset-names = "alpha", "beta";
};
```

Example usage:

```
DT_RESET_CTLR_BY_NAME(DT_NODELABEL(n), alpha) // DT_NODELABEL(reset1)
DT_RESET_CTLR_BY_NAME(DT_NODELABEL(n), beta) // DT_NODELABEL(reset2)
```

See also

DT_PHANDLE_BY_NAME()

Parameters

- `node_id` – node identifier
- `name` – lowercase-and-underscores name of a resets element as defined by the node’s reset-names property

Returns
the node identifier for the reset controller referenced by name

`DT_RESET_CELL_BY_IDX`(node_id, idx, cell)

Get a reset specifier's cell value at an index.

Example devicetree fragment:

```
reset: reset-controller@... {
        compatible = "vnd,reset";
        #reset-cells = <1>;
};

n: node {
        resets = <&reset 10>;
};
```

Bindings fragment for the vnd,reset compatible:

```
reset-cells:
  - id
```

Example usage:

```
DT_RESET_CELL_BY_IDX(DT_NODELABEL(n), 0, id) // 10
```

See also

DT_PHA_BY_IDX()

Parameters

- **node_id** – node identifier for a node with a resets property
- **idx** – logical index into resets property
- **cell** – lowercase-and-underscores cell name

Returns
the cell value at index "idx"

`DT_RESET_CELL_BY_NAME`(node_id, name, cell)

Get a reset specifier's cell value by name.

Example devicetree fragment:

```
reset: reset-controller@... {
        compatible = "vnd,reset";
        #reset-cells = <1>;
};

n: node {
        resets = <&reset 10>;
        reset-names = "alpha";
};
```

Bindings fragment for the vnd,reset compatible:

```
reset-cells:
  - id
```

Example usage:

```
DT_RESET_CELL_BY_NAME(DT_NODELABEL(n), alpha, id) // 10
```

See also

DT_PHA_BY_NAME()

Parameters

- `node_id` – node identifier for a node with a resets property
- `name` – lowercase-and-underscores name of a resets element as defined by the node's reset-names property
- `cell` – lowercase-and-underscores cell name

Returns
the cell value in the specifier at the named element

`DT_RESET_CELL`(node_id, cell)

Equivalent to *DT_RESET_CELL_BY_IDX(node_id, 0, cell)*

See also

DT_RESET_CELL_BY_IDX()

Parameters

- `node_id` – node identifier for a node with a resets property
- `cell` – lowercase-and-underscores cell name

Returns
the cell value at index 0

`DT_INST_RESET_CTLR_BY_IDX`(inst, idx)

Get the node identifier for the controller phandle from a "resets" phandle-array property at an index.

See also

DT_RESET_CTLR_BY_IDX()

Parameters

- `inst` – instance number
- `idx` – logical index into "resets"

Returns
the node identifier for the reset controller referenced at index "idx"

`DT_INST_RESET_CTLR`(inst)

Equivalent to *DT_INST_RESET_CTLR_BY_IDX(inst, 0)*

See also

DT_RESET_CTLR()

Parameters

- `inst` – instance number

Returns
a node identifier for the reset controller at index 0 in “resets”

`DT_INST_RESET_CTLR_BY_NAME`(inst, name)

Get the node identifier for the controller phandle from a resets phandle-array property by name.

See also

DT_RESET_CTLR_BY_NAME()

Parameters

- `inst` – instance number
- `name` – lowercase-and-underscores name of a resets element as defined by the node’s reset-names property

Returns
the node identifier for the reset controller referenced by the named element

`DT_INST_RESET_CELL_BY_IDX`(inst, idx, cell)

Get a DT_DRV_COMPAT instance’s reset specifier’s cell value at an index.

See also

DT_RESET_CELL_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `idx` – logical index into resets property
- `cell` – lowercase-and-underscores cell name

Returns
the cell value at index “idx”

`DT_INST_RESET_CELL_BY_NAME`(inst, name, cell)

Get a DT_DRV_COMPAT instance’s reset specifier’s cell value by name.

See also

DT_RESET_CELL_BY_NAME()

Parameters

- **inst** – DT_DRV_COMPAT instance number
- **name** – lowercase-and-underscores name of a resets element as defined by the node's reset-names property
- **cell** – lowercase-and-underscores cell name

Returns
the cell value in the specifier at the named element

DT_INST_RESET_CELL(inst, cell)

Equivalent to *DT_INST_RESET_CELL_BY_IDX(inst, 0, cell)*

Parameters

- **inst** – DT_DRV_COMPAT instance number
- **cell** – lowercase-and-underscores cell name

Returns
the value of the cell inside the specifier at index 0

DT_RESET_ID_BY_IDX(node_id, idx)

Get a Reset Controller specifier's id cell at an index.

This macro only works for Reset Controller specifiers with cells named "id". Refer to the node's binding to check if necessary.

Example devicetree fragment:

```
reset: reset-controller@... {
        compatible = "vnd,reset";
        #reset-cells = <1>;
};

n: node {
        resets = <&reset 10>;
};
```

Bindings fragment for the vnd,reset compatible:

```
reset-cells:
  - id
```

Example usage:

```
DT_RESET_ID_BY_IDX(DT_NODELABEL(n), 0) // 10
```

See also

DT_PHA_BY_IDX()

Parameters

- **node_id** – node identifier
- **idx** – logical index into "resets"

Returns
the id cell value at index “idx”

`DT_RESET_ID`(node_id)
Equivalent to *DT_RESET_ID_BY_IDX(node_id, 0)*

See also

DT_RESET_ID_BY_IDX()

Parameters

- `node_id` – node identifier

Returns
the id cell value at index 0

`DT_INST_RESET_ID_BY_IDX`(inst, idx)
Get a DT_DRV_COMPAT instance’s Reset Controller specifier’s id cell value at an index.

See also

DT_RESET_ID_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `idx` – logical index into “resets”

Returns
the id cell value at index “idx”

`DT_INST_RESET_ID`(inst)
Equivalent to *DT_INST_RESET_ID_BY_IDX(inst, 0)*

See also

DT_INST_RESET_ID_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number

Returns
the id cell value at index 0

SPI These conveniences may be used for nodes which describe either SPI controllers or devices, depending on the case.

group `devicetree-spi`

Defines

DT_SPI_HAS_CS_GPIOS(spi)

Does a SPI controller node have chip select GPIOs configured?

SPI bus controllers use the “cs-gpios” property for configuring chip select GPIOs. Its value is a phandle-array which specifies the chip select lines.

Example devicetree fragment:

```
spi1: spi@... {
        compatible = "vnd,spi";
        cs-gpios = <&gpio1 10 GPIO_ACTIVE_LOW>,
                   <&gpio2 20 GPIO_ACTIVE_LOW>;
};

spi2: spi@... {
        compatible = "vnd,spi";
};
```

Example usage:

```
DT_SPI_HAS_CS_GPIOS(DT_NODELABEL(spi1)) // 1
DT_SPI_HAS_CS_GPIOS(DT_NODELABEL(spi2)) // 0
```

Parameters

- **spi** – a SPI bus controller node identifier

Returns

1 if “spi” has a cs-gpios property, 0 otherwise

DT_SPI_NUM_CS_GPIOS(spi)

Number of chip select GPIOs in a SPI controller’s cs-gpios property.

Example devicetree fragment:

```
spi1: spi@... {
        compatible = "vnd,spi";
        cs-gpios = <&gpio1 10 GPIO_ACTIVE_LOW>,
                   <&gpio2 20 GPIO_ACTIVE_LOW>;
};

spi2: spi@... {
        compatible = "vnd,spi";
};
```

Example usage:

```
DT_SPI_NUM_CS_GPIOS(DT_NODELABEL(spi1)) // 2
DT_SPI_NUM_CS_GPIOS(DT_NODELABEL(spi2)) // 0
```

Parameters

- **spi** – a SPI bus controller node identifier

Returns

Logical length of spi’s cs-gpios property, or 0 if “spi” doesn’t have a cs-gpios property

DT_SPI_DEV_HAS_CS_GPIOS(spi_dev)

Does a SPI device have a chip select line configured? Example devicetree fragment:

```
spi1: spi@... {
        compatible = "vnd,spi";
        cs-gpios = <&gpio1 10 GPIO_ACTIVE_LOW>,
                   <&gpio2 20 GPIO_ACTIVE_LOW>;

        a: spi-dev-a@0 {
                reg = <0>;
        };

        b: spi-dev-b@1 {
                reg = <1>;
        };
};

spi2: spi@... {
        compatible = "vnd,spi";
        c: spi-dev-c@0 {
                reg = <0>;
        };
};
```

Example usage:

```
DT_SPI_DEV_HAS_CS_GPIOS(DT_NODELABEL(a)) // 1
DT_SPI_DEV_HAS_CS_GPIOS(DT_NODELABEL(b)) // 1
DT_SPI_DEV_HAS_CS_GPIOS(DT_NODELABEL(c)) // 0
```

Parameters

- **spi_dev** – a SPI device node identifier

Returns

1 if spi_dev's bus node *DT_BUS(spi_dev)* has a chip select pin at index *DT_REG_ADDR(spi_dev)*, 0 otherwise

DT_SPI_DEV_CS_GPIOS_CTLR(spi_dev)

Get a SPI device's chip select GPIO controller's node identifier.

Example devicetree fragment:

```
gpio1: gpio@... { ... };

gpio2: gpio@... { ... };

spi@... {
        compatible = "vnd,spi";
        cs-gpios = <&gpio1 10 GPIO_ACTIVE_LOW>,
                   <&gpio2 20 GPIO_ACTIVE_LOW>;

        a: spi-dev-a@0 {
                reg = <0>;
        };

        b: spi-dev-b@1 {
                reg = <1>;
        };
};
```

Example usage:

```
DT_SPI_DEV_CS_GPIOS_CTLR(DT_NODELABEL(a)) // DT_NODELABEL(gpio1)
DT_SPI_DEV_CS_GPIOS_CTLR(DT_NODELABEL(b)) // DT_NODELABEL(gpio2)
```

Parameters

- **spi_dev** – a SPI device node identifier

Returns
node identifier for spi_dev's chip select GPIO controller

DT_SPI_DEV_CS_GPIOS_PIN(spi_dev)

Get a SPI device's chip select GPIO pin number.

It's an error if the GPIO specifier for spi_dev's entry in its bus node's cs-gpios property has no pin cell.

Example devicetree fragment:

```
spi1: spi@... {
        compatible = "vnd,spi";
        cs-gpios = <&gpio1 10 GPIO_ACTIVE_LOW>,
                   <&gpio2 20 GPIO_ACTIVE_LOW>;

        a: spi-dev-a@0 {
                reg = <0>;
        };

        b: spi-dev-b@1 {
                reg = <1>;
        };
};
```

Example usage:

```
DT_SPI_DEV_CS_GPIOS_PIN(DT_NODELABEL(a)) // 10
DT_SPI_DEV_CS_GPIOS_PIN(DT_NODELABEL(b)) // 20
```

Parameters

- **spi_dev** – a SPI device node identifier

Returns
pin number of spi_dev's chip select GPIO

DT_SPI_DEV_CS_GPIOS_FLAGS(spi_dev)

Get a SPI device's chip select GPIO flags.

Example devicetree fragment:

```
spi1: spi@... {
        compatible = "vnd,spi";
        cs-gpios = <&gpio1 10 GPIO_ACTIVE_LOW>;

        a: spi-dev-a@0 {
                reg = <0>;
        };
};
```

Example usage:

```
DT_SPI_DEV_CS_GPIOS_FLAGS(DT_NODELABEL(a)) // GPIO_ACTIVE_LOW
```

If the GPIO specifier for spi_dev's entry in its bus node's cs-gpios property has no flags cell, this expands to zero.

Parameters

- **spi_dev** – a SPI device node identifier

Returns
flags value of spi_dev's chip select GPIO specifier, or zero if there is none

`DT_INST_SPI_DEV_HAS_CS_GPIOS`(inst)

Equivalent to *DT_SPI_DEV_HAS_CS_GPIOS(DT_DRV_INST(inst))*.

See also

DT_SPI_DEV_HAS_CS_GPIOS()

Parameters

- **inst** – DT_DRV_COMPAT instance number

Returns
1 if the instance's bus has a CS pin at index *DT_INST_REG_ADDR(inst)*, 0 otherwise

`DT_INST_SPI_DEV_CS_GPIOS_CTLR`(inst)

Get GPIO controller node identifier for a SPI device instance This is equivalent to *DT_SPI_DEV_CS_GPIOS_CTLR(DT_DRV_INST(inst))*.

See also

DT_SPI_DEV_CS_GPIOS_CTLR()

Parameters

- **inst** – DT_DRV_COMPAT instance number

Returns
node identifier for instance's chip select GPIO controller

`DT_INST_SPI_DEV_CS_GPIOS_PIN`(inst)

Equivalent to *DT_SPI_DEV_CS_GPIOS_PIN(DT_DRV_INST(inst))*.

See also

DT_SPI_DEV_CS_GPIOS_PIN()

Parameters

- **inst** – DT_DRV_COMPAT instance number

Returns
pin number of the instance's chip select GPIO

`DT_INST_SPI_DEV_CS_GPIOS_FLAGS`(inst)
DT_SPI_DEV_CS_GPIOS_FLAGS(DT_DRV_INST(inst)).

> **See also**
> *DT_SPI_DEV_CS_GPIOS_FLAGS()*

Parameters
- `inst` – DT_DRV_COMPAT instance number

Returns
flags value of the instance's chip select GPIO specifier, or zero if there is none

Chosen nodes The special `/chosen` node contains properties whose values describe system-wide settings. The *DT_CHOSEN()* macro can be used to get a node identifier for a chosen node.

group `devicetree-generic-chosen`

Defines

`DT_CHOSEN`(prop)
Get a node identifier for a `/chosen` node property.

This is only valid to call if *DT_HAS_CHOSEN(prop)* is 1.

Parameters
- `prop` – lowercase-and-underscores property name for the /chosen node

Returns
a node identifier for the chosen node property

`DT_HAS_CHOSEN`(prop)
Test if the devicetree has a `/chosen` node.

Parameters
- `prop` – lowercase-and-underscores devicetree property

Returns
1 if the chosen property exists and refers to a node, 0 otherwise

Zephyr-specific chosen nodes The following table documents some commonly used Zephyr-specific chosen nodes.

Sometimes, a chosen node's label property will be used to set the default value of a Kconfig option which in turn configures a hardware-specific device. This is usually for backwards compatibility in cases when the Kconfig option predates devicetree support in Zephyr. In other cases, there is no Kconfig option, and the devicetree node is used directly in the source code to select a device.

Table 1: Zephyr-specific chosen properties

Property	Purpose
zephyr,bt-c2h-uart	Selects the UART used for host communication in the bluetooth-hci-uart-sample
zephyr,bt-mon-uart	Sets UART device used for the Bluetooth monitor logging
zephyr,bt-hci	Selects the HCI device used by the Bluetooth host stack
zephyr,canbus	Sets the default CAN controller
zephyr,ccm	Core-Coupled Memory node on some STM32 SoCs
zephyr,code-partition	Flash partition that the Zephyr image's text section should be linked into
zephyr,console	Sets UART device used by console driver
zephyr,display	Sets the default display controller
zephyr,keyboard-scan	Sets the default keyboard scan controller
zephyr,dtcm	Data Tightly Coupled Memory node on some Arm SoCs
zephyr,entropy	A device which can be used as a system-wide entropy source
zephyr,flash	A node whose `reg` is sometimes used to set the defaults for `CONFIG_FLASH_BASE_ADDRESS` and `CONFIG_FLASH_SIZE`
zephyr,flash-controller	The node corresponding to the flash controller device for the `zephyr,flash` node
zephyr,gdbstub-uart	Sets UART device used by the *GDB stub* subsystem
zephyr,ieee802154	Used by the networking subsystem to set the IEEE 802.15.4 device
zephyr,ipc	Used by the OpenAMP subsystem to specify the inter-process communication (IPC) device
zephyr,ipc_shm	A node whose `reg` is used by the OpenAMP subsystem to determine the base address and size of the shared memory (SHM) usable for interprocess-communication (IPC)
zephyr,itcm	Instruction Tightly Coupled Memory node on some Arm SoCs
zephyr,log-uart	Sets the UART device(s) used by the logging subsystem's UART backend. If defined, the UART log backend would output to the devices listed in this node.
zephyr,ocm	On-chip memory node on Xilinx Zynq-7000 and ZynqMP SoCs
zephyr,osdp-uart	Sets UART device used by OSDP subsystem
zephyr,ot-uart	Used by the OpenThread to specify UART device for Spinel protocol
zephyr,pcie-controller	The node corresponding to the PCIe Controller
zephyr,ppp-uart	Sets UART device used by PPP
zephyr,settings-partition	Fixed partition node. If defined this selects the partition used by the NVS and FCB settings backends.
zephyr,shell-uart	Sets UART device used by serial shell backend
zephyr,sram	A node whose `reg` sets the base address and size of SRAM memory available to the Zephyr image, used during linking
zephyr,tracing-uart	Sets UART device used by tracing subsystem
zephyr,uart-mcumgr	UART used for *Device Management*
zephyr,uart-pipe	Sets UART device used by serial pipe driver
zephyr,usb-device	USB device node. If defined and has a `vbus-gpios` property, these will be used by the USB subsystem to enable/disable VBUS

Bindings index

This page documents the available devicetree bindings. See *Devicetree bindings* for an introduction to the Zephyr bindings file format.

Vendor index This section contains an index of hardware vendors. Click on a vendor's name to go to the list of bindings for that vendor.

- *Generic or vendor-independent*
- *Advanced Micro Devices (AMD), Inc. (amd)*
- *Altera Corp. (altr)*
- *Ambiq Micro, Inc. (ambiq)*
- *AMS AG (ams)*
- *Analog Devices, Inc. (adi)*
- *Andes Technology Corporation (andestech)*
- *Angst+Pfister (ap)*
- *Apa Electronic Co., Ltd (apa)*
- *Aptina Imaging (aptina)*
- *Arduino (arduino)*
- *ARM Ltd. (arm)*
- *Asahi Kasei Corp. (asahi-kasei)*
- *ASMedia Technology Inc. (asmedia)*
- *ASPEED Technology Inc. (aspeed)*
- *Atmel Corporation (atmel)*
- *Avago Technologies (avago)*
- *Bosch Sensortec GmbH (bosch)*
- *Broadcom Corporation (brcm)*
- *Cadence Design Systems Inc. (cdns)*
- *Chipsemi Corp. (chipsemi)*
- *Cirque Corporation (cirque)*
- *Cirrus Logic, Inc. (cirrus)*
- *Cypress Semiconductor Corporation (cypress)*
- *DFRobot (dfrobot)*
- *Diglent, Inc. (digilent)*
- *Diodes Incorporated (diodes)*
- *Efinix Inc (efinix)*
- *ENE Technology, Inc. (ene)*
- *EPCOS AG (epcos)*
- *Espressif Systems (espressif)*
- *Fairchild Semiconductor (fcs)*
- *Feature Integration Technology Inc. (fintek)*
- *Festo SE & Co. KG (festo)*
- *FocalTech Systems Co.,Ltd (focaltech)*
- *Freescale Semiconductor (fsl)*
- *Fujitsu Ltd. (fujitsu)*
- *Futaba Corporation (futaba)*
- *Future Technology Devices International Ltd. (ftdi)*

- *Gaisler (gaisler)*
- *Galaxycore, Inc. (galaxycore)*
- *Gas Sensing Solutions Ltd. (gss)*
- *GigaDevice Semiconductor (gd)*
- *GreeLed Electronic Ltd. (greeled)*
- *Guangzhou Aosong Electronic Co., Ltd. (aosong)*
- *Hamamatsu Photonics K.K. (hamamatsu)*
- *Hangzhou Grow Technology Co., Ltd. (hzgrow)*
- *Himax Technologies, Inc. (himax)*
- *Hitachi Ltd. (hit)*
- *Holtek Semiconductor, Inc. (holtek)*
- *Honeywell (honeywell)*
- *HOPERF Microelectronics Co. Ltd (hoperf)*
- *Hynitron (hynitron)*
- *ILI Technology Corporation (ILITEK) (ilitek)*
- *Imagination Technologies Ltd. (formerly MIPS Technologies Inc.) (mti)*
- *Infineon Technologies (infineon)*
- *Innovative Sensor Technology IST AG (ist)*
- *Integrated Silicon Solutions Inc. (issi)*
- *Intel Corporation (intel)*
- *Intersil (isil)*
- *InvenSense Inc. (invensense)*
- *Inventek Systems (inventek)*
- *Isentek Inc. (isentek)*
- *ITE Tech. Inc. (ite)*
- *JEDEC Solid State Technology Association (jedec)*
- *Kvaser (kvaser)*
- *Lattice Semiconductor (lattice)*
- *Linaro Limited (linaro)*
- *Linear Technology Corporation (lltc)*
- *LiteOn OptoElectronics (ltr)*
- *LiteX SoC builder (litex)*
- *lowRISC Community Interest Company (lowrisc)*
- *LuatOS Team (luatos)*
- *M5Stack (m5stack)*
- *Maxim Integrated Products (maxim)*
- *Measurement Specialties (meas)*
- *MediaTek Inc. (mediatek)*
- *MEMSIC Inc. (memsic)*

- *Micro Crystal AG (microcrystal)*
- *Micro:bit Educational Foundation (microbit)*
- *Microchip Technology Inc. (microchip)*
- *Micron Technology Inc. (micron)*
- *Motorola, Inc. (motorola)*
- *Murata Manufacturing Co., Ltd. (murata)*
- *National Semiconductor (national)*
- *Nordic Semiconductor (nordic)*
- *Noritake Co., Inc. Electronics Division (noritake)*
- *Nuclei System Technology (nuclei)*
- *Nuvoton Technology Corporation (nuvoton)*
- *NXP Semiconductors (nxp)*
- *OmniVision Technologies Co., Ltd. (ovti)*
- *ON Semiconductor Corp. (onnn)*
- *open-isa.org (openisa)*
- *OpenCores.org (opencores)*
- *OpenThread.io (openthread)*
- *Orise Technology (orisetech)*
- *Panasonic Corporation (panasonic)*
- *PixArt Imaging Inc. (pixart)*
- *Plantower Co., Ltd (plantower)*
- *Princeton Technology Corp. (ptc)*
- *QEMU, a generic and open source machine emulator and virtualizer (qemu)*
- *Qorvo, Inc (formerly Decawave) (decawave)*
- *Quectel Wireless Solutions Co., Ltd. (quectel)*
- *QuickLogic Corp. (quicklogic)*
- *Raspberry Pi Foundation (raspberrypi)*
- *Raydium Semiconductor Corp. (raydium)*
- *Realtek Semiconductor Corp. (realtek)*
- *Renesas Electronics Corporation (renesas)*
- *Reyax Technology Co., Ltd. (reyax)*
- *Richtek Technology Corporation (richtek)*
- *RISC-V Foundation (riscv)*
- *ROCKTECH DISPLAYS LIMITED (rocktech)*
- *ROHM Semiconductor Co., Ltd (rohm)*
- *Sciosense B.V. (sciosense)*
- *Seeed Technology Co., Ltd (seeed)*
- *SEGGER Microcontroller GmbH (segger)*
- *Semtech Corporation (semtech)*

- *Sensirion AG (sensirion)*
- *Sequans Communications (sqn)*
- *Sharp Corporation (sharp)*
- *Shenzhen Frida LCD Co., Ltd. (frida)*
- *Shenzhen Huiding Technology Co., Ltd. (goodix)*
- *Shenzhen Jinghua Displays Electronics Co., Ltd. (jhd)*
- *Shenzhen Xptek Technology Co., Ltd (xptek)*
- *Siemens AG (siemens)*
- *Sierra Wireless (swir)*
- *SiFive, Inc. (sifive)*
- *Silicon Laboratories (silabs)*
- *SIMCom Wireless Solutions Co., LTD (simcom)*
- *Sino Wealth Electronic Ltd (sinowealth)*
- *Sitronix Technology Corporation (sitronix)*
- *Skyworks Solutions, Inc. (skyworks)*
- *Smart Battery System (sbs)*
- *Solomon Systech Limited (solomon)*
- *SparkFun Electronics (sparkfun)*
- *Standard Microsystems Corporation (smsc)*
- *StarFive Technology Co. Ltd. (starfive)*
- *STMicroelectronics (st)*
- *Synopsys, Inc. (snps)*
- *Synopsys, Inc. (formerly ARC International PLC) (arc)*
- *TDK Corporation. (tdk)*
- *Telink Semiconductor (telink)*
- *Telit Cinterion (telit)*
- *Texas Instruments (ti)*
- *u-blox (u-blox)*
- *UltraChip Inc. (ultrachip)*
- *Vishay Intertechnology, Inc (vishay)*
- *Wistron NeWeb Corporation (wnc)*
- *WIZnet Co., Ltd. (wiznet)*
- *Worldsemi Co., Limited (worldsemi)*
- *Würth Elektronik GmbH. (we)*
- *X-Powers (x-powers)*
- *Xen Hypervisor (xen)*
- *Xilinx (xlnx)*
- *Zephyr-specific binding (zephyr)*
- *Zhengzhou Winsen Electronics Technology Co., Ltd. (winsen)*

- *Unknown vendor*

Bindings by vendor This section contains available bindings, grouped by vendor. Within each group, bindings are listed by the "compatible" property they apply to, like this:

Vendor name (vendor prefix)

- <compatible-A>
- <compatible-B> (on <bus-name> bus)
- <compatible-C>
- ...

The text "(on <bus-name> bus)" appears when bindings may behave differently depending on the bus the node appears on. For example, this applies to some sensor device nodes, which may appear as children of either I2C or SPI bus nodes.

Generic or vendor-independent

- dtbinding_adafruit_feather_header
- dtbinding_adc_keys
- dtbinding_ambiq_header
- dtbinding_analog_axis
- dtbinding_arduino_header_r3
- dtbinding_arduino_mkr_header
- dtbinding_arduino_nano_header_r3
- dtbinding_atmel_xplained_header
- dtbinding_atmel_xplained_pro_header
- dtbinding_can_transceiver_gpio
- dtbinding_current_sense_amplifier
- dtbinding_current_sense_shunt
- dtbinding_ethernet_phy
- dtbinding_fixed_clock
- dtbinding_fixed_factor_clock
- dtbinding_fixed_partitions
- dtbinding_generic_fem_two_ctrl_pins
- dtbinding_gnss_nmea_generic
- dtbinding_gpio_i2c
- dtbinding_gpio_i2c_switch
- dtbinding_gpio_kbd_matrix
- dtbinding_gpio_keys
- dtbinding_gpio_leds
- dtbinding_gpio_qdec
- dtbinding_gpio_radio_coex
- dtbinding_grove_header

- dtbinding_input_keymap
- dtbinding_led_strip_matrix
- dtbinding_lm35
- dtbinding_lm75
- dtbinding_lm77
- dtbinding_mikro_bus
- dtbinding_mmio_sram
- dtbinding_mspi_aps6404l
- dtbinding_mspi_atxp032
- dtbinding_neorv32_cpu
- dtbinding_neorv32_gpio
- dtbinding_neorv32_machine_timer
- dtbinding_neorv32_trng
- dtbinding_neorv32_uart
- dtbinding_niosv_machine_timer
- dtbinding_nordic_thingy53_edge_connector
- dtbinding_ns16550
- dtbinding_ntc_thermistor_generic
- dtbinding_nvme_controller
- dtbinding_particle_gen3_header
- dtbinding_pci_host_ecam_generic
- dtbinding_pcie_controller
- dtbinding_power_domain
- dtbinding_power_domain_gpio
- dtbinding_power_domain_gpio_monitor
- dtbinding_pwm_clock
- dtbinding_pwm_leds
- dtbinding_raspberrypi_40pins_header
- dtbinding_regulator_fixed
- dtbinding_regulator_gpio
- dtbinding_sample_controller
- dtbinding_shared_irq
- dtbinding_soc_nv_flash
- dtbinding_st_morpho_header
- dtbinding_swj_connector
- dtbinding_syscon
- dtbinding_vnd_gpio_enable_disable_interrupt
- dtbinding_usb_audio
- dtbinding_usb_audio_feature_volume

- dtbinding_usb_audio_hp
- dtbinding_usb_audio_hs
- dtbinding_usb_audio_mic
- dtbinding_usb_c_connector
- dtbinding_usb_nop_xceiv
- dtbinding_usb_ulpi_phy
- dtbinding_voltage_divider

Advanced Micro Devices (AMD), Inc. (amd)

- dtbinding_amd_sb_tsi

Altera Corp. (altr)

- dtbinding_altr_jtag_uart
- dtbinding_altr_msgdma
- dtbinding_altr_nios2_i2c
- dtbinding_altr_nios2_qspi
- dtbinding_altr_nios2_qspi_nor
- dtbinding_altr_nios2f
- dtbinding_altr_pio_1.0
- dtbinding_altr_uart

Ambiq Micro, Inc. (ambiq)

- dtbinding_ambiq_am1805
- dtbinding_ambiq_apollo3_pinctrl
- dtbinding_ambiq_apollo4_pinctrl
- dtbinding_ambiq_bt_hci_spi
- dtbinding_ambiq_clkctrl
- dtbinding_ambiq_counter
- dtbinding_ambiq_flash_controller
- dtbinding_ambiq_gpio
- dtbinding_ambiq_gpio_bank
- dtbinding_ambiq_i2c
- dtbinding_ambiq_mspi
- dtbinding_ambiq_mspi_controller
- dtbinding_ambiq_mspi_device
- dtbinding_ambiq_pwrctrl
- dtbinding_ambiq_spi
- dtbinding_ambiq_spi_bleif
- dtbinding_ambiq_stimer

- dtbinding_ambiq_uart
- dtbinding_ambiq_watchdog

AMS AG (ams)

- dtbinding_ams_as5600
- dtbinding_ams_as6212
- dtbinding_ams_ccs811
- dtbinding_ams_ens210
- dtbinding_ams_iaqcore
- dtbinding_ams_tcs3400
- dtbinding_ams_tmd2620
- dtbinding_ams_tsl2540
- dtbinding_ams_tsl2561
- dtbinding_ams_tsl2591

Analog Devices, Inc. (adi)

- dtbinding_adi_ad559x_i2c
- dtbinding_adi_ad559x_spi
- dtbinding_adi_ad559x_adc
- dtbinding_adi_ad559x_dac
- dtbinding_adi_ad559x_gpio
- dtbinding_adi_ad5628
- dtbinding_adi_ad5648
- dtbinding_adi_ad5668
- dtbinding_adi_ad5672
- dtbinding_adi_ad5674
- dtbinding_adi_ad5676
- dtbinding_adi_ad5679
- dtbinding_adi_ad5684
- dtbinding_adi_ad5686
- dtbinding_adi_ad5687
- dtbinding_adi_ad5689
- dtbinding_adi_ad5691
- dtbinding_adi_ad5692
- dtbinding_adi_ad5693
- dtbinding_adi_adin1100_phy
- dtbinding_adi_adin1110
- dtbinding_adi_adin2111
- dtbinding_adi_adin2111_mdio

- dtbinding_adi_adin2111_phy
- dtbinding_adi_adltc2990
- dtbinding_adi_adp5360
- dtbinding_adi_adp5360_regulator
- dtbinding_adi_adp5585
- dtbinding_adi_adp5585_gpio
- dtbinding_adi_adt7310
- dtbinding_adi_adt7420
- dtbinding_adi_adxl345_spi
- dtbinding_adi_adxl345_i2c
- dtbinding_adi_adxl362
- dtbinding_adi_adxl367_i2c
- dtbinding_adi_adxl367_spi
- dtbinding_adi_adxl372_spi
- dtbinding_adi_adxl372_i2c
- dtbinding_adi_max32_gcr
- dtbinding_adi_max32_gpio
- dtbinding_adi_max32_i2c
- dtbinding_adi_max32_pinctrl
- dtbinding_adi_max32_spi
- dtbinding_adi_max32_uart
- dtbinding_adi_sdp_120

Andes Technology Corporation (andestech)

- dtbinding_andes_andescore_v5
- dtbinding_andestech_atcdmac300
- dtbinding_andestech_atcgpio100
- dtbinding_andestech_atciic100
- dtbinding_andestech_atcpit100
- dtbinding_andestech.atcspi200
- dtbinding_andestech_atcwdt200
- dtbinding_andestech_l2c
- dtbinding_andestech_machine_timer
- dtbinding_andestech_plic_sw
- dtbinding_andestech_qspi_nor

Angst+Pfister (ap)

- dtbinding_ap_fcx_mldx5

Apa Electronic Co., Ltd (apa)

- dtbinding_apa_apa102

Aptina Imaging (aptina)

- dtbinding_aptina_mt9m114

Arduino (arduino)

- dtbinding_arduino_uno_adc

ARM Ltd. (arm)

- dtbinding_arm_armv6m_mpu
- dtbinding_arm_armv6m_systick
- dtbinding_arm_armv7m_itm
- dtbinding_arm_armv7m_mpu
- dtbinding_arm_armv7m_systick
- dtbinding_arm_armv8_timer
- dtbinding_arm_armv8.1m_mpu
- dtbinding_arm_armv8.1m_systick
- dtbinding_arm_armv8m_itm
- dtbinding_arm_armv8m_mpu
- dtbinding_arm_armv8m_systick
- dtbinding_arm_beetle_syscon
- dtbinding_arm_cmsdk_dtimer
- dtbinding_arm_cmsdk_gpio
- dtbinding_arm_cmsdk_timer
- dtbinding_arm_cmsdk_uart
- dtbinding_arm_cmsdk_watchdog
- dtbinding_arm_cortex_a53
- dtbinding_arm_cortex_a55
- dtbinding_arm_cortex_a72
- dtbinding_arm_cortex_a76
- dtbinding_arm_cortex_m0
- dtbinding_arm_cortex_m0+
- dtbinding_arm_cortex_m1
- dtbinding_arm_cortex_m23
- dtbinding_arm_cortex_m3
- dtbinding_arm_cortex_m33
- dtbinding_arm_cortex_m33f
- dtbinding_arm_cortex_m4

- dtbinding_arm_cortex_m4f
- dtbinding_arm_cortex_m55
- dtbinding_arm_cortex_m55f
- dtbinding_arm_cortex_m7
- dtbinding_arm_cortex_m85
- dtbinding_arm_cortex_m85f
- dtbinding_arm_cortex_r4
- dtbinding_arm_cortex_r4f
- dtbinding_arm_cortex_r5
- dtbinding_arm_cortex_r52
- dtbinding_arm_cortex_r5f
- dtbinding_arm_cortex_r7
- dtbinding_arm_cortex_r82
- dtbinding_arm_cryptocell_310
- dtbinding_arm_cryptocell_312
- dtbinding_arm_dma_pl330
- dtbinding_arm_dtcm
- dtbinding_arm_ethos_u
- dtbinding_arm_gic
- dtbinding_arm_gic_v1
- dtbinding_arm_gic_v2
- dtbinding_arm_gic_v3
- dtbinding_arm_gic_v3_its
- dtbinding_arm_itcm
- dtbinding_arm_mhu
- dtbinding_arm_mps2_fpgaio_gpio
- dtbinding_arm_mps3_fpgaio_gpio
- dtbinding_arm_pl011
- dtbinding_arm_pl022
- dtbinding_arm_psci_0.2
- dtbinding_arm_psci_1.1
- dtbinding_arm_sbsa_uart
- dtbinding_arm_scc
- dtbinding_arm_v6m_nvic
- dtbinding_arm_v7m_nvic
- dtbinding_arm_v8.1m_nvic
- dtbinding_arm_v8m_nvic
- dtbinding_arm_versatile_i2c

Asahi Kasei Corp. (asahi-kasei)

- dtbinding_asahi_kasei_ak8975
- dtbinding_asahi_kasei_akm09918c

ASMedia Technology Inc. (asmedia)

- dtbinding_asmedia_asm2364

ASPEED Technology Inc. (aspeed)

- dtbinding_aspeed_ast10x0_clock
- dtbinding_aspeed_ast10x0_reset

Atmel Corporation (atmel)

- dtbinding_atmel_at24
- dtbinding_atmel_24mac402
- dtbinding_atmel_at25
- dtbinding_atmel_at45
- dtbinding_atmel_ataes132a
- dtbinding_atmel_rf2xx
- dtbinding_atmel_sam_adc
- dtbinding_atmel_sam_afec
- dtbinding_atmel_sam_can
- dtbinding_atmel_sam_dac
- dtbinding_atmel_sam_flash
- dtbinding_atmel_sam_flash_controller
- dtbinding_atmel_sam_gmac
- dtbinding_atmel_sam_gpio
- dtbinding_atmel_sam_hsmci
- dtbinding_atmel_sam_i2c_twi
- dtbinding_atmel_sam_i2c_twihs
- dtbinding_atmel_sam_i2c_twim
- dtbinding_atmel_sam_mdio
- dtbinding_atmel_sam_pinctrl
- dtbinding_atmel_sam_pmc
- dtbinding_atmel_sam_pwm
- dtbinding_atmel_sam_rstc
- dtbinding_atmel_sam_rtc
- dtbinding_atmel_sam_smc
- dtbinding_atmel_sam_spi
- dtbinding_atmel_sam_ssc

- dtbinding_atmel_sam_supc
- dtbinding_atmel_sam_tc
- dtbinding_atmel_sam_tc_qdec
- dtbinding_atmel_sam_trng
- dtbinding_atmel_sam_uart
- dtbinding_atmel_sam_usart
- dtbinding_atmel_sam_usbc
- dtbinding_atmel_sam_usbhs
- dtbinding_atmel_sam_watchdog
- dtbinding_atmel_sam_xdmac
- dtbinding_atmel_sam0_adc
- dtbinding_atmel_sam0_can
- dtbinding_atmel_sam0_dac
- dtbinding_atmel_sam0_dmac
- dtbinding_atmel_sam0_eic
- dtbinding_atmel_sam0_gmac
- dtbinding_atmel_sam0_gpio
- dtbinding_atmel_sam0_i2c
- dtbinding_atmel_sam0_id
- dtbinding_atmel_sam0_nvmctrl
- dtbinding_atmel_sam0_pinctrl
- dtbinding_atmel_sam0_pinmux
- dtbinding_atmel_sam0_rtc
- dtbinding_atmel_sam0_sercom
- dtbinding_atmel_sam0_spi
- dtbinding_atmel_sam0_tc32
- dtbinding_atmel_sam0_tcc_pwm
- dtbinding_atmel_sam0_uart
- dtbinding_atmel_sam0_usb
- dtbinding_atmel_sam0_watchdog
- dtbinding_atmel_sam4l_flashcalw_controller
- dtbinding_atmel_sam4l_gpio
- dtbinding_atmel_sam4l_uid
- dtbinding_atmel_samc2x_gclk
- dtbinding_atmel_samc2x_mclk
- dtbinding_atmel_samd2x_gclk
- dtbinding_atmel_samd2x_pm
- dtbinding_atmel_samd5x_gclk
- dtbinding_atmel_samd5x_mclk

- dtbinding_atmel_saml2x_gclk
- dtbinding_atmel_saml2x_mclk
- dtbinding_atmel_winc1500

Avago Technologies (avago)

- dtbinding_avago_apds9960

Bosch Sensortec GmbH (bosch)

- dtbinding_bosch_bma280
- dtbinding_bosch_bma4xx_i2c
- dtbinding_bosch_bma4xx_spi
- dtbinding_bosch_bmc150_magn
- dtbinding_bosch_bme280_spi
- dtbinding_bosch_bme280_i2c
- dtbinding_bosch_bme680_spi
- dtbinding_bosch_bme680_i2c
- dtbinding_bosch_bmg160
- dtbinding_bosch_bmi08x_accel_i2c
- dtbinding_bosch_bmi08x_accel_spi
- dtbinding_bosch_bmi08x_gyro_i2c
- dtbinding_bosch_bmi08x_gyro_spi
- dtbinding_bosch_bmi160_i2c
- dtbinding_bosch_bmi160_spi
- dtbinding_bosch_bmi270_i2c
- dtbinding_bosch_bmi270_spi
- dtbinding_bosch_bmi270
- dtbinding_bosch_bmi323
- dtbinding_bosch_bmm150_spi
- dtbinding_bosch_bmm150_i2c
- dtbinding_bosch_bmp388_i2c
- dtbinding_bosch_bmp388_spi
- dtbinding_bosch_bmp581

Broadcom Corporation (brcm)

- dtbinding_brcm_bcm2711_aux_uart
- dtbinding_brcm_bcm2711_gpio
- dtbinding_brcm_brcmstb_gpio
- dtbinding_brcm_iproc_gpio
- dtbinding_brcm_iproc_i2c

- dtbinding_brcm_iproc_pax_dma_v1
- dtbinding_brcm_iproc_pax_dma_v2
- dtbinding_brcm_iproc_pcie_ep

Cadence Design Systems Inc. (cdns)

- dtbinding_cdns_i3c
- dtbinding_cdns_nand
- dtbinding_cdns_qspi_nor
- dtbinding_cdns_sdhc
- dtbinding_cdns_tensilica_xtensa_lx3
- dtbinding_cdns_tensilica_xtensa_lx4
- dtbinding_cdns_tensilica_xtensa_lx6
- dtbinding_cdns_tensilica_xtensa_lx7
- dtbinding_cdns_uart
- dtbinding_cdns_xtensa_core_intc

Chipsemi Corp. (chipsemi)

- dtbinding_chipsemi_chsc6x

Cirque Corporation (cirque)

- dtbinding_cirque_pinnacle_spi
- dtbinding_cirque_pinnacle_i2c

Cirrus Logic, Inc. (cirrus)

- dtbinding_cirrus_cp9314
- dtbinding_cirrus_cs47l63

Cypress Semiconductor Corporation (cypress)

- dtbinding_cypress_cy8c95xx_gpio
- dtbinding_cypress_cy8c95xx_gpio_port
- dtbinding_cypress_psoc6_flash_controller
- dtbinding_cypress_psoc6_gpio
- dtbinding_cypress_psoc6_hsiom
- dtbinding_cypress_psoc6_intmux
- dtbinding_cypress_psoc6_intmux_ch
- dtbinding_cypress_psoc6_spi
- dtbinding_cypress_psoc6_uart
- dtbinding_cypress_psoc6_uid

DFRobot (dfrobot)

- dtbinding_dfrobot_a01nyub

Diglent, Inc. (digilent)

- dtbinding_digilent_pmod

Diodes Incorporated (diodes)

- dtbinding_diodes_pi3usb9201

Efinix Inc (efinix)

- dtbinding_efinix_sapphire_gpio
- dtbinding_efinix_sapphire_timer0
- dtbinding_efinix_sapphire_uart0
- dtbinding_efinix_vexriscv_sapphire

ENE Technology, Inc. (ene)

- dtbinding_ene_kb1200_adc
- dtbinding_ene_kb1200_gcfg
- dtbinding_ene_kb1200_gpio
- dtbinding_ene_kb1200_i2c
- dtbinding_ene_kb1200_pinctrl
- dtbinding_ene_kb1200_pmu
- dtbinding_ene_kb1200_pwm
- dtbinding_ene_kb1200_tach
- dtbinding_ene_kb1200_uart
- dtbinding_ene_kb1200_watchdog

EPCOS AG (epcos)

- dtbinding_epcos_b57861s0103a039

Espressif Systems (espressif)

- dtbinding_espressif_esp_at
- dtbinding_espressif_esp32_adc
- dtbinding_espressif_esp32_bt_hci
- dtbinding_espressif_esp32_dac
- dtbinding_espressif_esp32_eth
- dtbinding_espressif_esp32_flash_controller
- dtbinding_espressif_esp32_gdma
- dtbinding_espressif_esp32_gpio

- dtbinding_espressif_esp32_i2c
- dtbinding_espressif_esp32_intc
- dtbinding_espressif_esp32_ipm
- dtbinding_espressif_esp32_ledc
- dtbinding_espressif_esp32_mcpwm
- dtbinding_espressif_esp32_mdio
- dtbinding_espressif_esp32_pcnt
- dtbinding_espressif_esp32_pinctrl
- dtbinding_espressif_esp32_rtc
- dtbinding_espressif_esp32_rtc_timer
- dtbinding_espressif_esp32_sdhc
- dtbinding_espressif_esp32_sdhc_slot
- dtbinding_espressif_esp32_spi
- dtbinding_espressif_esp32_systimer
- dtbinding_espressif_esp32_temp
- dtbinding_espressif_esp32_timer
- dtbinding_espressif_esp32_touch_sensor
- dtbinding_espressif_esp32_trng
- dtbinding_espressif_esp32_twai
- dtbinding_espressif_esp32_uart
- dtbinding_espressif_esp32_usb_serial
- dtbinding_espressif_esp32_watchdog
- dtbinding_espressif_esp32_wifi
- dtbinding_espressif_esp32_xt_wdt
- dtbinding_espressif_riscv
- dtbinding_espressif_xtensa_lx6
- dtbinding_espressif_xtensa_lx7

Fairchild Semiconductor (fcs)

- dtbinding_fcs_fxl6408

Feature Integration Technology Inc. (fintek)

- dtbinding_fintek_f75303

Festo SE & Co. KG (festo)

- dtbinding_festo_veaa_x_3

FocalTech Systems Co.,Ltd (focaltech)

- dtbinding_focaltech_ft5336

Freescale Semiconductor (fsl)

- dtbinding_fsl_imx21_i2c
- dtbinding_fsl_imx27_pwm

Fujitsu Ltd. (fujitsu)

- dtbinding_fujitsu_mb85rcxx

Futaba Corporation (futaba)

- dtbinding_futaba_sbus

Future Technology Devices International Ltd. (ftdi)

- dtbinding_ftdi_ft800

Gaisler (gaisler)

- dtbinding_gaisler_apbuart
- dtbinding_gaisler_gptimer
- dtbinding_gaisler_irqmp
- dtbinding_gaisler_leon3
- dtbinding_gaisler_spimctrl

Galaxycore, Inc. (galaxycore)

- dtbinding_galaxycore_gc9x01x

Gas Sensing Solutions Ltd. (gss)

- dtbinding_gss_explorir_m

GigaDevice Semiconductor (gd)

- dtbinding_gd_gd32_adc
- dtbinding_gd_gd32_afio
- dtbinding_gd_gd32_cctl
- dtbinding_gd_gd32_dac
- dtbinding_gd_gd32_dma
- dtbinding_gd_gd32_dma_v1
- dtbinding_gd_gd32_exti
- dtbinding_gd_gd32_flash_controller
- dtbinding_gd_gd32_fwdgt
- dtbinding_gd_gd32_gpio
- dtbinding_gd_gd32_i2c
- dtbinding_gd_gd32_nv_flash_v1

- dtbinding_gd_gd32_nv_flash_v2
- dtbinding_gd_gd32_nv_flash_v3
- dtbinding_gd_gd32_pinctrl_af
- dtbinding_gd_gd32_pinctrl_afio
- dtbinding_gd_gd32_pwm
- dtbinding_gd_gd32_rctl
- dtbinding_gd_gd32_rcu
- dtbinding_gd_gd32_spi
- dtbinding_gd_gd32_syscfg
- dtbinding_gd_gd32_timer
- dtbinding_gd_gd32_usart
- dtbinding_gd_gd32_wwdgt

GreeLed Electronic Ltd. (greeled)

- dtbinding_greeled_lpd8803
- dtbinding_greeled_lpd8806

Guangzhou Aosong Electronic Co., Ltd. (aosong)

- dtbinding_aosong_ags10
- dtbinding_aosong_aht20
- dtbinding_aosong_am2301b
- dtbinding_aosong_dht
- dtbinding_aosong_dht20

Hamamatsu Photonics K.K. (hamamatsu)

- dtbinding_hamamatsu_s11059

Hangzhou Grow Technology Co., Ltd. (hzgrow)

- dtbinding_hzgrow_r502a

Himax Technologies, Inc. (himax)

- dtbinding_himax_hx8394

Hitachi Ltd. (hit)

- dtbinding_hit_hd44780

Holtek Semiconductor, Inc. (holtek)

- dtbinding_holtek_ht16k33

Honeywell (honeywell)

- dtbinding_honeywell_hmc5883l
- dtbinding_honeywell_mpr
- dtbinding_honeywell_sm351lt

HOPERF Microelectronics Co. Ltd (hoperf)

- dtbinding_hoperf_hp206c
- dtbinding_hoperf_th02

Hynitron (hynitron)

- dtbinding_hynitron_cst816s

ILI Technology Corporation (ILITEK) (ilitek)

- dtbinding_ilitek_ili9340
- dtbinding_ilitek_ili9341
- dtbinding_ilitek_ili9342c
- dtbinding_ilitek_ili9488

Imagination Technologies Ltd. (formerly MIPS Technologies Inc.) (mti)

- dtbinding_mti_cpu_intc

Infineon Technologies (infineon)

- dtbinding_infineon_airoc_wifi
- dtbinding_infineon_cat1_adc
- dtbinding_infineon_cat1_bless_hci
- dtbinding_infineon_cat1_counter
- dtbinding_infineon_cat1_flash_controller
- dtbinding_infineon_cat1_gpio
- dtbinding_infineon_cat1_i2c
- dtbinding_infineon_cat1_pinctrl
- dtbinding_infineon_cat1_scb
- dtbinding_infineon_cat1_sdhc_sdio
- dtbinding_infineon_cat1_spi
- dtbinding_infineon_cat1_uart
- dtbinding_infineon_cat1_watchdog
- dtbinding_infineon_cyw208xx_hci
- dtbinding_infineon_cyw43xxx_bt_hci
- dtbinding_infineon_dps310
- dtbinding_infineon_tle9104

- dtbinding_infineon_tle9104_diagnostics
- dtbinding_infineon_tle9104_gpio
- dtbinding_infineon_xmc4xxx_adc
- dtbinding_infineon_xmc4xxx_can
- dtbinding_infineon_xmc4xxx_can_node
- dtbinding_infineon_xmc4xxx_ccu4_pwm
- dtbinding_infineon_xmc4xxx_ccu8_pwm
- dtbinding_infineon_xmc4xxx_dma
- dtbinding_infineon_xmc4xxx_ethernet
- dtbinding_infineon_xmc4xxx_flash_controller
- dtbinding_infineon_xmc4xxx_gpio
- dtbinding_infineon_xmc4xxx_i2c
- dtbinding_infineon_xmc4xxx_intc
- dtbinding_infineon_xmc4xxx_mdio
- dtbinding_infineon_xmc4xxx_nv_flash
- dtbinding_infineon_xmc4xxx_pinctrl
- dtbinding_infineon_xmc4xxx_spi
- dtbinding_infineon_xmc4xxx_temp
- dtbinding_infineon_xmc4xxx_uart
- dtbinding_infineon_xmc4xxx_watchdog

Innovative Sensor Technology IST AG (ist)

- dtbinding_ist_tsic_xx6

Integrated Silicon Solutions Inc. (issi)

- dtbinding_issi_is31fl3194
- dtbinding_issi_is31fl3216a
- dtbinding_issi_is31fl3733

Intel Corporation (intel)

- dtbinding_intel_ace_art_counter
- dtbinding_intel_ace_intc
- dtbinding_intel_ace_rtc_counter
- dtbinding_intel_ace_timestamp
- dtbinding_intel_adsp_communication_widget
- dtbinding_intel_adsp_dfpmcch
- dtbinding_intel_adsp_dfpmccu
- dtbinding_intel_adsp_dmic_vss
- dtbinding_intel_adsp_gpdma

- dtbinding_intel_adsp_hda_dmic_cap
- dtbinding_intel_adsp_hda_host_in
- dtbinding_intel_adsp_hda_host_out
- dtbinding_intel_adsp_hda_link_in
- dtbinding_intel_adsp_hda_link_out
- dtbinding_intel_adsp_hda_ssp_cap
- dtbinding_intel_adsp_host_ipc
- dtbinding_intel_adsp_idc
- dtbinding_intel_adsp_imr
- dtbinding_intel_adsp_mailbox
- dtbinding_intel_adsp_mem_window
- dtbinding_intel_adsp_mtl_tlb
- dtbinding_intel_adsp_power_domain
- dtbinding_intel_adsp_sha
- dtbinding_intel_adsp_shim_clkctl
- dtbinding_intel_adsp_timer
- dtbinding_intel_adsp_tlb
- dtbinding_intel_adsp_watchdog
- dtbinding_intel_agilex_clock
- dtbinding_intel_agilex5_clock
- dtbinding_intel_alder_lake
- dtbinding_intel_alh_dai
- dtbinding_intel_apollo_lake
- dtbinding_intel_blinky_pwm
- dtbinding_intel_cavs_i2s
- dtbinding_intel_cavs_intc
- dtbinding_intel_dai_dmic
- dtbinding_intel_e1000
- dtbinding_intel_elkhart_lake
- dtbinding_intel_emmc_host
- dtbinding_intel_gpio
- dtbinding_intel_hda_dai
- dtbinding_intel_hpet
- dtbinding_intel_ibecc
- dtbinding_intel_ioapic
- dtbinding_intel_ish
- dtbinding_intel_lakemont
- dtbinding_intel_loapic
- dtbinding_intel_lpss

- dtbinding_intel_lw_uart
- dtbinding_intel_multiboot_framebuffer
- dtbinding_intel_niosv
- dtbinding_intel_pch_smbus
- dtbinding_intel_penwell_spi
- dtbinding_intel_raptor_lake
- dtbinding_intel_sedi_dma
- dtbinding_intel_sedi_gpio
- dtbinding_intel_sedi_i2c
- dtbinding_intel_sedi_ipm
- dtbinding_intel_sedi_spi
- dtbinding_intel_sedi_uart
- dtbinding_intel_agilex_socfpga_sip_smc
- dtbinding_intel_socfpga_reset
- dtbinding_intel_ssp
- dtbinding_intel_ssp_dai
- dtbinding_intel_ssp_sspbase
- dtbinding_intel_tco_wdt
- dtbinding_intel_timeaware_gpio
- dtbinding_intel_vt_d
- dtbinding_intel_x86

Intersil (isil)

- dtbinding_isil_isl29035

InvenSense Inc. (invensense)

- dtbinding_invensense_icm42605
- dtbinding_invensense_icm42670
- dtbinding_invensense_icm42688
- dtbinding_invensense_icp10125
- dtbinding_invensense_mpu6050
- dtbinding_invensense_mpu9250

Inventek Systems (inventek)

- dtbinding_inventek_eswifi
- dtbinding_inventek_eswifi_uart

Isentek Inc. (isentek)

- dtbinding_istentek_ist8310

ITE Tech. Inc. (ite)

- dtbinding_ite_enhance_i2c
- dtbinding_ite_it82xx2_usb
- dtbinding_ite_it8xxx2_adc
- dtbinding_ite_it8xxx2_bbram
- dtbinding_ite_it8xxx2_espi
- dtbinding_ite_it8xxx2_flash_controller
- dtbinding_ite_it8xxx2_gpio
- dtbinding_ite_it8xxx2_gpio_v2
- dtbinding_ite_it8xxx2_gpiokscan
- dtbinding_ite_it8xxx2_i2c
- dtbinding_ite_it8xxx2_ilm
- dtbinding_ite_it8xxx2_intc
- dtbinding_ite_it8xxx2_intc_v2
- dtbinding_ite_it8xxx2_kbd
- dtbinding_ite_it8xxx2_peci
- dtbinding_ite_it8xxx2_pinctrl
- dtbinding_ite_it8xxx2_pinctrl_func
- dtbinding_ite_it8xxx2_pwm
- dtbinding_ite_it8xxx2_pwmprs
- dtbinding_ite_it8xxx2_sha
- dtbinding_ite_it8xxx2_sha_v2
- dtbinding_ite_it8xxx2_shi
- dtbinding_ite_it8xxx2_sspi
- dtbinding_ite_it8xxx2_tach
- dtbinding_ite_it8xxx2_timer
- dtbinding_ite_it8xxx2_uart
- dtbinding_ite_it8xxx2_usbpd
- dtbinding_ite_it8xxx2_vcmp
- dtbinding_ite_it8xxx2_watchdog
- dtbinding_ite_it8xxx2_wuc
- dtbinding_ite_it8xxx2_wuc_map
- dtbinding_ite_riscv_ite

JEDEC Solid State Technology Association (jedec)

- dtbinding_jedec_spi_nor

Kvaser (kvaser)

- dtbinding_kvaser_pcican

Lattice Semiconductor (lattice)

- dtbinding_lattice_ice40_fpga

Linaro Limited (linaro)

- dtbinding_linaro_96b_lscon_1v8
- dtbinding_linaro_96b_lscon_3v3
- dtbinding_linaro_ivshmem_ipm
- dtbinding_linaro_optee_tz

Linear Technology Corporation (lltc)

- dtbinding_lltc_ltc1660
- dtbinding_lltc_ltc1665
- dtbinding_lltc_ltc2451

LiteOn OptoElectronics (ltr)

- dtbinding_ltrf216a

LiteX SoC builder (litex)

- dtbinding_litex_clk
- dtbinding_litex_clkout
- dtbinding_litex_dna0
- dtbinding_litex_eth0
- dtbinding_litex_gpio
- dtbinding_litex_i2c
- dtbinding_litex_i2s
- dtbinding_litex_prbs
- dtbinding_litex_pwm
- dtbinding_litex_soc_controller
- dtbinding_litex_spi
- dtbinding_litex_timer0
- dtbinding_litex_uart0
- dtbinding_litex_vexriscv_intc0
- dtbinding_litex_vexriscv_standard

lowRISC Community Interest Company (lowrisc)

- dtbinding_lowrisc_ibex
- dtbinding_lowrisc_machine_timer
- dtbinding_lowrisc_opentitan_aontimer
- dtbinding_lowrisc_opentitan_pwrmgr

- dtbinding_lowrisc_opentitan_spi
- dtbinding_lowrisc_opentitan_uart

LuatOS Team (luatos)

- dtbinding_luatos_air530z

M5Stack (m5stack)

- dtbinding_m5stack_atom_header
- dtbinding_m5stack_mbus_header
- dtbinding_m5stack_stamps3_header

Maxim Integrated Products (maxim)

- dtbinding_maxim_ds1307
- dtbinding_maxim_ds18b20
- dtbinding_maxim_ds18s20
- dtbinding_maxim_ds2482_800
- dtbinding_maxim_ds2482_800_channel
- dtbinding_maxim_ds2484
- dtbinding_maxim_ds2485
- dtbinding_maxim_ds3231
- dtbinding_maxim_max11102
- dtbinding_maxim_max11103
- dtbinding_maxim_max11105
- dtbinding_maxim_max11106
- dtbinding_maxim_max11110
- dtbinding_maxim_max11111
- dtbinding_maxim_max11115
- dtbinding_maxim_max11116
- dtbinding_maxim_max11117
- dtbinding_maxim_max11253
- dtbinding_maxim_max11254
- dtbinding_maxim_max17048
- dtbinding_maxim_max17055
- dtbinding_maxim_max17262
- dtbinding_maxim_max20335
- dtbinding_maxim_max20335_charger
- dtbinding_maxim_max20335_regulator
- dtbinding_maxim_max30101
- dtbinding_maxim_max31790

- dtbinding_maxim_max31790_fan_fault
- dtbinding_maxim_max31790_fan_speed
- dtbinding_maxim_max31790_pwm
- dtbinding_maxim_max31855_spi
- dtbinding_maxim_max31865
- dtbinding_maxim_max31875
- dtbinding_maxim_max3421e_spi
- dtbinding_maxim_max44009
- dtbinding_maxim_max6675
- dtbinding_maxim_max7219

Measurement Specialties (meas)

- dtbinding_meas_ms5607_spi
- dtbinding_meas_ms5607_i2c
- dtbinding_meas_ms5837

MediaTek Inc. (mediatek)

- dtbinding_mediatek_adsp_intc
- dtbinding_mediatek_mt8195_cpuclk

MEMSIC Inc. (memsic)

- dtbinding_memsic_mc3419

Micro Crystal AG (microcrystal)

- dtbinding_microcrystal_rv3028

Micro:bit Educational Foundation (microbit)

- dtbinding_microbit_edge_connector

Microchip Technology Inc. (microchip)

- dtbinding_microchip_cap1203
- dtbinding_microchip_coreuart
- dtbinding_microchip_enc28j60
- dtbinding_microchip_enc424j600
- dtbinding_microchip_ksz8081
- dtbinding_microchip_ksz8794
- dtbinding_microchip_ksz8863
- dtbinding_microchip_lan865x
- dtbinding_microchip_mcp230xx

- dtbinding_microchip_mcp23s17
- dtbinding_microchip_mcp23sxx
- dtbinding_microchip_mcp2515
- dtbinding_microchip_mcp251xfd
- dtbinding_microchip_mcp3204
- dtbinding_microchip_mcp3208
- dtbinding_microchip_mcp4725
- dtbinding_microchip_mcp4728
- dtbinding_microchip_mcp7940n
- dtbinding_microchip_mcp9600
- dtbinding_microchip_mcp970x
- dtbinding_microchip_mcp9808
- dtbinding_microchip_mpfs_gpio
- dtbinding_microchip_mpfs_i2c
- dtbinding_microchip_mpfs_qspi
- dtbinding_microchip_mpfs_spi
- dtbinding_microchip_tcn75a
- dtbinding_microchip_xec_adc
- dtbinding_microchip_xec_bbled
- dtbinding_microchip_xec_bbram
- dtbinding_microchip_xec_dmac
- dtbinding_microchip_xec_ecia
- dtbinding_microchip_xec_ecia_girq
- dtbinding_microchip_xec_ecs
- dtbinding_microchip_xec_eeprom
- dtbinding_microchip_xec_espi
- dtbinding_microchip_xec_espi_host_dev
- dtbinding_microchip_xec_espi_saf
- dtbinding_microchip_xec_espi_saf_v2
- dtbinding_microchip_xec_espi_v2
- dtbinding_microchip_xec_espi_vw_routing
- dtbinding_microchip_xec_gpio
- dtbinding_microchip_xec_gpio_v2
- dtbinding_microchip_xec_i2c
- dtbinding_microchip_xec_i2c_v2
- dtbinding_microchip_xec_kbd
- dtbinding_microchip_xec_pcr
- dtbinding_microchip_xec_peci
- dtbinding_microchip_xec_pinctrl

- dtbinding_microchip_xec_ps2
- dtbinding_microchip_xec_pwm
- dtbinding_microchip_xec_pwmbbled
- dtbinding_microchip_xec_qmspi
- dtbinding_microchip_xec_qmspi_ldma
- dtbinding_microchip_xec_rtos_timer
- dtbinding_microchip_xec_symcr
- dtbinding_microchip_xec_tach
- dtbinding_microchip_xec_timer
- dtbinding_microchip_xec_uart
- dtbinding_microchip_xec_watchdog

Micron Technology Inc. (micron)

- dtbinding_micron_mt25qu02g

Motorola, Inc. (motorola)

- dtbinding_motorola_mc146818

Murata Manufacturing Co., Ltd. (murata)

- dtbinding_murata_ncp15wb473
- dtbinding_murata_ncp15xh103

National Semiconductor (national)

- dtbinding_national_lm95234

Nordic Semiconductor (nordic)

- dtbinding_nordic_mbox_nrf_ipc
- dtbinding_nordic_mram
- dtbinding_nordic_npm1100
- dtbinding_nordic_npm1300
- dtbinding_nordic_npm1300_charger
- dtbinding_nordic_npm1300_gpio
- dtbinding_nordic_npm1300_led
- dtbinding_nordic_npm1300_regulator
- dtbinding_nordic_npm1300_wdt
- dtbinding_nordic_npm6001
- dtbinding_nordic_npm6001_gpio
- dtbinding_nordic_npm6001_regulator
- dtbinding_nordic_npm6001_wdt

- dtbinding_nordic_nrf_acl
- dtbinding_nordic_nrf_adc
- dtbinding_nordic_nrf_auxpll
- dtbinding_nordic_nrf_bellboard_rx
- dtbinding_nordic_nrf_bellboard_tx
- dtbinding_nordic_nrf_bprot
- dtbinding_nordic_nrf_can
- dtbinding_nordic_nrf_ccm
- dtbinding_nordic_nrf_clic
- dtbinding_nordic_nrf_clock
- dtbinding_nordic_nrf_comp
- dtbinding_nordic_nrf_ctrlapperi
- dtbinding_nordic_nrf_dcnf
- dtbinding_nordic_nrf_dppic
- dtbinding_nordic_nrf_dppic_global
- dtbinding_nordic_nrf_dppic_local
- dtbinding_nordic_nrf_ecb
- dtbinding_nordic_nrf_egu
- dtbinding_nordic_nrf_exmif
- dtbinding_nordic_nrf_ficr
- dtbinding_nordic_nrf_gpio
- dtbinding_nordic_nrf_gpio_forwarder
- dtbinding_nordic_nrf_gpiote
- dtbinding_nordic_nrf_gpreget
- dtbinding_nordic_nrf_grtc
- dtbinding_nordic_nrf_hfxo
- dtbinding_nordic_nrf_hsfll
- dtbinding_nordic_nrf_i2s
- dtbinding_nordic_nrf_ieee802154
- dtbinding_nordic_nrf_ipc
- dtbinding_nordic_nrf_ipct_global
- dtbinding_nordic_nrf_ipct_local
- dtbinding_nordic_nrf_kmu
- dtbinding_nordic_nrf_led_matrix
- dtbinding_nordic_nrf_lfxo
- dtbinding_nordic_nrf_lpcomp
- dtbinding_nordic_nrf_mpu
- dtbinding_nordic_nrf_mutex
- dtbinding_nordic_nrf_mwu

- dtbinding_nordic_nrf_nfct
- dtbinding_nordic_nrf_oscillators
- dtbinding_nordic_nrf_pdm
- dtbinding_nordic_nrf_pinctrl
- dtbinding_nordic_nrf_power
- dtbinding_nordic_nrf_ppi
- dtbinding_nordic_nrf_pwm
- dtbinding_nordic_nrf_qdec
- dtbinding_nordic_nrf_qspi
- dtbinding_nordic_nrf_radio
- dtbinding_nordic_nrf_regulators
- dtbinding_nordic_nrf_reset
- dtbinding_nordic_nrf_resetinfo
- dtbinding_nordic_nrf_rng
- dtbinding_nordic_nrf_rtc
- dtbinding_nordic_nrf_saadc
- dtbinding_nordic_nrf_spi
- dtbinding_nordic_nrf_spim
- dtbinding_nordic_nrf_spis
- dtbinding_nordic_nrf_spu
- dtbinding_nordic_nrf_sw_pwm
- dtbinding_nordic_nrf_swi
- dtbinding_nordic_nrf_temp
- dtbinding_nordic_nrf_temp_nrfs
- dtbinding_nordic_nrf_timer
- dtbinding_nordic_nrf_twi
- dtbinding_nordic_nrf_twim
- dtbinding_nordic_nrf_twis
- dtbinding_nordic_nrf_uart
- dtbinding_nordic_nrf_uarte
- dtbinding_nordic_nrf_uicr
- dtbinding_nordic_nrf_uicr_v2
- dtbinding_nordic_nrf_usbd
- dtbinding_nordic_nrf_usbreg
- dtbinding_nordic_nrf_vevif_event_rx
- dtbinding_nordic_nrf_vevif_event_tx
- dtbinding_nordic_nrf_vevif_task_rx
- dtbinding_nordic_nrf_vevif_task_tx
- dtbinding_nordic_nrf_vmc

- dtbinding_nordic_nrf_vpr_coprocessor
- dtbinding_nordic_nrf_wdt
- dtbinding_nordic_nrf21540_fem
- dtbinding_nordic_nrf21540_fem_spi
- dtbinding_nordic_nrf51_flash_controller
- dtbinding_nordic_nrf52_flash_controller
- dtbinding_nordic_nrf53_flash_controller
- dtbinding_nordic_nrf91_flash_controller
- dtbinding_nordic_nrf91_slm
- dtbinding_nordic_owned_memory
- dtbinding_nordic_owned_partitions
- dtbinding_nordic_qspi_nor
- dtbinding_nordic_rram_controller
- dtbinding_nordic_vpr

Noritake Co., Inc. Electronics Division (noritake)

- dtbinding_noritake_itron

Nuclei System Technology (nuclei)

- dtbinding_nuclei_bumblebee
- dtbinding_nuclei_eclic
- dtbinding_nuclei_systimer

Nuvoton Technology Corporation (nuvoton)

- dtbinding_nuvoton_adc_cmp
- dtbinding_nuvoton_nct38xx
- dtbinding_nuvoton_nct38xx_gpio
- dtbinding_nuvoton_nct38xx_gpio_alert
- dtbinding_nuvoton_nct38xx_gpio_port
- dtbinding_nuvoton_npcx_adc
- dtbinding_nuvoton_npcx_bbram
- dtbinding_nuvoton_npcx_booter_variant
- dtbinding_nuvoton_npcx_drbg
- dtbinding_nuvoton_npcx_espi
- dtbinding_nuvoton_npcx_espi_taf
- dtbinding_nuvoton_npcx_espi_vw_conf
- dtbinding_nuvoton_npcx_fiu_nor
- dtbinding_nuvoton_npcx_fiu_qspi
- dtbinding_nuvoton_npcx_gpio

- dtbinding_nuvoton_npcx_host_sub
- dtbinding_nuvoton_npcx_host_uart
- dtbinding_nuvoton_npcx_i2c_ctrl
- dtbinding_nuvoton_npcx_i2c_port
- dtbinding_nuvoton_npcx_i3c
- dtbinding_nuvoton_npcx_itim_timer
- dtbinding_nuvoton_npcx_kbd
- dtbinding_nuvoton_npcx_leakage_io
- dtbinding_nuvoton_npcx_lvolctrl_conf
- dtbinding_nuvoton_npcx_miwu
- dtbinding_nuvoton_npcx_miwu_int_map
- dtbinding_nuvoton_npcx_miwu_wui_map
- dtbinding_nuvoton_npcx_pcc
- dtbinding_nuvoton_npcx_peci
- dtbinding_nuvoton_npcx_pinctrl
- dtbinding_nuvoton_npcx_pinctrl_conf
- dtbinding_nuvoton_npcx_pinctrl_def
- dtbinding_nuvoton_npcx_power_psl
- dtbinding_nuvoton_npcx_ps2_channel
- dtbinding_nuvoton_npcx_ps2_ctrl
- dtbinding_nuvoton_npcx_pwm
- dtbinding_nuvoton_npcx_rst
- dtbinding_nuvoton_npcx_scfg
- dtbinding_nuvoton_npcx_sha
- dtbinding_nuvoton_npcx_shi
- dtbinding_nuvoton_npcx_shi_enhanced
- dtbinding_nuvoton_npcx_soc_id
- dtbinding_nuvoton_npcx_spip
- dtbinding_nuvoton_npcx_tach
- dtbinding_nuvoton_npcx_uart
- dtbinding_nuvoton_npcx_watchdog
- dtbinding_nuvoton_numaker_adc
- dtbinding_nuvoton_numaker_canfd
- dtbinding_nuvoton_numaker_ethernet
- dtbinding_nuvoton_numaker_fmc
- dtbinding_nuvoton_numaker_gpio
- dtbinding_nuvoton_numaker_i2c
- dtbinding_nuvoton_numaker_pcc
- dtbinding_nuvoton_numaker_pinctrl

- dtbinding_nuvoton_numaker_pwm
- dtbinding_nuvoton_numaker_rmc
- dtbinding_nuvoton_numaker_rst
- dtbinding_nuvoton_numaker_rtc
- dtbinding_nuvoton_numaker_scc
- dtbinding_nuvoton_numaker_spi
- dtbinding_nuvoton_numaker_uart
- dtbinding_nuvoton_numaker_usbd
- dtbinding_nuvoton_numaker_wwdt
- dtbinding_nuvoton_numicro_gpio
- dtbinding_nuvoton_numicro_pinctrl
- dtbinding_nuvoton_numicro_uart

NXP Semiconductors (nxp)

- dtbinding_nxp_bt_hci_uart
- dtbinding_nxp_cam_44pins_connector
- dtbinding_nxp_ctimer_pwm
- dtbinding_nxp_dai_esai
- dtbinding_nxp_dai_sai
- dtbinding_nxp_dcnano_lcdif
- dtbinding_nxp_dmic
- dtbinding_nxp_edma
- dtbinding_nxp_ehci
- dtbinding_nxp_enet
- dtbinding_nxp_enet_mac
- dtbinding_nxp_enet_mdio
- dtbinding_nxp_enet_ptp_clock
- dtbinding_nxp_enet_qos
- dtbinding_nxp_enet_qos_mac
- dtbinding_nxp_enet_qos_mdio
- dtbinding_nxp_enet1g
- dtbinding_nxp_flexcan
- dtbinding_nxp_flexcan_fd
- dtbinding_nxp_flexio
- dtbinding_nxp_flexio_pwm
- dtbinding_nxp_flexio_spi
- dtbinding_nxp_flexpwm
- dtbinding_nxp_flexram
- dtbinding_nxp_fs26_wdog

- dtbinding_nxp_fxas21002_spi
- dtbinding_nxp_fxas21002_i2c
- dtbinding_nxp_fxos8700_i2c
- dtbinding_nxp_fxos8700_spi
- dtbinding_nxp_gau_adc
- dtbinding_nxp_gau_dac
- dtbinding_nxp_gpio_cluster
- dtbinding_nxp_gpt_hw_timer
- dtbinding_nxp_hci_ble
- dtbinding_nxp_i2c_tsc_fpc
- dtbinding_nxp_iap_fmc11
- dtbinding_nxp_iap_fmc54
- dtbinding_nxp_iap_fmc55
- dtbinding_nxp_iap_fmc553
- dtbinding_nxp_iap_msf1
- dtbinding_nxp_imx_anatop
- dtbinding_nxp_imx_caam
- dtbinding_nxp_imx_ccm
- dtbinding_nxp_imx_ccm_fnpll
- dtbinding_nxp_imx_ccm_rev2
- dtbinding_nxp_imx_csi
- dtbinding_nxp_imx_dtcm
- dtbinding_nxp_imx_ecspi
- dtbinding_nxp_imx_elcdif
- dtbinding_nxp_imx_epit
- dtbinding_nxp_imx_flexspi
- dtbinding_nxp_imx_flexspi_aps6408l
- dtbinding_nxp_imx_flexspi_hyperflash
- dtbinding_nxp_imx_flexspi_is66wvq8m4
- dtbinding_nxp_imx_flexspi_mx25um51345g
- dtbinding_nxp_imx_flexspi_nor
- dtbinding_nxp_imx_flexspi_s27ks0641
- dtbinding_nxp_imx_flexspi_w956a8mbya
- dtbinding_nxp_imx_gpio
- dtbinding_nxp_imx_gpr
- dtbinding_nxp_imx_gpt
- dtbinding_nxp_imx_iomuxc
- dtbinding_nxp_imx_iomuxc_scu
- dtbinding_nxp_imx_itcm

- dtbinding_nxp_imx_iuart
- dtbinding_nxp_imx_lpi2c
- dtbinding_nxp_imx_lpspi
- dtbinding_nxp_imx_mipi_dsi
- dtbinding_nxp_imx_mu
- dtbinding_nxp_imx_pwm
- dtbinding_nxp_imx_qtmr
- dtbinding_nxp_imx_rgpio
- dtbinding_nxp_imx_semc
- dtbinding_nxp_imx_snvs_rtc
- dtbinding_nxp_imx_tmr
- dtbinding_nxp_imx_uart
- dtbinding_nxp_imx_usdhc
- dtbinding_nxp_imx_wdog
- dtbinding_nxp_imx7d_pinctrl
- dtbinding_nxp_imx8_pinctrl
- dtbinding_nxp_imx8m_pinctrl
- dtbinding_nxp_imx8mp_pinctrl
- dtbinding_nxp_imx8ulp_pinctrl
- dtbinding_nxp_imx93_pinctrl
- dtbinding_nxp_irqsteer_intc
- dtbinding_nxp_irqsteer_master
- dtbinding_nxp_kinetis_acmp
- dtbinding_nxp_kinetis_adc12
- dtbinding_nxp_kinetis_adc16
- dtbinding_nxp_kinetis_dac
- dtbinding_nxp_kinetis_dac32
- dtbinding_nxp_kinetis_dspi
- dtbinding_nxp_kinetis_ethernet
- dtbinding_nxp_kinetis_ftfa
- dtbinding_nxp_kinetis_ftfe
- dtbinding_nxp_kinetis_ftfl
- dtbinding_nxp_kinetis_ftm
- dtbinding_nxp_kinetis_ftm_pwm
- dtbinding_nxp_kinetis_gpio
- dtbinding_nxp_kinetis_i2c
- dtbinding_nxp_kinetis_ke1xf_sim
- dtbinding_nxp_kinetis_lpsci
- dtbinding_nxp_kinetis_lptmr

- dtbinding_nxp_kinetis_lpuart
- dtbinding_nxp_kinetis_mcg
- dtbinding_nxp_kinetis_pcc
- dtbinding_nxp_kinetis_pinctrl
- dtbinding_nxp_kinetis_pinmux
- dtbinding_nxp_kinetis_ptp
- dtbinding_nxp_kinetis_pwt
- dtbinding_nxp_kinetis_rnga
- dtbinding_nxp_kinetis_rtc
- dtbinding_nxp_kinetis_scg
- dtbinding_nxp_kinetis_sim
- dtbinding_nxp_kinetis_temperature
- dtbinding_nxp_kinetis_tpm
- dtbinding_nxp_kinetis_trng
- dtbinding_nxp_kinetis_uart
- dtbinding_nxp_kinetis_usbd
- dtbinding_nxp_kinetis_wdog
- dtbinding_nxp_kinetis_wdog32
- dtbinding_nxp_kw41z_ieee802154
- dtbinding_nxp_lcdic
- dtbinding_nxp_lp_flexcomm
- dtbinding_nxp_lpc_ctimer
- dtbinding_nxp_lpc_dma
- dtbinding_nxp_lpc_flexcomm
- dtbinding_nxp_lpc_gpio
- dtbinding_nxp_lpc_gpio_port
- dtbinding_nxp_lpc_i2c
- dtbinding_nxp_lpc_i2s
- dtbinding_nxp_lpc_iocon
- dtbinding_nxp_lpc_iocon_pinctrl
- dtbinding_nxp_lpc_iocon_pio
- dtbinding_nxp_lpc_lpadc
- dtbinding_nxp_lpc_mailbox
- dtbinding_nxp_lpc_mcan
- dtbinding_nxp_lpc_rng
- dtbinding_nxp_lpc_rtc
- dtbinding_nxp_lpc_rtc_highres
- dtbinding_nxp_lpc_sdif
- dtbinding_nxp_lpc_spi

- dtbinding_nxp_lpc_syscon
- dtbinding_nxp_lpc_syscon_reset
- dtbinding_nxp_lpc_uid
- dtbinding_nxp_lpc_usart
- dtbinding_nxp_lpc_wwdt
- dtbinding_nxp_lpc11u6x_eeprom
- dtbinding_nxp_lpc11u6x_gpio
- dtbinding_nxp_lpc11u6x_i2c
- dtbinding_nxp_lpc11u6x_pinctrl
- dtbinding_nxp_lpc11u6x_syscon
- dtbinding_nxp_lpc11u6x_uart
- dtbinding_nxp_lpcip3511
- dtbinding_nxp_lpcmp
- dtbinding_nxp_lpdac
- dtbinding_nxp_lptmr
- dtbinding_nxp_mbox_imx_mu
- dtbinding_nxp_mbox_mailbox
- dtbinding_nxp_mci_io_mux
- dtbinding_nxp_mcr20a
- dtbinding_nxp_mcux_12b1msps_sar
- dtbinding_nxp_mcux_dcp
- dtbinding_nxp_mcux_edma
- dtbinding_nxp_mcux_edma_v3
- dtbinding_nxp_mcux_edma_v4
- dtbinding_nxp_mcux_i2s
- dtbinding_nxp_mcux_i3c
- dtbinding_nxp_mcux_qdec
- dtbinding_nxp_mcux_rt_pinctrl
- dtbinding_nxp_mcux_rt11xx_pinctrl
- dtbinding_nxp_mcux_xbar
- dtbinding_nxp_mipi_csi2rx
- dtbinding_nxp_mipi_dbi_flexio_lcdif
- dtbinding_nxp_mipi_dsi_2l
- dtbinding_nxp_mrt
- dtbinding_nxp_mrt_channel
- dtbinding_nxp_nx20p3483
- dtbinding_nxp_os_timer
- dtbinding_nxp_parallel_lcd_connector
- dtbinding_nxp_pca9420

- dtbinding_nxp_pca95xx
- dtbinding_nxp_pca9633
- dtbinding_nxp_pca9685
- dtbinding_nxp_pcal6408a
- dtbinding_nxp_pcal6416a
- dtbinding_nxp_pcf8523
- dtbinding_nxp_pcf8563
- dtbinding_nxp_pcf857x
- dtbinding_nxp_pdcfg_power
- dtbinding_nxp_pint
- dtbinding_nxp_pit
- dtbinding_nxp_pit_channel
- dtbinding_nxp_pxp
- dtbinding_nxp_s32_qdec
- dtbinding_nxp_qtmr_pwm
- dtbinding_nxp_rdc
- dtbinding_nxp_rstctl
- dtbinding_nxp_rt_iocon_pinctrl
- dtbinding_nxp_rw_pmu
- dtbinding_nxp_rw_soc_ctrl
- dtbinding_nxp_s32_adc_sar
- dtbinding_nxp_s32_canxl
- dtbinding_nxp_s32_clock
- dtbinding_nxp_s32_emios
- dtbinding_nxp_s32_emios_pwm
- dtbinding_nxp_s32_gmac
- dtbinding_nxp_s32_gmac_mdio
- dtbinding_nxp_s32_gpio
- dtbinding_nxp_s32_lcu
- dtbinding_nxp_s32_linflexd
- dtbinding_nxp_s32_mru
- dtbinding_nxp_s32_netc_emdio
- dtbinding_nxp_s32_netc_psi
- dtbinding_nxp_s32_netc_vsi
- dtbinding_nxp_s32_qspi
- dtbinding_nxp_s32_qspi_device
- dtbinding_nxp_s32_qspi_nor
- dtbinding_nxp_s32_siul2_eirq
- dtbinding_nxp_s32_spi

- dtbinding_nxp_s32_swt
- dtbinding_nxp_s32_sys_timer
- dtbinding_nxp_s32_trgmux
- dtbinding_nxp_s32_wkpu
- dtbinding_nxp_s32k3_pinctrl
- dtbinding_nxp_s32ze_pinctrl
- dtbinding_nxp_sc18im704
- dtbinding_nxp_sc18im704_gpio
- dtbinding_nxp_sc18im704_i2c
- dtbinding_nxp_sctimer_pwm
- dtbinding_nxp_smartdma
- dtbinding_nxp_sof_host_dma
- dtbinding_nxp_tempmon
- dtbinding_nxp_tja1103
- dtbinding_nxp_tpm_timer
- dtbinding_nxp_usbphy
- dtbinding_nxp_vf610_adc
- dtbinding_nxp_vref

OmniVision Technologies Co., Ltd. (ovti)

- dtbinding_ovti_ov2640
- dtbinding_ovti_ov5640
- dtbinding_ovti_ov7670
- dtbinding_ovti_ov7725

ON Semiconductor Corp. (onnn)

- dtbinding_onnn_ncp5623
- dtbinding_onnn_nct75

open-isa.org (openisa)

- dtbinding_openisa_ri5cy
- dtbinding_openisa_rv32m1_event_unit
- dtbinding_openisa_rv32m1_ftfe
- dtbinding_openisa_rv32m1_genfsk
- dtbinding_openisa_rv32m1_gpio
- dtbinding_openisa_rv32m1_intmux
- dtbinding_openisa_rv32m1_intmux_ch
- dtbinding_openisa_rv32m1_lpi2c
- dtbinding_openisa_rv32m1_lpspi

- dtbinding_openisa_rv32m1_lptmr
- dtbinding_openisa_rv32m1_lpuart
- dtbinding_openisa_rv32m1_pcc
- dtbinding_openisa_rv32m1_pinctrl
- dtbinding_openisa_rv32m1_pinmux
- dtbinding_openisa_rv32m1_tpm
- dtbinding_openisa_rv32m1_trng
- dtbinding_openisa_zero_ri5cy

OpenCores.org (opencores)

- dtbinding_opencores_spi_simple

OpenThread.io (openthread)

- dtbinding_openthread_config

Orise Technology (orisetech)

- dtbinding_orisetech_otm8009a

Panasonic Corporation (panasonic)

- dtbinding_panasonic_amg88xx
- dtbinding_panasonic_reduced_arduino_header

PixArt Imaging Inc. (pixart)

- dtbinding_pixart_pat912x
- dtbinding_pixart_paw32xx
- dtbinding_pixart_pmw3610

Plantower Co., Ltd (plantower)

- dtbinding_plantower_pms7003

Princeton Technology Corp. (ptc)

- dtbinding_ptc_pt6314

QEMU, a generic and open source machine emulator and virtualizer (qemu)

- dtbinding_qemu_ivshmem
- dtbinding_qemu_nios2_zephyr
- dtbinding_qemu_riscv_virt

Qorvo, Inc (formerly Decawave) (decawave)

- dtbinding_decawave_dw1000

Quectel Wireless Solutions Co., Ltd. (quectel)

- dtbinding_quectel_bg95
- dtbinding_quectel_bg9x
- dtbinding_quectel_eg25_g
- dtbinding_quectel_lc26g
- dtbinding_quectel_lc76g
- dtbinding_quectel_lc86g

QuickLogic Corp. (quicklogic)

- dtbinding_quicklogic_eos_s3_gpio
- dtbinding_quicklogic_eos_s3_pinctrl
- dtbinding_quicklogic_usbserialport_s3b

Raspberry Pi Foundation (raspberrypi)

- dtbinding_raspberrypi_core_supply_regulator
- dtbinding_raspberrypi_pico_adc
- dtbinding_raspberrypi_pico_clock
- dtbinding_raspberrypi_pico_clock_controller
- dtbinding_raspberrypi_pico_dma
- dtbinding_raspberrypi_pico_flash_controller
- dtbinding_raspberrypi_pico_gpio
- dtbinding_raspberrypi_pico_header
- dtbinding_raspberrypi_pico_i2c
- dtbinding_raspberrypi_pico_pinctrl
- dtbinding_raspberrypi_pico_pio
- dtbinding_raspberrypi_pico_pio_device
- dtbinding_raspberrypi_pico_pll
- dtbinding_raspberrypi_pico_pwm
- dtbinding_raspberrypi_pico_reset
- dtbinding_raspberrypi_pico_rosc
- dtbinding_raspberrypi_pico_rtc
- dtbinding_raspberrypi_pico_spi
- dtbinding_raspberrypi_pico_spi_pio
- dtbinding_raspberrrypi_pico_temp
- dtbinding_raspberrypi_pico_timer
- dtbinding_raspberrypi_pico_uart

- dtbinding_raspberrypi_pico_uart_pio
- dtbinding_raspberrypi_pico_usbd
- dtbinding_raspberrypi_pico_watchdog

Raydium Semiconductor Corp. (raydium)

- dtbinding_raydium_rm67162
- dtbinding_raydium_rm68200

Realtek Semiconductor Corp. (realtek)

- dtbinding_realtek_rtl8211f

Renesas Electronics Corporation (renesas)

- dtbinding_renesas_bt_hci_da1469x
- dtbinding_renesas_hs300x
- dtbinding_renesas_pwm_rcar
- dtbinding_renesas_r8a7795_cpg_mssr
- dtbinding_renesas_r8a779f0_cpg_mssr
- dtbinding_renesas_ra_clock_generation_circuit
- dtbinding_renesas_ra_gpio
- dtbinding_renesas_ra_interrupt_controller_unit
- dtbinding_renesas_ra_pinctrl
- dtbinding_renesas_ra_sci
- dtbinding_renesas_ra_uart_sci
- dtbinding_renesas_ra8_cgc_busclk
- dtbinding_renesas_ra8_cgc_external_clock
- dtbinding_renesas_ra8_cgc_pclk
- dtbinding_renesas_ra8_cgc_pclk_block
- dtbinding_renesas_ra8_cgc_pll
- dtbinding_renesas_ra8_cgc_subclk
- dtbinding_renesas_ra8_gpio
- dtbinding_renesas_ra8_pinctrl
- dtbinding_renesas_ra8_uart_sci_b
- dtbinding_renesas_rcar_can
- dtbinding_renesas_rcar_cmt
- dtbinding_renesas_rcar_gpio
- dtbinding_renesas_rcar_hscif
- dtbinding_renesas_rcar_i2c
- dtbinding_renesas_rcar_emmc
- dtbinding_renesas_rcar_pfc

- dtbinding_renesas_rcar_scif
- dtbinding_renesas_rzt2m_gpio
- dtbinding_renesas_rzt2m_gpio_common
- dtbinding_renesas_rzt2m_pinctrl
- dtbinding_renesas_rzt2m_uart
- dtbinding_renesas_smartbond_gpadc
- dtbinding_renesas_smartbond_crypto
- dtbinding_renesas_smartbond_display
- dtbinding_renesas_smartbond_dma
- dtbinding_renesas_smartbond_flash_controller
- dtbinding_renesas_smartbond_gpio
- dtbinding_renesas_smartbond_i2c
- dtbinding_renesas_smartbond_lp_clock
- dtbinding_renesas_smartbond_lp_osc
- dtbinding_renesas_smartbond_mipi_dbi
- dtbinding_renesas_smartbond_nor_psram
- dtbinding_renesas_smartbond_pinctrl
- dtbinding_renesas_da1469x_regulator
- dtbinding_renesas_smartbond_rtc
- dtbinding_renesas_smartbond_sdadc
- dtbinding_renesas_smartbond_spi
- dtbinding_renesas_smartbond_sys_clock
- dtbinding_renesas_smartbond_timer
- dtbinding_renesas_smartbond_trng
- dtbinding_renesas_smartbond_uart
- dtbinding_renesas_smartbond_usbd
- dtbinding_renesas_smartbond_watchdog

Reyax Technology Co., Ltd. (reyax)

- dtbinding_reyax_rylrxxx

Richtek Technology Corporation (richtek)

- dtbinding_richtek_rt1718s
- dtbinding_richtek_rt1718s_gpio_port

RISC-V Foundation (riscv)

- dtbinding_riscv_cpu_intc

ROCKTECH DISPLAYS LIMITED (rocktech)

- dtbinding_rocktech_rk043fn02h_ct

ROHM Semiconductor Co., Ltd (rohm)

- dtbinding_rohm_bd8lb600fs
- dtbinding_rohm_bd8lb600fs_diagnostics
- dtbinding_rohm_bd8lb600fs_gpio
- dtbinding_rohm_bh1750

Sciosense B.V. (sciosense)

- dtbinding_sciosense_ens160_spi
- dtbinding_sciosense_ens160_common
- dtbinding_sciosense_ens160_i2c

Seeed Technology Co., Ltd (seeed)

- dtbinding_seeed_grove_lcd_rgb
- dtbinding_seeed_grove_light
- dtbinding_seeed_grove_temperature
- dtbinding_seeed_hm330x
- dtbinding_seeed_xiao_header

SEGGER Microcontroller GmbH (segger)

- dtbinding_segger_rtt_uart

Semtech Corporation (semtech)

- dtbinding_semtech_sx1261
- dtbinding_semtech_sx1262
- dtbinding_semtech_sx1272
- dtbinding_semtech_sx1276
- dtbinding_semtech_sx1509b
- dtbinding_semtech_sx9500

Sensirion AG (sensirion)

- dtbinding_sensirion_sgp40
- dtbinding_sensirion_sht21
- dtbinding_sensirion_sht3xd
- dtbinding_sensirion_sht4x
- dtbinding_sensirion_shtcx

Sequans Communications (sqn)

- dtbinding_sqn_gm02s
- dtbinding_sqn_hwspinlock

Sharp Corporation (sharp)

- dtbinding_sharp_ls0xx

Shenzhen Frida LCD Co., Ltd. (frida)

- dtbinding_frida_nt35510

Shenzhen Huiding Technology Co., Ltd. (goodix)

- dtbinding_goodix_gt911

Shenzhen Jinghua Displays Electronics Co., Ltd. (jhd)

- dtbinding_jhd_jhd1313

Shenzhen Xptek Technology Co., Ltd (xptek)

- dtbinding_xptek_xpt2046

Siemens AG (siemens)

- dtbinding_siemens_ivshmem_eth

Sierra Wireless (swir)

- dtbinding_swir_hl7800

SiFive, Inc. (sifive)

- dtbinding_sifive_clint0
- dtbinding_sifive_dtim0
- dtbinding_sifive_e24
- dtbinding_sifive_e31
- dtbinding_sifive_e51
- dtbinding_sifive_fu740_c000_ddr
- dtbinding_sifive_gpio0
- dtbinding_sifive_i2c0
- dtbinding_sifive_pinctrl
- dtbinding_sifive_plic_1.0.0
- dtbinding_sifive_pwm0
- dtbinding_sifive_s7
- dtbinding_sifive_spi0

- dtbinding_sifive_u54
- dtbinding_sifive_uart0
- dtbinding_sifive_wdt

Silicon Laboratories (silabs)

- dtbinding_silabs_bt_hci
- dtbinding_silabs_gecko_adc
- dtbinding_silabs_gecko_burtc
- dtbinding_silabs_gecko_ethernet
- dtbinding_silabs_gecko_flash_controller
- dtbinding_silabs_gecko_gpio
- dtbinding_silabs_gecko_gpio_port
- dtbinding_silabs_gecko_i2c
- dtbinding_silabs_gecko_iadc
- dtbinding_silabs_gecko_leuart
- dtbinding_silabs_gecko_pinctrl
- dtbinding_silabs_gecko_pwm
- dtbinding_silabs_gecko_rtcc
- dtbinding_silabs_gecko_semailbox
- dtbinding_silabs_gecko_spi_usart
- dtbinding_silabs_gecko_stimer
- dtbinding_silabs_gecko_timer
- dtbinding_silabs_gecko_trng
- dtbinding_silabs_gecko_uart
- dtbinding_silabs_gecko_usart
- dtbinding_silabs_gecko_wdog
- dtbinding_silabs_hfxo
- dtbinding_silabs_si7006
- dtbinding_silabs_si7055
- dtbinding_silabs_si7060
- dtbinding_silabs_si7210

SIMCom Wireless Solutions Co., LTD (simcom)

- dtbinding_simcom_sim7080

Sino Wealth Electronic Ltd (sinowealth)

- dtbinding_sinowealth_sh1106_i2c
- dtbinding_sinowealth_sh1106_spi

Sitronix Technology Corporation (sitronix)

- dtbinding_sitronix_cf1133
- dtbinding_sitronix_st7735r
- dtbinding_sitronix_st7789v
- dtbinding_sitronix_st7796s

Skyworks Solutions, Inc. (skyworks)

- dtbinding_skyworks_sky13317
- dtbinding_skyworks_sky13351

Smart Battery System (sbs)

- dtbinding_sbs_default_sbs_gauge
- dtbinding_sbs_sbs_charger
- dtbinding_sbs_sbs_gauge
- dtbinding_sbs_sbs_gauge_new_api

Solomon Systech Limited (solomon)

- dtbinding_solomon_ssd1306fb_i2c
- dtbinding_solomon_ssd1306fb_spi
- dtbinding_solomon_ssd1608
- dtbinding_solomon_ssd1673
- dtbinding_solomon_ssd1675a
- dtbinding_solomon_ssd1680
- dtbinding_solomon_ssd1681

SparkFun Electronics (sparkfun)

- dtbinding_sparkfun_micromod_gpio
- dtbinding_sparkfun_pro_micro_header
- dtbinding_sparkfun_serlcd

Standard Microsystems Corporation (smsc)

- dtbinding_smsc_lan91c111
- dtbinding_smsc_lan91c111_mdio
- dtbinding_smsc_lan9220

StarFive Technology Co. Ltd. (starfive)

- dtbinding_starfive_jh7100_clint

STMicroelectronics (st)

- dtbinding_st_dsi_lcd_qsh_030
- dtbinding_st_hci_spi_v1
- dtbinding_st_hci_spi_v2
- dtbinding_st_hci_stm32wba
- dtbinding_st_hts221_spi
- dtbinding_st_hts221_i2c
- dtbinding_st_i3g4250d
- dtbinding_st_iis2dh_spi
- dtbinding_st_iis2dh_i2c
- dtbinding_st_iis2dlpc_i2c
- dtbinding_st_iis2dlpc_spi
- dtbinding_st_iis2iclx_spi
- dtbinding_st_iis2iclx_i2c
- dtbinding_st_iis2mdc_spi
- dtbinding_st_iis2mdc_i2c
- dtbinding_st_iis328dq_spi
- dtbinding_st_iis328dq_i2c
- dtbinding_st_iis3dhhc_spi
- dtbinding_st_ism330dhcx_i2c
- dtbinding_st_ism330dhcx_spi
- dtbinding_st_lis2de12_i2c
- dtbinding_st_lis2de12_spi
- dtbinding_st_lis2dh_spi
- dtbinding_st_lis2dh_i2c
- dtbinding_st_lis2dh12_i2c
- dtbinding_st_lis2ds12_spi
- dtbinding_st_lis2ds12_i2c
- dtbinding_st_lis2du12_i2c
- dtbinding_st_lis2du12_spi
- dtbinding_st_lis2dux12_spi
- dtbinding_st_lis2dux12_i2c
- dtbinding_st_lis2dw12_i2c
- dtbinding_st_lis2dw12_spi
- dtbinding_st_lis2mdl_i2c
- dtbinding_st_lis2mdl_spi
- dtbinding_st_lis3dh_i2c
- dtbinding_st_lis3mdl_magn
- dtbinding_st_lps22df_i3c

- dtbinding_st_lps22df_i2c
- dtbinding_st_lps22df_spi
- dtbinding_st_lps22hb_press
- dtbinding_st_lps22hh_spi
- dtbinding_st_lps22hh_i3c
- dtbinding_st_lps22hh_i2c
- dtbinding_st_lps25hb_press
- dtbinding_st_lps28dfw_i3c
- dtbinding_st_lps28dfw_i2c
- dtbinding_st_lsm303agr_accel_spi
- dtbinding_st_lsm303agr_accel_i2c
- dtbinding_st_lsm303dlhc_accel
- dtbinding_st_lsm303dlhc_magn
- dtbinding_st_lsm6ds0
- dtbinding_st_lsm6dsl_i2c
- dtbinding_st_lsm6dsl_spi
- dtbinding_st_lsm6dso_i2c
- dtbinding_st_lsm6dso_spi
- dtbinding_st_lsm6dso16is_spi
- dtbinding_st_lsm6dso16is_i2c
- dtbinding_st_lsm6dso32_i2c
- dtbinding_st_lsm6dso32_spi
- dtbinding_st_lsm6dsv16x_spi
- dtbinding_st_lsm6dsv16x_i2c
- dtbinding_st_lsm9ds0_gyro_i2c
- dtbinding_st_lsm9ds0_mfd_i2c
- dtbinding_st_mbox_stm32_hsem
- dtbinding_st_mpxxdtyy_i2s
- dtbinding_st_stm32_adc
- dtbinding_st_stm32_aes
- dtbinding_st_stm32_backup_sram
- dtbinding_st_stm32_bbram
- dtbinding_st_stm32_bdma
- dtbinding_st_stm32_bxcan
- dtbinding_st_stm32_ccm
- dtbinding_st_stm32_clock_mux
- dtbinding_st_stm32_counter
- dtbinding_st_stm32_cryp
- dtbinding_st_stm32_dac

- dtbinding_st_stm32_dcmi
- dtbinding_st_stm32_digi_temp
- dtbinding_st_stm32_dma
- dtbinding_st_stm32_dma_v1
- dtbinding_st_stm32_dma_v2
- dtbinding_st_stm32_dma_v2bis
- dtbinding_st_stm32_dmamux
- dtbinding_st_stm32_eeprom
- dtbinding_st_stm32_ethernet
- dtbinding_st_stm32_exti
- dtbinding_st_stm32_fdcan
- dtbinding_st_stm32_flash_controller
- dtbinding_st_stm32_fmc
- dtbinding_st_stm32_fmc_nor_psram
- dtbinding_st_stm32_fmc_sdram
- dtbinding_st_stm32_gpio
- dtbinding_st_stm32_hse_clock
- dtbinding_st_stm32_hsem_mailbox
- dtbinding_st_stm32_hsi48_clock
- dtbinding_st_stm32_i2c_v1
- dtbinding_st_stm32_i2c_v2
- dtbinding_st_stm32_i2s
- dtbinding_st_stm32_ipcc_mailbox
- dtbinding_st_stm32_lptim
- dtbinding_st_stm32_lpuart
- dtbinding_st_stm32_lse_clock
- dtbinding_st_stm32_ltdc
- dtbinding_st_stm32_mdio
- dtbinding_st_stm32_mipi_dsi
- dtbinding_st_stm32_msi_clock
- dtbinding_st_stm32_nv_flash
- dtbinding_st_stm32_ospi
- dtbinding_st_stm32_ospi_nor
- dtbinding_st_stm32_otgfs
- dtbinding_st_stm32_otghs
- dtbinding_st_stm32_pinctrl
- dtbinding_st_stm32_pwm
- dtbinding_st_stm32_pwr
- dtbinding_st_stm32_qdec

- dtbinding_st_stm32_qspi
- dtbinding_st_stm32_qspi_nor
- dtbinding_st_stm32_rcc
- dtbinding_st_stm32_rcc_rctl
- dtbinding_st_stm32_rng
- dtbinding_st_stm32_rtc
- dtbinding_st_stm32_sdmmc
- dtbinding_st_stm32_smbus
- dtbinding_st_stm32_spi
- dtbinding_st_stm32_spi_fifo
- dtbinding_st_stm32_spi_host_cmd
- dtbinding_st_stm32_spi_subghz
- dtbinding_st_stm32_temp
- dtbinding_st_stm32_temp_cal
- dtbinding_st_stm32_timers
- dtbinding_st_stm32_uart
- dtbinding_st_stm32_ucpd
- dtbinding_st_stm32_usart
- dtbinding_st_stm32_usb
- dtbinding_st_stm32_usbphyc
- dtbinding_st_stm32_vbat
- dtbinding_st_stm32_vref
- dtbinding_st_stm32_watchdog
- dtbinding_st_stm32_window_watchdog
- dtbinding_st_stm32_xspi
- dtbinding_st_stm32_xspi_nor
- dtbinding_st_stm32c0_hsi_clock
- dtbinding_st_stm32c0_temp_cal
- dtbinding_st_stm32f0_pll_clock
- dtbinding_st_stm32f0_rcc
- dtbinding_st_stm32f1_adc
- dtbinding_st_stm32f1_flash_controller
- dtbinding_st_stm32f1_pinctrl
- dtbinding_st_stm32f1_pll_clock
- dtbinding_st_stm32f1_rcc
- dtbinding_st_stm32f100_pll_clock
- dtbinding_st_stm32f105_pll_clock
- dtbinding_st_stm32f105_pll2_clock
- dtbinding_st_stm32f2_flash_controller

- dtbinding_st_stm32f2_pll_clock
- dtbinding_st_stm32f3_rcc
- dtbinding_st_stm32f4_adc
- dtbinding_st_stm32f4_flash_controller
- dtbinding_st_stm32f4_fsotg
- dtbinding_st_stm32f4_nv_flash
- dtbinding_st_stm32f4_pll_clock
- dtbinding_st_stm32f4_plli2s_clock
- dtbinding_st_stm32f412_plli2s_clock
- dtbinding_st_stm32f7_flash_controller
- dtbinding_st_stm32f7_pll_clock
- dtbinding_st_stm32g0_exti
- dtbinding_st_stm32g0_flash_controller
- dtbinding_st_stm32g0_hsi_clock
- dtbinding_st_stm32g0_pll_clock
- dtbinding_st_stm32g4_flash_controller
- dtbinding_st_stm32g4_pll_clock
- dtbinding_st_stm32h7_fdcan
- dtbinding_st_stm32h7_flash_controller
- dtbinding_st_stm32h7_fmc
- dtbinding_st_stm32h7_hsi_clock
- dtbinding_st_stm32h7_i2s
- dtbinding_st_stm32h7_pll_clock
- dtbinding_st_stm32h7_rcc
- dtbinding_st_stm32h7_spi
- dtbinding_st_stm32h7rs_exti
- dtbinding_st_stm32h7rs_pll_clock
- dtbinding_st_stm32h7rs_rcc
- dtbinding_st_stm32l0_msi_clock
- dtbinding_st_stm32l0_nv_flash
- dtbinding_st_stm32l0_pll_clock
- dtbinding_st_stm32l4_flash_controller
- dtbinding_st_stm32l4_pll_clock
- dtbinding_st_stm32l5_flash_controller
- dtbinding_st_stm32mp1_rcc
- dtbinding_st_stm32u5_dma
- dtbinding_st_stm32u5_msi_clock
- dtbinding_st_stm32u5_pll_clock
- dtbinding_st_stm32u5_rcc

- dtbinding_st_stm32wb_flash_controller
- dtbinding_st_stm32wb_pll_clock
- dtbinding_st_stm32wb_rcc
- dtbinding_st_stm32wb_ble_rf
- dtbinding_st_stm32wba_flash_controller
- dtbinding_st_stm32wba_hse_clock
- dtbinding_st_stm32wba_pll_clock
- dtbinding_st_stm32wba_rcc
- dtbinding_st_stm32wl_hse_clock
- dtbinding_st_stm32wl_rcc
- dtbinding_st_stm32wl_subghz_radio
- dtbinding_st_stmpe1600
- dtbinding_st_stmpe811
- dtbinding_st_stts22h_i2c
- dtbinding_st_stts751_i2c
- dtbinding_st_vl53l0x
- dtbinding_st_vl53l1x

Synopsys, Inc. (snps)

- dtbinding_snps_arc_iot_sysconf
- dtbinding_snps_arc_timer
- dtbinding_snps_arcem
- dtbinding_snps_archs_ici
- dtbinding_snps_archs_idu_intc
- dtbinding_snps_arcv2_intc
- dtbinding_snps_creg_gpio
- dtbinding_snps_designware_dma
- dtbinding_snps_designware_ethernet
- dtbinding_snps_designware_gpio
- dtbinding_snps_designware_i2c
- dtbinding_snps_designware_intc
- dtbinding_snps_designware_spi
- dtbinding_snps_designware_usb
- dtbinding_snps_designware_watchdog
- dtbinding_snps_dw_timers
- dtbinding_snps_dwc2
- dtbinding_snps_emsdp_pinctrl
- dtbinding_snps_ethernet_cyclonev
- dtbinding_snps_hostlink_uart

- dtbinding_snps_nsim_uart

Synopsys, Inc. (formerly ARC International PLC) (arc)

- dtbinding_arc_dccm
- dtbinding_arc_iccm
- dtbinding_arc_xccm
- dtbinding_arc_yccm

TDK Corporation. (tdk)

- dtbinding_tdk_ntcg163jf103ft1

Telink Semiconductor (telink)

- dtbinding_telink_b91
- dtbinding_telink_b91_adc
- dtbinding_telink_b91_flash_controller
- dtbinding_telink_b91_gpio
- dtbinding_telink_b91_i2c
- dtbinding_telink_b91_pinctrl
- dtbinding_telink_b91_power
- dtbinding_telink_b91_pwm
- dtbinding_telink_b91_spi
- dtbinding_telink_b91_trng
- dtbinding_telink_b91_uart
- dtbinding_telink_b91_zb
- dtbinding_telink_machine_timer

Telit Cinterion (telit)

- dtbinding_telit_me910g1

Texas Instruments (ti)

- dtbinding_ti_ads1013
- dtbinding_ti_ads1014
- dtbinding_ti_ads1015
- dtbinding_ti_ads1112
- dtbinding_ti_ads1113
- dtbinding_ti_ads1114
- dtbinding_ti_ads1115
- dtbinding_ti_ads1119
- dtbinding_ti_ads114s08

- dtbinding_ti_ads114s0x_gpio
- dtbinding_ti_ads7052
- dtbinding_ti_am654_dmtimer
- dtbinding_ti_boosterpack_header
- dtbinding_ti_bq24190
- dtbinding_ti_bq25180
- dtbinding_ti_bq274xx
- dtbinding_ti_bq27z746
- dtbinding_ti_cc1200
- dtbinding_ti_cc13xx_cc26xx_adc
- dtbinding_ti_cc13xx_cc26xx_flash_controller
- dtbinding_ti_cc13xx_cc26xx_gpio
- dtbinding_ti_cc13xx_cc26xx_i2c
- dtbinding_ti_cc13xx_cc26xx_ieee802154
- dtbinding_ti_cc13xx_cc26xx_ieee802154_subghz
- dtbinding_ti_cc13xx_cc26xx_pinctrl
- dtbinding_ti_cc13xx_cc26xx_radio
- dtbinding_ti_cc13xx_cc26xx_rtc_timer
- dtbinding_ti_cc13xx_cc26xx_spi
- dtbinding_ti_cc13xx_cc26xx_timer
- dtbinding_ti_cc13xx_cc26xx_timer_pwm
- dtbinding_ti_cc13xx_cc26xx_trng
- dtbinding_ti_cc13xx_cc26xx_uart
- dtbinding_ti_cc13xx_cc26xx_watchdog
- dtbinding_ti_cc2520
- dtbinding_ti_cc32xx_adc
- dtbinding_ti_cc32xx_gpio
- dtbinding_ti_cc32xx_i2c
- dtbinding_ti_cc32xx_pinctrl
- dtbinding_ti_cc32xx_uart
- dtbinding_ti_cc32xx_watchdog
- dtbinding_ti_dac43608
- dtbinding_ti_dac53608
- dtbinding_ti_dac60508
- dtbinding_ti_dac70508
- dtbinding_ti_dac80508
- dtbinding_ti_dacx0501
- dtbinding_ti_davinci_gpio
- dtbinding_ti_davinci_gpio_nexus

- dtbinding_ti_fdc2x1x
- dtbinding_ti_hdc
- dtbinding_ti_hdc2010
- dtbinding_ti_hdc2021
- dtbinding_ti_hdc2022
- dtbinding_ti_hdc2080
- dtbinding_ti_hdc20xx
- dtbinding_ti_ina219
- dtbinding_ti_ina226
- dtbinding_ti_ina230
- dtbinding_ti_ina237
- dtbinding_ti_ina3221
- dtbinding_ti_k3_pinctrl
- dtbinding_ti_lmp90077
- dtbinding_ti_lmp90078
- dtbinding_ti_lmp90079
- dtbinding_ti_lmp90080
- dtbinding_ti_lmp90097
- dtbinding_ti_lmp90098
- dtbinding_ti_lmp90099
- dtbinding_ti_lmp90100
- dtbinding_ti_lmp90xxx_gpio
- dtbinding_ti_lp3943
- dtbinding_ti_lp5009
- dtbinding_ti_lp5012
- dtbinding_ti_lp5018
- dtbinding_ti_lp5024
- dtbinding_ti_lp5030
- dtbinding_ti_lp5036
- dtbinding_ti_lp5562
- dtbinding_ti_lp5569
- dtbinding_ti_msp432p4xx_uart
- dtbinding_ti_opt3001
- dtbinding_ti_sn74hc595
- dtbinding_ti_stellaris_ethernet
- dtbinding_ti_stellaris_flash_controller
- dtbinding_ti_stellaris_gpio
- dtbinding_ti_stellaris_uart
- dtbinding_ti_tas6422dac

- dtbinding_ti_tca6424a
- dtbinding_ti_tca9538
- dtbinding_ti_tca9546a
- dtbinding_ti_tca9548a
- dtbinding_ti_tcan4x5x
- dtbinding_ti_tla2021
- dtbinding_ti_tlc59108
- dtbinding_ti_tlc5971
- dtbinding_ti_tlc59731
- dtbinding_ti_tlv320dac
- dtbinding_ti_tmag5170
- dtbinding_ti_tmag5273
- dtbinding_ti_tmp007
- dtbinding_ti_tmp108
- dtbinding_ti_tmp112
- dtbinding_ti_tmp114
- dtbinding_ti_tmp116
- dtbinding_ti_tmp116_eeprom
- dtbinding_ti_tps382x
- dtbinding_ti_vim

u-blox (u-blox)

- dtbinding_u_blox_m10
- dtbinding_u_blox_sara_r4
- dtbinding_u_blox_sara_r5

UltraChip Inc. (ultrachip)

- dtbinding_ultrachip_uc8175
- dtbinding_ultrachip_uc8176
- dtbinding_ultrachip_uc8179

Vishay Intertechnology, Inc (vishay)

- dtbinding_vishay_vcnl36825t
- dtbinding_vishay_vcnl4040
- dtbinding_vishay_veml7700

Wistron NeWeb Corporation (wnc)

- dtbinding_wnc_m14a2a

WIZnet Co., Ltd. (wiznet)

- dtbinding_wiznet_w5500

Worldsemi Co., Limited (worldsemi)

- dtbinding_worldsemi_ws2812_gpio
- dtbinding_worldsemi_ws2812_i2s
- dtbinding_worldsemi_ws2812_rpi_pico_pio
- dtbinding_worldsemi_ws2812_spi

Würth Elektronik GmbH. (we)

- dtbinding_we_wsen_hids_spi
- dtbinding_we_wsen_hids_i2c
- dtbinding_we_wsen_itds
- dtbinding_we_wsen_pads_i2c
- dtbinding_we_wsen_pads_spi
- dtbinding_we_wsen_pdus
- dtbinding_we_wsen_tids

X-Powers (x-powers)

- dtbinding_x_powers_axp192
- dtbinding_x_powers_axp192_gpio
- dtbinding_x_powers_axp192_regulator

Xen Hypervisor (xen)

- dtbinding_xen_hvc_consoleio
- dtbinding_xen_hvc_uart
- dtbinding_xen_xen

Xilinx (xlnx)

- dtbinding_xlnx_fpga
- dtbinding_xlnx_gem
- dtbinding_xlnx_pinctrl_zynq
- dtbinding_xlnx_pinctrl_zynqmp
- dtbinding_xlnx_ps_gpio
- dtbinding_xlnx_ps_gpio_bank
- dtbinding_xlnx_ttcps
- dtbinding_xlnx_xps_gpio_1.00.a
- dtbinding_xlnx_xps_gpio_1.00.a_gpio2
- dtbinding_xlnx_xps_iic_2.00.a

- dtbinding_xlnx_xps_iic_2.1
- dtbinding_xlnx_xps_spi_2.00.a
- dtbinding_xlnx_xps_timebase_wdt_1.00.a
- dtbinding_xlnx_xps_timer_1.00.a
- dtbinding_xlnx_xps_timer_1.00.a_pwm
- dtbinding_xlnx_xps_uartlite_1.00.a
- dtbinding_xlnx_xuartps
- dtbinding_xlnx_zynq_ocm
- dtbinding_xlnx_zynqmp_ipi_mailbox

Zephyr-specific binding (zephyr)

- dtbinding_zephyr_adc_emul
- dtbinding_zephyr_bbram_emul
- dtbinding_zephyr_bt_hci_3wire_uart
- dtbinding_zephyr_bt_hci_entropy
- dtbinding_zephyr_bt_hci_ipc
- dtbinding_zephyr_bt_hci_ll_sw_split
- dtbinding_zephyr_bt_hci_spi
- dtbinding_zephyr_bt_hci_spi_slave
- dtbinding_zephyr_bt_hci_uart
- dtbinding_zephyr_bt_hci_userchan
- dtbinding_zephyr_can_loopback
- dtbinding_zephyr_cdc_acm_uart
- dtbinding_zephyr_cdc_ecm_ethernet
- dtbinding_zephyr_coredump
- dtbinding_zephyr_counter_watchdog
- dtbinding_zephyr_devmux
- dtbinding_zephyr_dma_emul
- dtbinding_zephyr_dummy_dc
- dtbinding_zephyr_emu_eeprom
- dtbinding_zephyr_espi_emul_controller
- dtbinding_zephyr_fake_can
- dtbinding_zephyr_fake_eeprom
- dtbinding_zephyr_fake_regulator
- dtbinding_zephyr_fake_rtc
- dtbinding_zephyr_flash_disk
- dtbinding_zephyr_fstab
- dtbinding_zephyr_fstab_littlefs
- dtbinding_zephyr_gnss_emul

- dtbinding_zephyr_gpio_emul
- dtbinding_zephyr_gpio_emul_sdl
- dtbinding_zephyr_hid_device
- dtbinding_zephyr_i2c_dump_allowlist
- dtbinding_zephyr_i2c_emul_controller
- dtbinding_zephyr_i2c_target_eeprom
- dtbinding_zephyr_ieee802154_uart_pipe
- dtbinding_zephyr_input_longpress
- dtbinding_zephyr_input_sdl_touch
- dtbinding_zephyr_ipc_icbmsg
- dtbinding_zephyr_ipc_icmsg
- dtbinding_zephyr_ipc_icmsg_me_follower
- dtbinding_zephyr_ipc_icmsg_me_initiator
- dtbinding_zephyr_ipc_openamp_static_vrings
- dtbinding_zephyr_kscan_input
- dtbinding_zephyr_log_uart
- dtbinding_zephyr_lvgl_button_input
- dtbinding_zephyr_lvgl_encoder_input
- dtbinding_zephyr_lvgl_keypad_input
- dtbinding_zephyr_lvgl_pointer_input
- dtbinding_zephyr_mdio_gpio
- dtbinding_zephyr_memory_region
- dtbinding_zephyr_mipi_dbi_spi
- dtbinding_zephyr_mmc_disk
- dtbinding_zephyr_modbus_serial
- dtbinding_zephyr_mspi_emul_controller
- dtbinding_zephyr_mspi_emul_device
- dtbinding_zephyr_mspi_emul_flash
- dtbinding_zephyr_native_linux_can
- dtbinding_zephyr_native_linux_evdev
- dtbinding_zephyr_native_posix_counter
- dtbinding_zephyr_native_posix_cpu
- dtbinding_zephyr_native_posix_rng
- dtbinding_zephyr_native_posix_uart
- dtbinding_zephyr_native_posix_udc
- dtbinding_zephyr_native_tty_uart
- dtbinding_zephyr_nus_uart
- dtbinding_panel_timing
- dtbinding_zephyr_power_state

- dtbinding_zephyr_psa_crypto_rng
- dtbinding_zephyr_ram_disk
- dtbinding_zephyr_retained_ram
- dtbinding_zephyr_retained_reg
- dtbinding_zephyr_retention
- dtbinding_zephyr_rtc_emul
- dtbinding_zephyr_sdhc_spi_slot
- dtbinding_zephyr_sdl_dc
- dtbinding_zephyr_sdmmc_disk
- dtbinding_zephyr_sensing
- dtbinding_zephyr_sensing_hinge_angle
- dtbinding_zephyr_sensing_phy_3d_sensor
- dtbinding_zephyr_sim_eeprom
- dtbinding_zephyr_sim_flash
- dtbinding_zephyr_spi_bitbang
- dtbinding_zephyr_spi_emul_controller
- dtbinding_zephyr_swdp_gpio
- dtbinding_zephyr_uac2
- dtbinding_zephyr_uac2_audio_streaming
- dtbinding_zephyr_uac2_clock_source
- dtbinding_zephyr_uac2_input_terminal
- dtbinding_zephyr_uac2_output_terminal
- dtbinding_zephyr_uart_emul
- dtbinding_zephyr_udc_skeleton
- dtbinding_zephyr_udc_virtual
- dtbinding_zephyr_uhc_virtual
- dtbinding_zephyr_usb_c_vbus_adc
- dtbinding_zephyr_w1_gpio
- dtbinding_zephyr_w1_serial

Zhengzhou Winsen Electronics Technology Co., Ltd. (winsen)

- dtbinding_winsen_mhz19b

Unknown vendor

- dtbinding_swerv_pic

5.3 Configuration System (Kconfig)

The Zephyr kernel and subsystems can be configured at build time to adapt them for specific application and platform needs. Configuration is handled through Kconfig, which is the same configuration system used by the Linux kernel. The goal is to support configuration without having to change any source code.

Configuration options (often called *symbols*) are defined in `Kconfig` files, which also specify dependencies between symbols that determine what configurations are valid. Symbols can be grouped into menus and sub-menus to keep the interactive configuration interfaces organized.

The output from Kconfig is a header file `autoconf.h` with macros that can be tested at build time. Code for unused features can be compiled out to save space.

The following sections explain how to set Kconfig configuration options, go into detail on how Kconfig is used within the Zephyr project, and have some tips and best practices for writing `Kconfig` files.

5.3.1 Interactive Kconfig interfaces

There are two interactive configuration interfaces available for exploring the available Kconfig options and making temporary changes: `menuconfig` and `guiconfig`. `menuconfig` is a curses-based interface that runs in the terminal, while `guiconfig` is a graphical configuration interface.

Note

The configuration can also be changed by editing `zephyr/.config` in the application build directory by hand. Using one of the configuration interfaces is often handier, as they correctly handle dependencies between configuration symbols.

If you try to enable a symbol with unsatisfied dependencies in `zephyr/.config`, the assignment will be ignored and overwritten when re-configuring.

To make a setting permanent, you should set it in a `*.conf` file, as described in *Setting Kconfig configuration values*.

Tip

Saving a minimal configuration file (with e.g. `D` in menuconfig) and inspecting it can be handy when making settings permanent. The minimal configuration file only lists symbols that differ from their default value.

To run one of the configuration interfaces, do this:

1. Build your application as usual using either `west` or `cmake`:

   Using west:

   ```
   west build -b <board>
   ```

   Using CMake and ninja:

   ```
   mkdir build && cd build
   cmake -GNinja -DBOARD=<board> ..
   ninja
   ```

2. To run the terminal-based `menuconfig` interface, use either of these commands:

```
west build -t menuconfig
```

```
ninja menuconfig
```

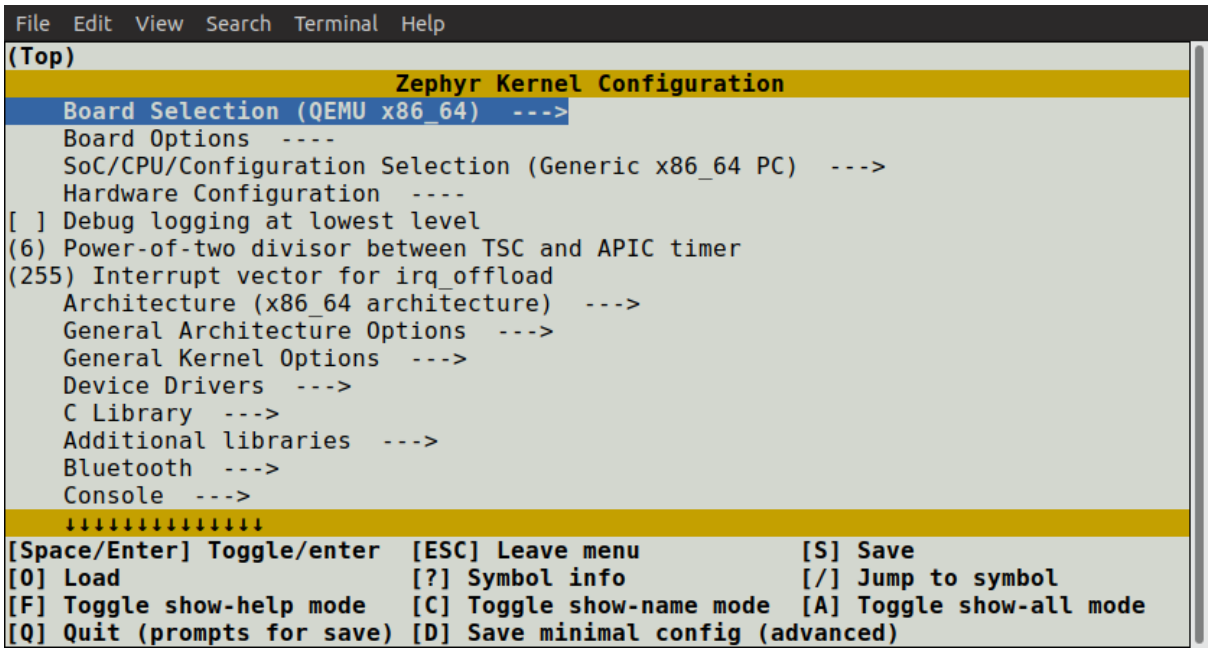
To run the graphical `guiconfig`, use either of these commands:

```
west build -t guiconfig
```

```
ninja guiconfig
```

> **Note**
>
> If you get an import error for `tkinter` when trying to run `guiconfig`, you are missing required packages. See *Install Linux Host Dependencies*. The package you need is usually called something like `python3-tk`/`python3-tkinter`.
>
> `tkinter` is not included by default in many Python installations, despite being part of the standard library.

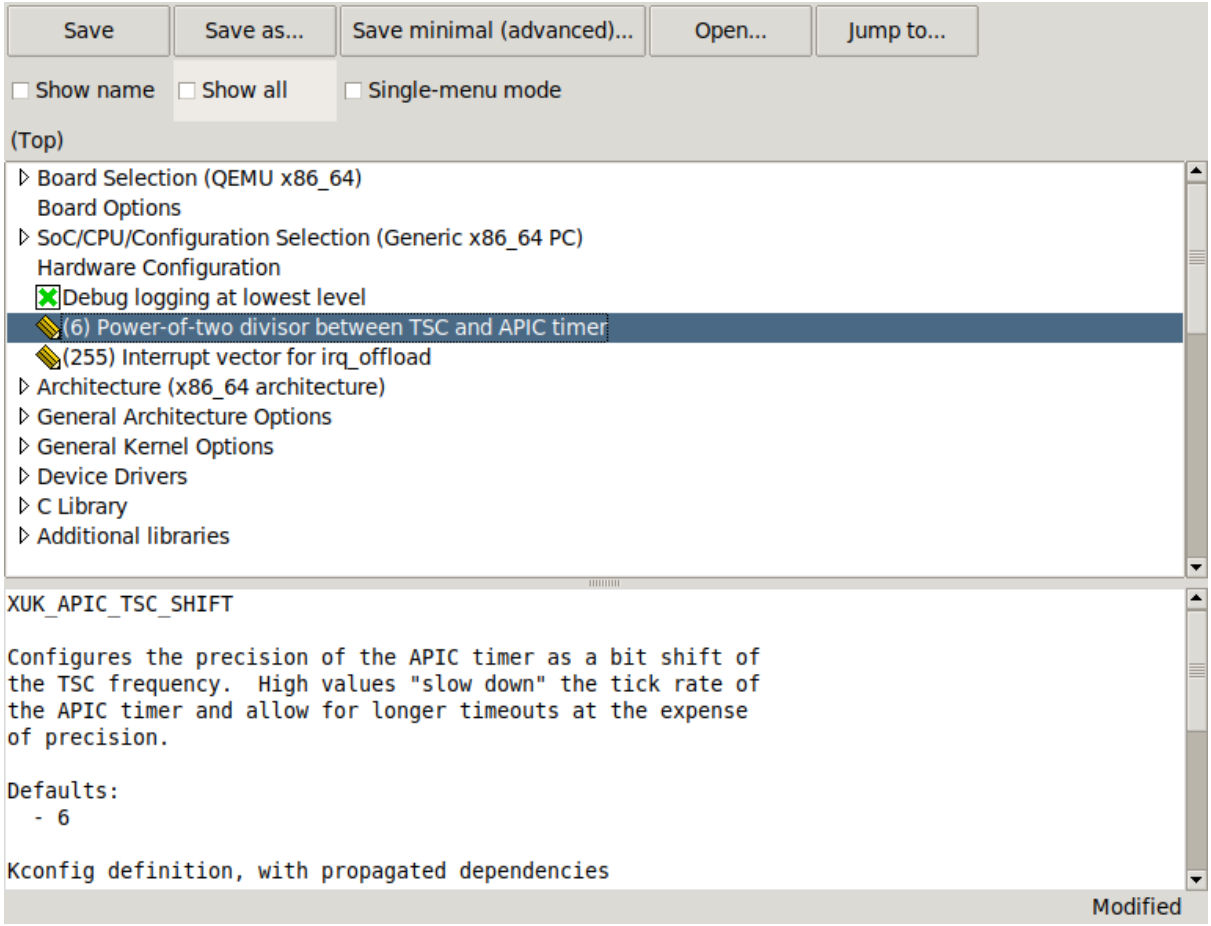
The two interfaces are shown below:



`guiconfig` always shows the help text and other information related to the currently selected item in the bottom window pane. In the terminal interface, press ? to view the same information.

> **Note**
>
> If you prefer to work in the `guiconfig` interface, then it's a good idea to check any changes to Kconfig files you make in *single-menu mode*, which is toggled via a checkbox at the top. Unlike full-tree mode, single-menu mode will distinguish between symbols defined with `config` and symbols defined with `menuconfig`, showing you what things would look like in the `menuconfig` interface.

3. Change configuration values in the `menuconfig` interface as follows:



- Navigate the menu with the arrow keys. Common Vim key bindings are supported as well.
- Use `Space` and `Enter` to enter menus and toggle values. Menus appear with `--->` next to them. Press `ESC` to return to the parent menu.

  Boolean configuration options are shown with *[ ]* brackets, while numeric and string-valued configuration symbols are shown with *( )* brackets. Symbol values that can't be changed are shown as - - or -*-.

  > **Note**
  >
  > You can also press `Y` or `N` to set a boolean configuration symbol to the corresponding value.

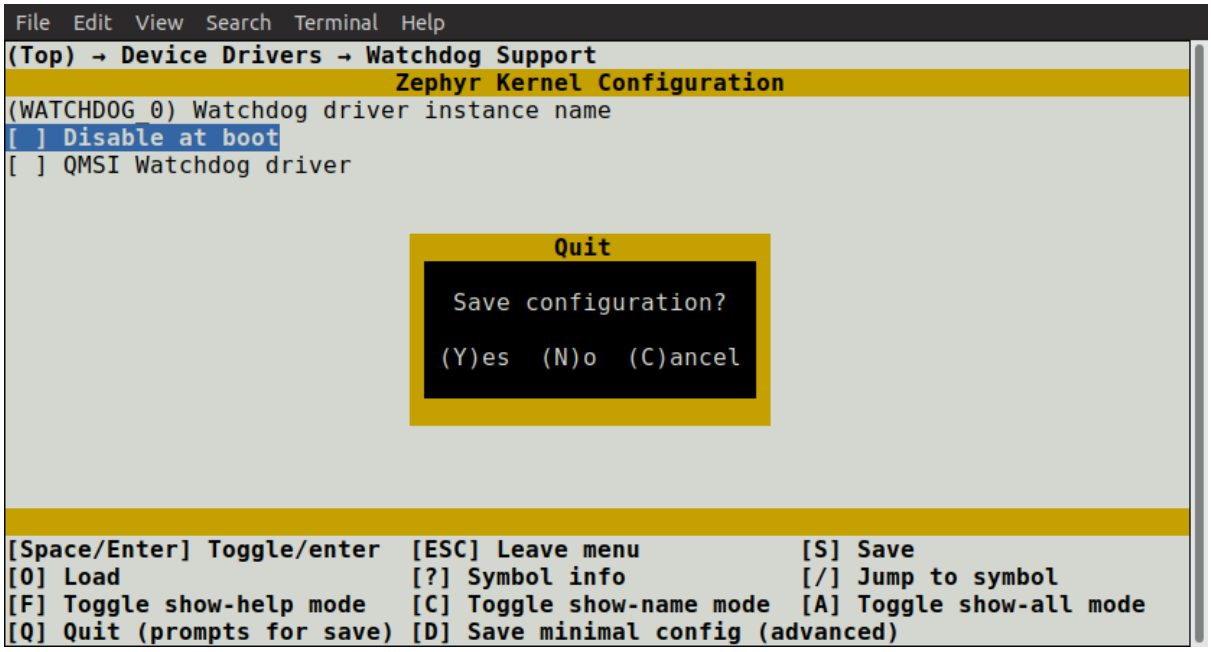
- Press `?` to display information about the currently selected symbol, including its help text. Press `ESC` or `Q` to return from the information display to the menu.

In the `guiconfig` interface, either click on the image next to the symbol to change its value, or double-click on the row with the symbol (this only works if the symbol has no children, as double-clicking a symbol with children open/closes its menu instead).

`guiconfig` also supports keyboard controls, which are similar to `menuconfig`.

4. Pressing `Q` in the `menuconfig` interface will bring up the save-and-quit dialog (if there are changes to save):

   Press `Y` to save the kernel configuration options to the default filename (`zephyr/.config`). You will typically save to the default filename unless you are experimenting with different configurations.



The `guiconfig` interface will also prompt for saving the configuration on exit if it has been modified.

> **Note**
>
> The configuration file used during the build is always `zephyr/.config`. If you have another saved configuration that you want to build with, copy it to `zephyr/.config`. Make sure to back up your original configuration file.
>
> Also note that filenames starting with `.` are not listed by `ls` by default on Linux and macOS. Use the `-a` flag to see them.

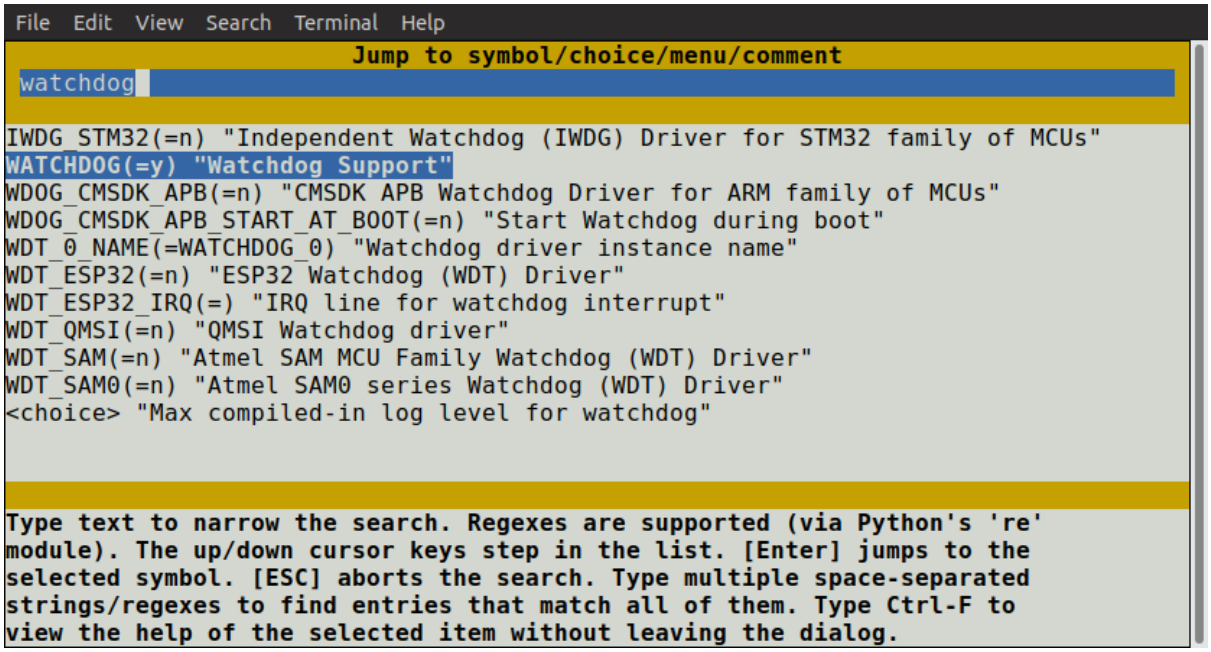
Finding a symbol in the menu tree and navigating to it can be tedious. To jump directly to a symbol, press the `/` key (this also works in `guiconfig`). This brings up the following dialog, where you can search for symbols by name and jump to them. In `guiconfig`, you can also change symbol values directly within the dialog.

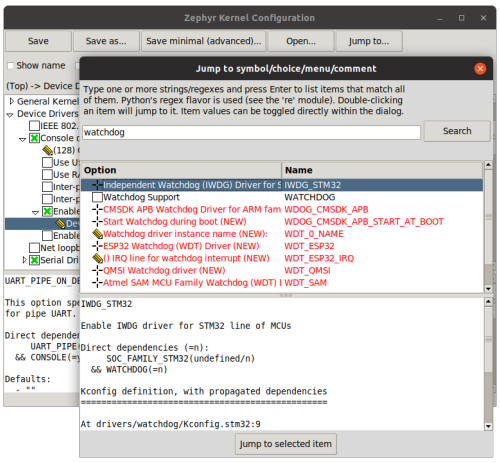
If you jump to a symbol that isn't currently visible (e.g., due to having unsatisfied dependencies), then *show-all mode* will be enabled. In show-all mode, all symbols are displayed, including currently invisible symbols. To turn off show-all mode, press `A` in `menuconfig` or `Ctrl-A` in `guiconfig`.

> **Note**
>
> Show-all mode can't be turned off if there are no visible items in the current menu.

To figure out why a symbol you jumped to isn't visible, inspect its dependencies, either by pressing `?` in `menuconfig` or in the information pane at the bottom in `guiconfig`. If you discover that the symbol depends on another symbol that isn't enabled, you can jump to that symbol in turn to see if it can be enabled.

> **Note**
>
> In `menuconfig`, you can press `Ctrl-F` to view the help of the currently selected item in the


File Edit View Search Terminal Help
Jump to symbol/choice/menu/comment
watchdog
IWDG_STM32(=n) "Independent Watchdog (IWDG) Driver for STM32 family of MCUs"
WATCHDOG(=y) "Watchdog Support"
WDOG_CMSDK_APB(=n) "CMSDK APB Watchdog Driver for ARM family of MCUs"
WDOG_CMSDK_APB_START_AT_BOOT(=n) "Start Watchdog during boot"
WDT_0_NAME(=WATCHDOG_0) "Watchdog driver instance name"
WDT_ESP32(=n) "ESP32 Watchdog (WDT) Driver"
WDT_ESP32_IRQ(=) "IRQ line for watchdog interrupt"
WDT_QMSI(=n) "QMSI Watchdog driver"
WDT_SAM(=n) "Atmel SAM MCU Family Watchdog (WDT) Driver"
WDT_SAM0(=n) "Atmel SAM0 series Watchdog (WDT) Driver"
<choice> "Max compiled-in log level for watchdog"
Type text to narrow the search. Regexes are supported (via Python's 're' module). The up/down cursor keys step in the list. [Enter] jumps to the selected symbol. [ESC] aborts the search. Type multiple space-separated strings/regexes to find entries that match all of them. Type Ctrl-F to view the help of the selected item without leaving the dialog.


Zephyr Kernel Configuration
Save
Save as...
Save minimal (advanced)...
Open...
Jump to...
Jump to symbol/choice/menu/comment
Type one or more strings/regexes and press Enter to list items that match all of them. Python's regex flavor is used (see the 're' module). Double-clicking an item will jump to it. Item values can be toggled directly within the dialog.
watchdog
Search
Option
Name
Independent Watchdog (IWDG) Driver for S
IWDG_STM32
Watchdog Support
WATCHDOG
CMSDK APB Watchdog Driver for ARM fam
WDOG_CMSDK_APB
Start Watchdog during boot (NEW)
WDOG_CMSDK_APB_START_AT_BOOT
Watchdog driver instance name (NEW):
WDT_0_NAME
ESP32 Watchdog (WDT) Driver (NEW)
WDT_ESP32
() IRQ line for watchdog interrupt (NEW)
WDT_ESP32_IRQ
QMSI Watchdog driver (NEW)
WDT_QMSI
Atmel SAM MCU Family Watchdog (WDT) (
WDT_SAM
IWDG_STM32
Enable IWDG driver for STM32 line of MCUs
Direct dependencies (=n):
SOC_FAMILY_STM32(undefined/n)
&& WATCHDOG(=n)
Kconfig definition, with propagated dependencies
At drivers/watchdog/Kconfig.stm32:9
Jump to selected item

jump-to dialog without leaving the dialog.

For more information on `menuconfig` and `guiconfig`, see the Python docstrings at the top of menuconfig.py and guiconfig.py.

5.3.2 Setting Kconfig configuration values

The *menuconfig and guiconfig interfaces* can be used to test out configurations during application development. This page explains how to make settings permanent.

All Kconfig options can be searched in the Kconfig search page.

Note

Before making changes to Kconfig files, it's a good idea to also go through the *Kconfig - Tips and Best Practices* page.

Visible and invisible Kconfig symbols

When making Kconfig changes, it's important to understand the difference between *visible* and *invisible* symbols.

- A visible symbol is a symbol defined with a prompt. Visible symbols show up in the interactive configuration interfaces (hence *visible*), and can be set in configuration files.

  Here's an example of a visible symbol:

  ```
  config FPU
     bool "Support floating point operations"
     depends on HAS_FPU
  ```

  The symbol is shown like this in `menuconfig`, where it can be toggled:

  ```
  [ ] Support floating point operations
  ```

- An *invisible* symbol is a symbol without a prompt. Invisible symbols are not shown in the interactive configuration interfaces, and users have no direct control over their value. They instead get their value from defaults or from other symbols.

  Here's an example of an invisible symbol:

  ```
  config CPU_HAS_FPU
     bool
     help
       This symbol is y if the CPU has a hardware floating point unit.
  ```

  In this case, `CPU_HAS_FPU` is enabled through other symbols having `select CPU_HAS_FPU`.

Setting symbols in configuration files

Visible symbols can be configured by setting them in configuration files. The initial configuration is produced by merging a `*_defconfig` file for the board with application settings, usually from `prj.conf`. See *The Initial Configuration* below for more details.

Assignments in configuration files use this syntax:

```
CONFIG_<symbol name>=<value>
```

There should be no spaces around the equals sign.

`bool` symbols can be enabled or disabled by setting them to `y` or `n`, respectively. The `FPU` symbol from the example above could be enabled like this:

```
CONFIG_FPU=y
```

Note

A boolean symbol can also be set to `n` with a comment formatted like this:

```
# CONFIG_SOME_OTHER_BOOL is not set
```

This is the format you will see in the merged configuration in `zephyr/.config`.

This style is accepted for historical reasons: Kconfig configuration files can be parsed as makefiles (though Zephyr doesn't use this). Having `n`-valued symbols correspond to unset variables simplifies tests in Make.

Other symbol types are assigned like this:

```
CONFIG_SOME_STRING="cool value"
CONFIG_SOME_INT=123
```

Comments use a #:

```
# This is a comment
```

Assignments in configuration files are only respected if the dependencies for the symbol are satisfied. A warning is printed otherwise. To figure out what the dependencies of a symbol are, use one of the *interactive configuration interfaces* (you can jump directly to a symbol with /), or look up the symbol in the Kconfig search page.

The Initial Configuration

The initial configuration for an application comes from merging configuration settings from three sources:

1. A `BOARD`-specific configuration file stored in `boards/<VENDOR>/<BOARD>/<BOARD>_defconfig`
2. Any CMake cache entries prefix with `CONFIG_`
3. The application configuration

The application configuration can come from the sources below (each file is known as a Kconfig fragment, which are then merged to get the final configuration used for a particular build). By default, `prj.conf` is used.

1. If `CONF_FILE` is set, the configuration file(s) specified in it are merged and used as the application configuration. `CONF_FILE` can be set in various ways:
   1. In `CMakeLists.txt`, before calling `find_package(Zephyr)`
   2. By passing `-DCONF_FILE=<conf file(s)>`, either directly or via `west`
   3. From the CMake variable cache

   Furthermore if `CONF_FILE` is set as single configuration file of the form `prj_<build>.conf` and if file `boards/<BOARD>_<build>.conf` exists in same folder as file `prj_<build>.conf`, the result of merging `prj_<build>.conf` and `boards/<BOARD>_<build>.conf` is used - note that this feature is deprecated, *File Suffixes* should be used instead.

2. Otherwise, if boards/<BOARD>.conf exists in the application configuration directory, the result of merging it with prj.conf is used.
3. Otherwise, if board revisions are used and boards/<BOARD>_<revision>.conf exists in the application configuration directory, the result of merging it with prj.conf and boards/<BOARD>.conf is used.
4. Otherwise, prj.conf is used from the application configuration directory. If it does not exist then a fatal error will be emitted.

Furthermore, applications can have SoC overlay configuration that is applied to it, the file socs/<SOC>_<BOARD_QUALIFIERS>.conf will be applied if it exists, after the main project configuration has been applied and before any board overlay configuration files have been applied.

All configuration files will be taken from the application's configuration directory except for files with an absolute path that are given with the CONF_FILE, EXTRA_CONF_FILE, DTC_OVERLAY_FILE, and EXTRA_DTC_OVERLAY_FILE arguments. For these, a file in a Zephyr module can be referred by escaping the Zephyr module dir variable like this \${ZEPHYR_<module>_MODULE_DIR}/<path-to>/<file> when setting any of said variables in the application's CMakeLists.txt.

See *Application Configuration Directory* on how the application configuration directory is defined.

If a symbol is assigned both in <BOARD>_defconfig and in the application configuration, the value set in the application configuration takes precedence.

The merged configuration is saved to zephyr/.config in the build directory.

As long as zephyr/.config exists and is up-to-date (is newer than any BOARD and application configuration files), it will be used in preference to producing a new merged configuration. zephyr/.config is also the configuration that gets modified when making changes in the *interactive configuration interfaces*.

Tracking Kconfig symbols

It is possible to create Kconfig symbols which takes the default value of another Kconfig symbol.

This is valuable when you want a symbol specific to an application or subsystem but do not want to rely directly on the common symbol. For example, you might want to decouple the settings so they can be independently configured, or to ensure you always have a locally named setting, even if the external setting name changes. is later changed.

For example, consider the common FOO_STRING setting where a subsystem wants to have a SUB_FOO_STRING but still allow for customization.

This can be done like this:

```
config FOO_STRING
        string "Foo"
        default "foo"

config SUB_FOO_STRING
        string "Sub-foo"
        default FOO_STRING
```

This ensures that the default value of SUB_FOO_STRING is identical to FOO_STRING while still allows users to configure both settings independently.

It is also possible to make SUB_FOO_STRING invisible and thereby keep the two symbols in sync, unless the value of the tracking symbol is changed in a defconfig file.

```
config FOO_STRING
        string "Foo"
```

(continues on next page)

(continued from previous page)

```
        default "foo"

config SUB_FOO_STRING
        string
        default FOO_STRING
        help
          Hidden symbol which follows FOO_STRING
          Can be changed through *.defconfig files.
```

Configuring invisible Kconfig symbols

When making changes to the default configuration for a board, you might have to configure invisible symbols. This is done in `boards/<VENDOR>/<BOARD>/Kconfig.defconfig`, which is a regular `Kconfig` file.

Note

Assignments in `.config` files have no effect on invisible symbols, so this scheme is not just an organizational issue.

Assigning values in `Kconfig.defconfig` relies on defining a Kconfig symbol in multiple locations. As an example, say we want to set `FOO_WIDTH` below to 32:

```
config FOO_WIDTH
    int
```

To do this, we extend the definition of `FOO_WIDTH` as follows, in `Kconfig.defconfig`:

```
if BOARD_MY_BOARD

config FOO_WIDTH
    default 32

endif
```

Note

Since the type of the symbol (`int`) has already been given at the first definition location, it does not need to be repeated here. Only giving the type once at the "base" definition of the symbol is a good idea for reasons explained in *Common Kconfig shorthands*.

`default` values in `Kconfig.defconfig` files have priority over `default` values given on the "base" definition of a symbol. Internally, this is implemented by including the `Kconfig.defconfig` files first. Kconfig uses the first `default` with a satisfied condition, where an empty condition corresponds to `if y` (is always satisfied).

Note that conditions from surrounding top-level `if`s are propagated to symbol properties, so the above `default` is equivalent to `default 32 if BOARD_MY_BOARD`.

Multiple symbol definitions When a symbol is defined in multiple locations, each definition acts as an independent symbol that happens to share the same name. This means that properties are not appended to previous definitions. If the conditions for **ANY** definition result in the symbol resolving to `y`, the symbol will be `y`. It is therefore not possible to make the dependencies of a symbol more restrictive by defining it in multiple locations.

For example, the dependencies of the symbol FOO below are satisfied if either DEP1 **OR** DEP2 are true, it does not require both:

```
config FOO
   ...
   depends on DEP1

config FOO
   ...
   depends on DEP2
```

Warning

Symbols without explicit dependencies still follow the above rule. A symbol without any dependencies will result in the symbol always being assignable. The definition below will result in FOO always being enabled by default, regardless of the value of DEP1.

```
config FOO
   bool "FOO"
   depends on DEP1

config FOO
   default y
```

This dependency weakening can be avoided with the *configdefault* extension if the desire is only to add a new default without modifying any other behaviour of the symbol.

Note

When making changes to Kconfig.defconfig files, always check the symbol's direct dependencies in one of the *interactive configuration interfaces* afterwards. It is often necessary to repeat dependencies from the base definition of the symbol to avoid weakening a symbol's dependencies.

Motivation for Kconfig.defconfig files One motivation for this configuration scheme is to avoid making fixed BOARD-specific settings configurable in the interactive configuration interfaces. If all board configuration were done via <BOARD>_defconfig, all symbols would have to be visible, as values given in <BOARD>_defconfig have no effect on invisible symbols.

Having fixed settings be user-configurable would clutter up the configuration interfaces and make them harder to understand, and would make it easier to accidentally create broken configurations.

When dealing with fixed board-specific settings, also consider whether they should be handled via *devicetree* instead.

Configuring choices There are two ways to configure a Kconfig choice:

1. By setting one of the choice symbols to y in a configuration file.

   Setting one choice symbol to y automatically gives all other choice symbols the value n.

   If multiple choice symbols are set to y, only the last one set to y will be honored (the rest will get the value n). This allows a choice selection from a board defconfig file to be overridden from an application prj.conf file.

2. By changing the default of the choice in Kconfig.defconfig.

As with symbols, changing the default for a choice is done by defining the choice in multiple locations. For this to work, the choice must have a name.

As an example, assume that a choice has the following base definition (here, the name of the choice is FOO):

```
choice FOO
    bool "Foo choice"
    default B

config A
    bool "A"

config B
    bool "B"

endchoice
```

To change the default symbol of FOO to A, you would add the following definition to `Kconfig.defconfig`:

```
choice FOO
    default A
endchoice
```

The `Kconfig.defconfig` method should be used when the dependencies of the choice might not be satisfied. In that case, you're setting the default selection whenever the user makes the choice visible.

More Kconfig resources The *Kconfig - Tips and Best Practices* page has some tips for writing Kconfig files.

The kconfiglib.py docstring (at the top of the file) goes over how symbol values are calculated in detail.

5.3.3 Kconfig - Tips and Best Practices

This page covers some Kconfig best practices and explains some Kconfig behaviors and features that might be cryptic or that are easily overlooked.

Note

The official Kconfig documentation is kconfig-language.rst and kconfig-macro-language.rst.

- *What to turn into Kconfig options*
- *What not to turn into Kconfig options*
  - *Options that specify a device in the system by name*
  - *Options that specify fixed hardware configuration*
- *`select` statements*
  - *`select` pitfalls*
  - *Alternatives to `select`*
  - *Using `select` for helper symbols*

  - *select recommendations*
- *(Lack of) conditional includes*
- *"Stuck" symbols in menuconfig and guiconfig*
- *Assignments to promptless symbols in configuration files*
- *depends on and string/int/hex symbols*
- *menuconfig symbols*
- *Commas in macro arguments*
- *Checking changes in menuconfig/guiconfig*
- *Checking changes with scripts/kconfig/lint.py*
- *Style recommendations and shorthands*
  - *Factoring out common dependencies*
  - *Redundant defaults*
  - *Common Kconfig shorthands*
  - *Prompt strings*
  - *Header comments and other nits*
- *Lesser-known/used Kconfig features*
  - *The imply statement*
  - *Optional prompts*
  - *Optional choices*
  - *visible if conditions*
- *Other resources*

What to turn into Kconfig options

When deciding whether something belongs in Kconfig, it helps to distinguish between symbols that have prompts and symbols that don't.

If a symbol has a prompt (e.g. `bool "Enable foo"`), then the user can change the symbol's value in the `menuconfig` or `guiconfig` interface (see *Interactive Kconfig interfaces*), or by manually editing configuration files. Conversely, a symbol without a prompt can never be changed directly by the user, not even by manually editing configuration files.

Only put a prompt on a symbol if it makes sense for the user to change its value.

Symbols without prompts are called *hidden* or *invisible* symbols, because they don't show up in `menuconfig` and `guiconfig`. Symbols that have prompts can also be invisible, when their dependencies are not satisfied.

Symbols without prompts can't be configured directly by the user (they derive their value from other symbols), so less restrictions apply to them. If some derived setting is easier to calculate in Kconfig than e.g. during the build, then do it in Kconfig, but keep the distinction between symbols with and without prompts in mind.

See the *optional prompts* section for a way to deal with settings that are fixed on some machines and configurable on other machines.

What not to turn into Kconfig options

In Zephyr, Kconfig configuration is done after selecting a target board. In general, it does not make sense to use Kconfig for a value that corresponds to a fixed machine-specific setting. Usually, such settings should be handled via *devicetree* instead.

In particular, avoid adding new Kconfig options of the following types:

Options that specify a device in the system by name For example, if you are writing an I2C device driver, avoid creating an option named `MY_DEVICE_I2C_BUS_NAME` for specifying the bus node your device is controlled by. See *Device drivers that depend on other devices* for alternatives.

Similarly, if your application depends on a hardware-specific PWM device to control an RGB LED, avoid creating an option like `MY_PWM_DEVICE_NAME`. See *Applications that depend on board-specific devices* for alternatives.

Options that specify fixed hardware configuration For example, avoid Kconfig options specifying a GPIO pin.

An alternative applicable to device drivers is to define a GPIO specifier with type phandle-array in the device binding, and using the *GPIO* devicetree API from C. Similar advice applies to other cases where devicetree.h provides *Hardware specific APIs* for referring to other nodes in the system. Search the source code for drivers using these APIs for examples.

An application-specific devicetree *binding* to identify board specific properties may be appropriate. See tests/drivers/gpio/gpio_basic_api for an example.

For applications, see blinky for a devicetree-based alternative.

`select` statements

The `select` statement is used to force one symbol to `y` whenever another symbol is `y`. For example, the following code forces `CONSOLE` to `y` whenever `USB_CONSOLE` is `y`:

```
config CONSOLE
     bool "Console support"

...

config USB_CONSOLE
     bool "USB console support"
     select CONSOLE
```

This section covers some pitfalls and good uses for `select`.

`select` pitfalls `select` might seem like a generally useful feature at first, but can cause configuration issues if overused.

For example, say that a new dependency is added to the `CONSOLE` symbol above, by a developer who is unaware of the `USB_CONSOLE` symbol (or simply forgot about it):

```
config CONSOLE
     bool "Console support"
     depends on STRING_ROUTINES
```

Enabling `USB_CONSOLE` now forces `CONSOLE` to `y`, even if `STRING_ROUTINES` is `n`.

To fix the problem, the `STRING_ROUTINES` dependency needs to be added to `USB_CONSOLE` as well:

```
config USB_CONSOLE
     bool "USB console support"
     select CONSOLE
     depends on STRING_ROUTINES

...

config STRING_ROUTINES
     bool "Include string routines"
```

More insidious cases with dependencies inherited from `if` and `menu` statements are common.

An alternative attempt to solve the issue might be to turn the `depends on` into another `select`:

```
config CONSOLE
     bool "Console support"
     select STRING_ROUTINES

...

config USB_CONSOLE
     bool "USB console support"
     select CONSOLE
```

In practice, this often amplifies the problem, because any dependencies added to `STRING_ROUTINES` now need to be copied to both `CONSOLE` and `USB_CONSOLE`.

In general, whenever the dependencies of a symbol are updated, the dependencies of all symbols that (directly or indirectly) select it have to be updated as well. This is very often overlooked in practice, even for the simplest case above.

Chains of symbols selecting each other should be avoided in particular, except for simple helper symbols, as covered below in *Using select for helper symbols*.

Liberal use of `select` also tends to make Kconfig files harder to read, both due to the extra dependencies and due to the non-local nature of `select`, which hides ways in which a symbol might get enabled.

Alternatives to `select` For the example in the previous section, a better solution is usually to turn the `select` into a `depends on`:

```
config CONSOLE
     bool "Console support"

...

config USB_CONSOLE
     bool "USB console support"
     depends on CONSOLE
```

This makes it impossible to generate an invalid configuration, and means that dependencies only ever have to be updated in a single spot.

An objection to using `depends on` here might be that configuration files that enable `USB_CONSOLE` now also need to enable `CONSOLE`:

```
CONFIG_CONSOLE=y
CONFIG_USB_CONSOLE=y
```

This comes down to a trade-off, but if enabling `CONSOLE` is the norm, then a mitigation is to make `CONSOLE` default to `y`:

```
config CONSOLE
     bool "Console support"
     default y
```

This gives just a single assignment in configuration files:

```
CONFIG_USB_CONSOLE=y
```

Note that configuration files that do not want `CONSOLE` enabled now have to explicitly disable it:

```
CONFIG_CONSOLE=n
```

Using `select` for helper symbols A good and safe use of `select` is for setting "helper" symbols that capture some condition. Such helper symbols should preferably have no prompt or dependencies.

For example, a helper symbol for indicating that a particular CPU/SoC has an FPU could be defined as follows:

```
config CPU_HAS_FPU
     bool
     help
       If y, the CPU has an FPU

...

config SOC_FOO
     bool "FOO SoC"
     select CPU_HAS_FPU

...

config SOC_BAR
     bool "BAR SoC"
     select CPU_HAS_FPU
```

This makes it possible for other symbols to check for FPU support in a generic way, without having to look for particular architectures:

```
config FPU
     bool "Support floating point operations"
     depends on CPU_HAS_FPU
```

The alternative would be to have dependencies like the following, possibly duplicated in several spots:

```
config FPU
     bool "Support floating point operations"
     depends on SOC_FOO || SOC_BAR || ...
```

Invisible helper symbols can also be useful without `select`. For example, the following code defines a helper symbol that has the value `y` if the machine has some arbitrarily-defined "large" amount of memory:

```
config LARGE_MEM
     def_bool MEM_SIZE >= 64
```

Note

This is short for the following:

```
config LARGE_MEM
  bool
  default MEM_SIZE >= 64
```

`select` recommendations In summary, here are some recommended practices for `select`:

- Avoid selecting symbols with prompts or dependencies. Prefer `depends on`. If `depends on` causes annoying bloat in configuration files, consider adding a Kconfig default for the most common value.

  Rare exceptions might include cases where you're sure that the dependencies of the selecting and selected symbol will never drift out of sync, e.g. when dealing with two simple symbols defined close to one another within the same `if`.

  Common sense applies, but be aware that `select` often causes issues in practice. `depends on` is usually a cleaner and safer solution.

- Select simple helper symbols without prompts and dependencies however much you like. They're a great tool for simplifying Kconfig files.
- An exemption are buses like I2C and SPI, and following the same thought process things like MFD as well. Drivers on these buses should use `select` to allow the automatic activation of the necessary bus drivers when devices on the bus are enabled in the devicetree.

```
config ADC_FOO
   bool "external SPI ADC foo driver"
   select SPI
```

(Lack of) conditional includes

`if` blocks add dependencies to each item within the `if`, as if `depends on` was used.

A common misunderstanding related to `if` is to think that the following code conditionally includes the file `Kconfig.other`:

```
if DEP
source "Kconfig.other"
endif
```

In reality, there are no conditional includes in Kconfig. `if` has no special meaning around a `source`.

> **Note**
>
> Conditional includes would be impossible to implement, because `if` conditions may contain (either directly or indirectly) forward references to symbols that haven't been defined yet.

Say that `Kconfig.other` above contains this definition:

```
config FOO
     bool "Support foo"
```

In this case, `FOO` will end up with this definition:

```
config FOO
     bool "Support foo"
     depends on DEP
```

Note that it is redundant to add `depends on DEP` to the definition of `FOO` in `Kconfig.other`, because the `DEP` dependency has already been added by `if DEP`.

In general, try to avoid adding redundant dependencies. They can make the structure of the Kconfig files harder to understand, and also make changes more error-prone, since it can be hard to spot that the same dependency is added twice.

"Stuck" symbols in menuconfig and guiconfig

There is a common subtle gotcha related to interdependent configuration symbols with prompts. Consider these symbols:

```
config FOO
     bool "Foo"

config STACK_SIZE
     hex "Stack size"
     default 0x200 if FOO
     default 0x100
```

Assume that the intention here is to use a larger stack whenever `FOO` is enabled, and that the configuration initially has `FOO` disabled. Also, remember that Zephyr creates an initial configuration in `zephyr/.config` in the build directory by merging configuration files (including e.g. `prj.conf`). This configuration file exists before `menuconfig` or `guiconfig` is run.

When first entering the configuration interface, the value of `STACK_SIZE` is 0x100, as expected. After enabling `FOO`, you might reasonably expect the value of `STACK_SIZE` to change to 0x200, but it stays as 0x100.

To understand what's going on, remember that `STACK_SIZE` has a prompt, meaning it is user-configurable, and consider that all Kconfig has to go on from the initial configuration is this:

```
CONFIG_STACK_SIZE=0x100
```

Since Kconfig can't know if the 0x100 value came from a `default` or was typed in by the user, it has to assume that it came from the user. Since `STACK_SIZE` is user-configurable, the value from the configuration file is respected, and any symbol defaults are ignored. This is why the value of `STACK_SIZE` appears to be "frozen" at 0x100 when toggling `FOO`.

The right fix depends on what the intention is. Here's some different scenarios with suggestions:

- If `STACK_SIZE` can always be derived automatically and does not need to be user-configurable, then just remove the prompt:

  ```
  config STACK_SIZE
     hex
     default 0x200 if FOO
     default 0x100
  ```

  Symbols without prompts ignore any value from the saved configuration.

- If `STACK_SIZE` should usually be user-configurable, but needs to be set to 0x200 when `FOO` is enabled, then disable its prompt when `FOO` is enabled, as described in *optional prompts*:

  ```
  config STACK_SIZE
     hex "Stack size" if !FOO
     default 0x200 if FOO
     default 0x100
  ```

- If `STACK_SIZE` should usually be derived automatically, but needs to be set to a custom value in rare circumstances, then add another option for making `STACK_SIZE` user-configurable:

```
config CUSTOM_STACK_SIZE
   bool "Use a custom stack size"
   help
     Enable this if you need to use a custom stack size. When disabled, a
     suitable stack size is calculated automatically.

config STACK_SIZE
   hex "Stack size" if CUSTOM_STACK_SIZE
   default 0x200 if FOO
   default 0x100
```

As long as `CUSTOM_STACK_SIZE` is disabled, `STACK_SIZE` will ignore the value from the saved configuration.

It is a good idea to try out changes in the `menuconfig` or `guiconfig` interface, to make sure that things behave the way you expect. This is especially true when making moderately complex changes like these.

Assignments to promptless symbols in configuration files

Assignments to hidden (promptless, also called *invisible*) symbols in configuration files are always ignored. Hidden symbols get their value indirectly from other symbols, via e.g. `default` and `select`.

A common source of confusion is opening the output configuration file (`zephyr/.config`), seeing a bunch of assignments to hidden symbols, and assuming that those assignments must be respected when the configuration is read back in by Kconfig. In reality, all assignments to hidden symbols in `zephyr/.config` are ignored by Kconfig, like for other configuration files.

To understand why `zephyr/.config` still includes assignments to hidden symbols, it helps to realize that `zephyr/.config` serves two separate purposes:

1. It holds the saved configuration, and
2. it holds configuration output. `zephyr/.config` is parsed by the CMake files to let them query configuration settings, for example.

The assignments to hidden symbols in `zephyr/.config` are just configuration output. Kconfig itself ignores assignments to hidden symbols when calculating symbol values.

Note

A *minimal configuration*, which can be generated from within the *menuconfig and guiconfig interfaces*, could be considered closer to just a saved configuration, without the full configuration output.

`depends on` and `string`/`int`/`hex` symbols

`depends on` works not just for `bool` symbols, but also for `string`, `int`, and `hex` symbols (and for choices).

The Kconfig definitions below will hide the `FOO_DEVICE_FREQUENCY` symbol and disable any configuration output for it when `FOO_DEVICE` is disabled.

```
config FOO_DEVICE
     bool "Foo device"

config FOO_DEVICE_FREQUENCY
```

(continues on next page)

(continued from previous page)

```
    int "Foo device frequency"
    depends on FOO_DEVICE
```

In general, it's a good idea to check that only relevant symbols are ever shown in the menuconfig/guiconfig interface. Having FOO_DEVICE_FREQUENCY show up when FOO_DEVICE is disabled (and possibly hidden) makes the relationship between the symbols harder to understand, even if code never looks at FOO_DEVICE_FREQUENCY when FOO_DEVICE is disabled.

menuconfig symbols

If the definition of a symbol FOO is immediately followed by other symbols that depend on FOO, then those symbols become children of FOO. If FOO is defined with config FOO, then the children are shown indented relative to FOO. Defining FOO with menuconfig FOO instead puts the children in a separate menu rooted at FOO.

menuconfig has no effect on evaluation. It's just a display option.

menuconfig can cut down on the number of menus and make the menu structure easier to navigate. For example, say you have the following definitions:

```
menu "Foo subsystem"

config FOO_SUBSYSTEM
    bool "Foo subsystem"

if FOO_SUBSYSTEM

config FOO_FEATURE_1
    bool "Foo feature 1"

config FOO_FEATURE_2
    bool "Foo feature 2"

config FOO_FREQUENCY
    int "Foo frequency"

... lots of other FOO-related symbols

endif # FOO_SUBSYSTEM

endmenu
```

In this case, it's probably better to get rid of the menu and turn FOO_SUBSYSTEM into a menuconfig symbol:

```
menuconfig FOO_SUBSYSTEM
    bool "Foo subsystem"

if FOO_SUBSYSTEM

config FOO_FEATURE_1
    bool "Foo feature 1"

config FOO_FEATURE_2
    bool "Foo feature 2"

config FOO_FREQUENCY
    int "Foo frequency"

... lots of other FOO-related symbols
```

(continues on next page)

(continued from previous page)

```
endif # FOO_SUBSYSTEM
```

In the `menuconfig` interface, this will be displayed as follows:

```
[*] Foo subsystem  --->
```

Note that making a symbol without children a `menuconfig` is meaningless. It should be avoided, because it looks identical to a symbol with all children invisible:

```
[*] I have no children  ----
[*] All my children are invisible  ----
```

Commas in macro arguments

Kconfig uses commas to separate macro arguments. This means a construct like this will fail:

```
config FOO
    bool
    default y if $(dt_chosen_enabled,"zephyr,bar")
```

To solve this problem, create a variable with the text and use this variable as argument, as follows:

```
DT_CHOSEN_ZEPHYR_BAR := zephyr,bar

config FOO
    bool
    default y if $(dt_chosen_enabled,$(DT_CHOSEN_ZEPHYR_BAR))
```

Checking changes in menuconfig/guiconfig

When adding new symbols or making other changes to Kconfig files, it is a good idea to look up the symbols in *menuconfig or guiconfig* afterwards. To get to a symbol quickly, use the jump-to feature (press /).

Here are some things to check:

- Are the symbols placed in a good spot? Check that they appear in a menu where they make sense, close to related symbols.

  If one symbol depends on another, then it's often a good idea to place it right after the symbol it depends on. It will then be shown indented relative to the symbol it depends on in the `menuconfig` interface, and in a separate menu rooted at the symbol in `guiconfig`. This also works if several symbols are placed after the symbol they depend on.

- Is it easy to guess what the symbols do from their prompts?
- If many symbols are added, do all combinations of values they can be set to make sense?

  For example, if two symbols `FOO_SUPPORT` and `NO_FOO_SUPPORT` are added, and both can be enabled at the same time, then that makes a nonsensical configuration. In this case, it's probably better to have a single `FOO_SUPPORT` symbol.

- Are there any duplicated dependencies?

  This can be checked by selecting a symbol and pressing ? to view the symbol information. If there are duplicated dependencies, then use the `Included via ...` path shown in the symbol information to figure out where they come from.

Checking changes with scripts/kconfig/lint.py

After you make Kconfig changes, you can use the scripts/kconfig/lint.py script to check for some potential issues, like unused symbols and symbols that are impossible to enable. Use `--help` to see available options.

Some checks are necessarily a bit heuristic, so a symbol being flagged by a check does not necessarily mean there's a problem. If a check returns a false positive e.g. due to token pasting in C (`CONFIG_FOO_##index##_BAR`), just ignore it.

When investigating an unknown symbol `FOO_BAR`, it is a good idea to run `git grep FOO_BAR` to look for references. It is also a good idea to search for some components of the symbol name with e.g. `git grep FOO` and `git grep BAR`, as it can help uncover token pasting.

Style recommendations and shorthands

This section gives some style recommendations and explains some common Kconfig shorthands.

Factoring out common dependencies If a sequence of symbols/choices share a common dependency, the dependency can be factored out with an `if`.

As an example, consider the following code:

```
config FOO
     bool "Foo"
     depends on DEP

config BAR
     bool "Bar"
     depends on DEP

choice
     prompt "Choice"
     depends on DEP

config BAZ
     bool "Baz"

config QAZ
     bool "Qaz"

endchoice
```

Here, the `DEP` dependency can be factored out like this:

```
if DEP

config FOO
     bool "Foo"

config BAR
     bool "Bar"

choice
     prompt "Choice"

config BAZ
     bool "Baz"

config QAZ
```

(continues on next page)

(continued from previous page)

```
    bool "Qaz"

endchoice

endif # DEP
```

Note

Internally, the second version of the code is transformed into the first.

If a sequence of symbols/choices with shared dependencies are all in the same menu, the dependency can be put on the menu itself:

```
menu "Foo features"
    depends on FOO_SUPPORT

config FOO_FEATURE_1
    bool "Foo feature 1"

config FOO_FEATURE_2
    bool "Foo feature 2"

endmenu
```

If `FOO_SUPPORT` is `n`, the entire menu disappears.

Redundant defaults `bool` symbols implicitly default to `n`, and `string` symbols implicitly default to the empty string. Therefore, `default n` and `default ""` are (almost) always redundant.

The recommended style in Zephyr is to skip redundant defaults for `bool` and `string` symbols. That also generates clearer documentation: (*Implicitly defaults to n* instead of *n if <dependencies, possibly inherited>*).

Defaults *should* always be given for `int` and `hex` symbols, however, as they implicitly default to the empty string. This is partly for compatibility with the C Kconfig tools, though an implicit 0 default might be less likely to be what was intended compared to other symbol types as well.

The one case where `default n`/`default ""` is not redundant is when defining a symbol in multiple locations and wanting to override e.g. a `default y` on a later definition. Note that a `default n` does not override a previously defined `default y`.

That is, FOO will be set to `n` in the example below. If the `default n` was omitted in the first definition, FOO would have been set to `y`.

```
config FOO
   bool "foo"
   default n

config FOO
   bool "foo"
   default y
```

In the following example FOO will get the value `y`.

```
config FOO
   bool "foo"
   default y
```

(continues on next page)

(continued from previous page)

```
config FOO
   bool "foo"
   default n
```

Common Kconfig shorthands Kconfig has two shorthands that deal with prompts and defaults.

- `<type> "prompt"` is a shorthand for giving a symbol/choice a type and a prompt at the same time. These two definitions are equal:

```
config FOO
   bool "foo"
```

```
config FOO
   bool
   prompt "foo"
```

  The first style, with the shorthand, is preferred in Zephyr.

- `def_<type> <value>` is a shorthand for giving a type and a value at the same time. These two definitions are equal:

```
config FOO
   def_bool BAR && BAZ
```

```
config FOO
   bool
   default BAR && BAZ
```

Using both the `<type> "prompt"` and the `def_<type> <value>` shorthand in the same definition is redundant, since it gives the type twice.

The `def_<type> <value>` shorthand is generally only useful for symbols without prompts, and somewhat obscure.

Note

For a symbol defined in multiple locations (e.g., in a `Kconfig.defconfig` file in Zephyr), it is best to only give the symbol type for the "base" definition of the symbol, and to use `default` (instead of `def_<type> value`) for the remaining definitions. That way, if the base definition of the symbol is removed, the symbol ends up without a type, which generates a warning that points to the other definitions. That makes the extra definitions easier to discover and remove.

Prompt strings For a Kconfig symbol that enables a driver/subsystem FOO, consider having just "Foo" as the prompt, instead of "Enable Foo support" or the like. It will usually be clear in the context of an option that can be toggled on/off, and makes things consistent.

Header comments and other nits A few formatting nits, to help keep things consistent:

- Use this format for any header comments at the top of `Kconfig` files:

```
# <Overview of symbols defined in the file, preferably in plain English>
(Blank line)
# Copyright (c) 2019 ...
```

(continues on next page)

(continued from previous page)

```
# SPDX-License-Identifier: <License>
(Blank line)
(Kconfig definitions)
```

- Format comments as `# Comment` rather than `#Comment`
- Put a blank line before/after each top-level `if` and `endif`
- Use a single tab for each indentation
- Indent help text with two extra spaces

Lesser-known/used Kconfig features

This section lists some more obscure Kconfig behaviors and features that might still come in handy.

The `imply` statement The `imply` statement is similar to `select`, but respects dependencies and doesn't force a value. For example, the following code could be used to enable USB keyboard support by default on the FOO SoC, while still allowing the user to turn it off:

```
config SOC_FOO
	bool "FOO SoC"
	imply USB_KEYBOARD

...

config USB_KEYBOARD
	bool "USB keyboard support"
```

`imply` acts like a suggestion, whereas `select` forces a value.

Optional prompts A condition can be put on a symbol's prompt to make it optionally configurable by the user. For example, a value `MASK` that's hardcoded to 0xFF on some boards and configurable on others could be expressed as follows:

```
config MASK
	hex "Bitmask" if HAS_CONFIGURABLE_MASK
	default 0xFF
```

Note

This is short for the following:

```
config MASK
  hex
  prompt "Bitmask" if HAS_CONFIGURABLE_MASK
  default 0xFF
```

The `HAS_CONFIGURABLE_MASK` helper symbol would get selected by boards to indicate that `MASK` is configurable. When `MASK` is configurable, it will also default to 0xFF.

Optional choices Defining a choice with the `optional` keyword allows the whole choice to be toggled off to select none of the symbols:

```
choice
     prompt "Use legacy protocol"
     optional

config LEGACY_PROTOCOL_1
     bool "Legacy protocol 1"

config LEGACY_PROTOCOL_2
     bool "Legacy protocol 2"

endchoice
```

In the `menuconfig` interface, this will be displayed e.g. as `[*] Use legacy protocol (Legacy protocol 1) --->`, where the choice can be toggled off to enable neither of the symbols.

`visible if` conditions Putting a `visible if` condition on a menu hides the menu and all the symbols within it, while still allowing symbol default values to kick in.

As a motivating example, consider the following code:

```
menu "Foo subsystem"
     depends on HAS_CONFIGURABLE_FOO

config FOO_SETTING_1
     int "Foo setting 1"
     default 1

config FOO_SETTING_2
     int "Foo setting 2"
     default 2

endmenu
```

When `HAS_CONFIGURABLE_FOO` is `n`, no configuration output is generated for `FOO_SETTING_1` and `FOO_SETTING_2`, as the code above is logically equivalent to the following code:

```
config FOO_SETTING_1
     int "Foo setting 1"
     default 1
     depends on HAS_CONFIGURABLE_FOO

config FOO_SETTING_2
     int "Foo setting 2"
     default 2
     depends on HAS_CONFIGURABLE_FOO
```

If we want the symbols to still get their default values even when `HAS_CONFIGURABLE_FOO` is `n`, but not be configurable by the user, then we can use `visible if` instead:

```
menu "Foo subsystem"
     visible if HAS_CONFIGURABLE_FOO

config FOO_SETTING_1
     int "Foo setting 1"
     default 1

config FOO_SETTING_2
     int "Foo setting 2"
     default 2

endmenu
```

This is logically equivalent to the following:

```
config FOO_SETTING_1
     int "Foo setting 1" if HAS_CONFIGURABLE_FOO
     default 1

config FOO_SETTING_2
     int "Foo setting 2" if HAS_CONFIGURABLE_FOO
     default 2
```

Note

See the *optional prompts* section for the meaning of the conditions on the prompts.

When `HAS_CONFIGURABLE_FOO` is `n`, we now get the following configuration output for the symbols, instead of no output:

```
...
CONFIG_FOO_SETTING_1=1
CONFIG_FOO_SETTING_2=2
...
```

Other resources

The *Intro to symbol values* section in the Kconfiglib docstring goes over how symbols values are calculated in more detail.

5.3.4 Custom Kconfig Preprocessor Functions

Kconfiglib supports custom Kconfig preprocessor functions written in Python. These functions are defined in scripts/kconfig/kconfigfunctions.py.

Note

The official Kconfig preprocessor documentation can be found here.

Most of the custom preprocessor functions are used to get devicetree information into Kconfig. For example, the default value of a Kconfig symbol can be fetched from a devicetree `reg` property.

Devicetree-related Functions

The functions listed below are used to get devicetree information into Kconfig. See the Python docstrings in scripts/kconfig/kconfigfunctions.py for detailed documentation.

The `*_int` version of each function returns the value as a decimal integer, while the `*_hex` version returns a hexadecimal value starting with `0x`.

```
$(dt_alias_enabled,<node alias>)
$(dt_chosen_bool_prop, <property in /chosen>, <prop>)
$(dt_chosen_enabled,<property in /chosen>)
$(dt_chosen_has_compat,<property in /chosen>)
$(dt_chosen_label,<property in /chosen>)
$(dt_chosen_partition,addr_hex,<chosen>[,<index>,<unit>])
```

(continues on next page)

(continued from previous page)

```
$(dt_chosen_partition,addr_int,<chosen>[,<index>,<unit>])
$(dt_chosen_path,<property in /chosen>)
$(dt_chosen_reg_addr_hex,<property in /chosen>[,<index>,<unit>])
$(dt_chosen_reg_addr_int,<property in /chosen>[,<index>,<unit>])
$(dt_chosen_reg_size_hex,<property in /chosen>[,<index>,<unit>])
$(dt_chosen_reg_size_int,<property in /chosen>[,<index>,<unit>])
$(dt_compat_any_has_prop,<compatible string>,<prop>)
$(dt_compat_any_on_bus,<compatible string>,<prop>)
$(dt_compat_enabled,<compatible string>)
$(dt_compat_on_bus,<compatible string>,<bus>)
$(dt_gpio_hogs_enabled)
$(dt_has_compat,<compatible string>)
$(dt_has_compat_enabled,<compatible string>)
$(dt_node_array_prop_hex,<node path>,<prop>,<index>[,<unit>])
$(dt_node_array_prop_int,<node path>,<prop>,<index>[,<unit>])
$(dt_node_bool_prop,<node path>,<prop>)
$(dt_node_has_compat,<node path>,<compatible string>)
$(dt_node_has_prop,<node path>,<prop>)
$(dt_node_int_prop_hex,<node path>,<prop>[,<unit>])
$(dt_node_int_prop_int,<node path>,<prop>[,<unit>])
$(dt_node_parent,<node path>)
$(dt_node_ph_array_prop_hex,<node path>,<prop>,<index>,<cell>[,<unit>])
$(dt_node_ph_array_prop_int,<node path>,<prop>,<index>,<cell>[,<unit>])
$(dt_node_ph_prop_path,<node path>,<prop>)
$(dt_node_reg_addr_hex,<node path>[,<index>,<unit>])
$(dt_node_reg_addr_int,<node path>[,<index>,<unit>])
$(dt_node_reg_size_hex,<node path>[,<index>,<unit>])
$(dt_node_reg_size_int,<node path>[,<index>,<unit>])
$(dt_node_str_prop_equals,<node path>,<prop>,<value>)
$(dt_nodelabel_array_prop_has_val, <node label>, <prop>, <value>)
$(dt_nodelabel_bool_prop,<node label>,<prop>)
$(dt_nodelabel_enabled,<node label>)
$(dt_nodelabel_enabled_with_compat,<node label>,<compatible string>)
$(dt_nodelabel_has_compat,<node label>,<compatible string>)
$(dt_nodelabel_has_prop,<node label>,<prop>)
$(dt_nodelabel_path,<node label>)
$(dt_nodelabel_reg_addr_hex,<node label>[,<index>,<unit>])
$(dt_nodelabel_reg_addr_int,<node label>[,<index>,<unit>])
$(dt_nodelabel_reg_size_hex,<node label>[,<index>,<unit>])
$(dt_nodelabel_reg_size_int,<node label>[,<index>,<unit>])
$(dt_path_enabled,<node path>)
$(normalize_upper,<string>)
$(shields_list_contains,<shield name>)
$(substring,<string>,<start>[,<stop>])
```

Example Usage Assume that the devicetree for some board looks like this:

```
{
     soc {
             #address-cells = <1>;
             #size-cells = <1>;

             spi0: spi@10014000 {
                     compatible = "sifive,spi0";
                     reg = <0x10014000 0x1000 0x20010000 0x3c0900>;
                     reg-names = "control", "mem";
                     ...
             };
};
```

The second entry in `reg` in `spi@1001400` (`<0x20010000 0x3c0900>`) corresponds to `mem`, and has the address `0x20010000`. This address can be inserted into Kconfig as follows:

```
config FLASH_BASE_ADDRESS
     default $(dt_node_reg_addr_hex,/soc/spi@1001400,1)
```

After preprocessor expansion, this turns into the definition below:

```
config FLASH_BASE_ADDRESS
     default 0x20010000
```

5.3.5 Kconfig extensions

Zephyr uses the Kconfiglib implementation of Kconfig, which includes some Kconfig extensions:

- Default values can be applied to existing symbols without *weakening* the symbols dependencies through the use of `configdefault`.

  ```
  config FOO
      bool "FOO"
      depends on BAR

  configdefault FOO
      default y if FIZZ
  ```

  The statement above is equivalent to:

  ```
  config FOO
      bool "Foo"
      default y if FIZZ
      depends on BAR
  ```

  `configdefault` symbols cannot contain any fields other than `default`, however they can be wrapped in `if` statements. The two statements below are equivalent:

  ```
  configdefault FOO
      default y if BAR

  if BAR
  configdefault FOO
      default y
  endif # BAR
  ```

- Environment variables in `source` statements are expanded directly, meaning no "bounce" symbols with `option env="ENV_VAR"` need to be defined.

  Note

  `option env` has been removed from the C tools as of Linux 4.18 as well.

  The recommended syntax for referencing environment variables is `$(FOO)` rather than `$FOO`. This uses the new Kconfig preprocessor. The `$FOO` syntax for expanding environment variables is only supported for backwards compatibility.

- The `source` statement supports glob patterns and includes each matching file. A pattern is required to match at least one file.

  Consider the following example:

```
source "foo/bar/*/Kconfig"
```

If the pattern `foo/bar/*/Kconfig` matches the files `foo/bar/baz/Kconfig` and `foo/bar/qaz/Kconfig`, the statement above is equivalent to the following two `source` statements:

```
source "foo/bar/baz/Kconfig"
source "foo/bar/qaz/Kconfig"
```

If no files match the pattern, an error is generated.

The wildcard patterns accepted are the same as for the Python glob module.

For cases where it's okay for a pattern to match no files (or for a plain filename to not exist), a separate `osource` (*optional source*) statement is available. `osource` is a no-op if no file matches.

Note

`source` and `osource` are analogous to `include` and `-include` in Make.

- An `rsource` statement is available for including files specified with a relative path. The path is relative to the directory of the `Kconfig` file that contains the `rsource` statement.

  As an example, assume that `foo/Kconfig` is the top-level `Kconfig` file, and that `foo/bar/Kconfig` has the following statements:

  ```
  source "qaz/Kconfig1"
  rsource "qaz/Kconfig2"
  ```

  This will include the two files `foo/qaz/Kconfig1` and `foo/bar/qaz/Kconfig2`.

  `rsource` can be used to create `Kconfig` "subtrees" that can be moved around freely.

  `rsource` also supports glob patterns.

  A drawback of `rsource` is that it can make it harder to figure out where a file gets included, so only use it if you need it.

- An `orsource` statement is available that combines `osource` and `rsource`.

  For example, the following statement will include `Kconfig1` and `Kconfig2` from the current directory (if they exist):

  ```
  orsource "Kconfig[12]"
  ```

- `def_int`, `def_hex`, and `def_string` keywords are available, analogous to `def_bool`. These set the type and add a `default` at the same time.

Users interested in optimizing their configuration for security should refer to the Zephyr Security Guide's section on the *Hardening Tool*.

5.4 Snippets

Snippets are a way to save build system settings in one place, and then use those settings when you build any Zephyr application. This lets you save common configuration separately when it applies to multiple different applications.

Some example use cases for snippets are:

- changing your board's console backend from a "real" UART to a USB CDC-ACM UART
- enabling frequently-used debugging options

- applying interrelated configuration settings to your “main” CPU and a co-processor core on an AMP SoC

The following pages document this feature.

5.4.1 Using Snippets

> **Tip**
>
> See *Built-in snippets* for a list of snippets that are provided by Zephyr.

Snippets have names. You use snippets by giving their names to the build system.

With west build

To use a snippet named `foo` when building an application `app`:

```
west build -S foo app
```

To use multiple snippets:

```
west build -S snippet1 -S snippet2 [...] app
```

With cmake

If you are running CMake directly instead of using `west build`, use the `SNIPPET` variable. This is a whitespace- or semicolon-separated list of snippet names you want to use. For example:

```
cmake -Sapp -Bbuild -DSNIPPET="snippet1;snippet2" [...]
cmake --build build
```

Application required snippets

If an application should always be compiled with a given snippet, it can be added to that application’s `CMakeLists.txt` file. For example:

```
if(NOT snippet1 IN_LIST SNIPPET)
  set(SNIPPET snippet1 ${SNIPPET} CACHE STRING "" FORCE)
endif()

find_package(Zephyr ....)
```

5.4.2 Built-in snippets

CDC-ACM Console Snippet (cdc-acm-console)

```
west build -S cdc-acm-console [...]
```

Overview This snippet redirects serial console output to a CDC ACM UART. The USB device which should be used is configured using *Devicetree*.

Requirements Hardware support for:

- CONFIG_USB_DEVICE_STACK
- CONFIG_SERIAL
- CONFIG_CONSOLE
- CONFIG_UART_CONSOLE
- CONFIG_UART_LINE_CTRL

A devicetree node with node label zephyr_udc0 that points to an enabled USB device node with driver support. This should look roughly like this in *your devicetree*:

```
zephyr_udc0: usbd@deadbeef {
     compatible = "vnd,usb-device";
     /* ... */
};
```

Nordic FLPR snippet with execution in place (nordic-flpr-xip)

Overview This snippet allows users to build Zephyr with the capability to boot Nordic FLPR (Fast Lightweight Peripheral Processor) from application core. FLPR code is to be executed from RRAM, so the FLPR image must be built for the xip board variant, or with CONFIG_XIP enabled.

Nordic FLPR snippet with execution from SRAM (nordic-flpr)

Overview This snippet allows users to build Zephyr with the capability to boot Nordic FLPR (Fast Lightweight Peripheral Processor) from application core. FLPR code is to be executed from SRAM, so the FLPR image must be built without the xip board variant, or with CONFIG_XIP disabled.

Nordic boot PPR snippet with execution in place (nordic-ppr-xip)

Overview This snippet allows users to build Zephyr with the capability to boot Nordic PPR (Peripheral Processor) from another core. PPR code is to be executed from MRAM, so the PPR image must be built for the xip board variant, or with CONFIG_XIP enabled.

Nordic PPR snippet (nordic-ppr)

Overview This snippet allows users to build Zephyr with the capability to boot Nordic PPR (Peripheral Processor) from another core.

NUS Console Snippet (nus-console)

```
west build -S nus-console [...]
```

Overview This snippet redirects serial console output to a UART over NUS (Bluetooth LE) instance. The Bluetooth Serial device used shall be configured using *Devicetree*.

Requirements Hardware support for:

- CONFIG_BT
- CONFIG_BT_PERIPHERAL
- CONFIG_BT_ZEPHYR_NUS
- CONFIG_SERIAL
- CONFIG_CONSOLE
- CONFIG_UART_CONSOLE

RAM Console Snippet (ram-console)

```
west build -S ram-console [...]
```

Overview This snippet redirects console output to a RAM buffer. The RAM console buffer is a global array located in RAM region by default, whose address is unknown before building. The RAM console driver also supports using a dedicated section for the RAM console buffer with prefined address.

How to enable RAM console buffer section Add board dts overlay to this snippet to add property `zephyr,ram-console` in the chosen node and memory-region node with compatible string `zephyr,memory-region` as the following:

```
/ {
    chosen {
        zephyr,ram-console = &snippet_ram_console;
    };

   snippet_ram_console: memory@93d00000 {
        compatible = "zephyr,memory-region";
        reg = <0x93d00000 DT_SIZE_K(4)>;
        zephyr,memory-region = "RAM_CONSOLE";
   };
};
```

RTT Console Snippet (rtt-console)

```
west build -S rtt-console [...]
```

Overview This snippet redirects serial console output to SEGGER RTT.

Requirements Hardware support for:

- CONFIG_HAS_SEGGER_RTT
- CONFIG_CONSOLE

Xen Dom0: universal snippet for XEN control domain

Overview This snippet allows user to build Zephyr as a Xen initial domain (Dom0). The feature is implemented as configuration snippet to allow support for any compatible platform.

How to add support of a new board

- add board dts overlay to this snippet which deletes/adds memory and deletes UART nodes;
- add correct memory and hypervisor nodes, based on regions Xen picked for Domain-0 on your setup.

Programming Correct snippet designation for Xen must be entered when you invoke `west build`. For example:

```
west build -b qemu_cortex_a53 -S xen_dom0 samples/synchronization
```

QEMU example with Xen Overlay for qemu_cortex_a53 board, that is present in *board/* directory of this snippet is QEMU Xen control domain example. To run such setup, you need to:

- fetch and build Xen (e.g. RELEASE-4.17.0) for arm64 platform
- take and compile sample device tree from *example/* directory
- build your Zephyr sample/application with *xen_dom0* snippet and start it as Xen control domain

For starting you can use QEMU from Zephyr SDK by following command:

```
<path to Zephyr SDK>/sysroots/x86_64-pokysdk-linux/usr/bin/qemu-system-aarch64 -cpu cortex-
↪a53 \
-m 6G -nographic -machine virt,gic-version=3,virtualization=true -chardev stdio,id=con,
↪mux=on \
-serial chardev:con -mon chardev=con,mode=readline -pidfile qemu.pid \
-device loader,file=<path to Zephyr app build>/zephyr.bin,addr=0x40600000 \
-dtb <path to DTB>/xen.dtb -kernel <path to Xen build>/xen
```

This will start you a Xen hypervisor with your application as Xen control domain. To make it usable, you can add *zephyr-xenlib* by Xen-troops library to your project. It'll provide basic domain management functionalities - domain creation and configuration.

5.4.3 Writing Snippets

- *Basics*
- *Namespacing*
- *Where snippets are located*
- *Processing order*
- *Devicetree overlays (`.overlay`)*
- *`.conf` files*
- *`DTS_EXTRA_CPPFLAGS`*
- *Board-specific settings*
  - *By name*
  - *By regular expression*

Basics

Snippets are defined using YAML files named `snippet.yml`.

A `snippet.yml` file contains the name of the snippet, along with additional build system settings, like this:

```
name: snippet-name
# ... build system settings go here ...
```

Build system settings go in other keys in the file as described later on in this page.

You can combine settings whenever they appear under the same keys. For example, you can combine a snippet-specific devicetree overlay and a `.conf` file like this:

```
name: foo
append:
  EXTRA_DTC_OVERLAY_FILE: foo.overlay
  EXTRA_CONF_FILE: foo.conf
```

Namespacing

When writing devicetree overlays in a snippet, use `snippet_<name>` or `snippet-<name>` as a namespace prefix when choosing names for node labels, node names, etc. This avoids namespace conflicts.

For example, if your snippet is named `foo-bar`, write your devicetree overlays like this:

```
chosen {
        zephyr,baz = &snippet_foo_bar_dev;
};

snippet_foo_bar_dev: device@12345678 {
        /* ... */
};
```

Where snippets are located

The build system looks for snippets in these places:

1. In directories configured by the **SNIPPET_ROOT** CMake variable. This always includes the zephyr repository (so snippets/ is always a source of snippets) and the application source directory (so `<app>/snippets` is also).

   Additional directories can be added manually at CMake time.

   The variable is a whitespace- or semicolon-separated list of directories which may contain snippet definitions.

   For each directory in the list, the build system looks for `snippet.yml` files underneath a subdirectory named `snippets/`, if one exists.

   For example, if **SNIPPET_ROOT** is set to `/foo;/bar`, the build system will look for `snippet.yml` files underneath the following subdirectories:

   - `/foo/snippets/`
   - `/bar/snippets/`

   The `snippet.yml` files can be nested anywhere underneath these locations.

2. In any *module* whose `module.yml` file provides a `snippet_root` setting.

   For example, in a zephyr module named `baz`, you can add this to your `module.yml` file:

   ```
   settings:
     snippet_root: .
   ```

   And then any `snippet.yml` files in `baz/snippets` will automatically be discovered by the build system, just as if the path to `baz` had appeared in **SNIPPET_ROOT**.

Processing order

Snippets are processed in the order they are listed in the **SNIPPET** variable, or in the order of the `-S` arguments if using west.

To apply `bar` after `foo`:

```
cmake -Sapp -Bbuild -DSNIPPET="foo;bar" [...]
cmake --build build
```

The same can be achieved with west as follows:

```
west build -S foo -S bar [...] app
```

When multiple snippets set the same configuration, the configuration value set by the last processed snippet ends up in the final configurations.

For instance, if `foo` sets `CONFIG_FOO=1` and `bar` sets `CONFIG_FOO=2` in the above example, the resulting final configuration will be `CONFIG_FOO=2` because `bar` is processed after `foo`.

This principle applies to both Kconfig fragments (`.conf` files) and devicetree overlays (`.overlay` files).

Devicetree overlays (`.overlay`)

This `snippet.yml` adds `foo.overlay` to the build:

```
name: foo
append:
  EXTRA_DTC_OVERLAY_FILE: foo.overlay
```

The path to `foo.overlay` is relative to the directory containing `snippet.yml`.

`.conf` files

This `snippet.yml` adds `foo.conf` to the build:

```
name: foo
append:
  EXTRA_CONF_FILE: foo.conf
```

The path to `foo.conf` is relative to the directory containing `snippet.yml`.

DTS_EXTRA_CPPFLAGS

This `snippet.yml` adds `DTS_EXTRA_CPPFLAGS` CMake Cache variables to the build:

```
name: foo
append:
  DTS_EXTRA_CPPFLAGS: -DMY_DTS_CONFIGURE
```

Adding these flags enables control over the content of a devicetree file.

Board-specific settings

You can write settings that only apply to some boards.

The settings described here are applied in **addition** to snippet settings that apply to all boards. (This is similar, for example, to the way that an application with both `prj.conf` and `boards/foo.conf` files will use both `.conf` files in the build when building for board `foo`, instead of just `boards/foo.conf`)

By name

```
name: ...
boards:
  bar: # settings for board "bar" go here
    append:
      EXTRA_DTC_OVERLAY_FILE: bar.overlay
  baz: # settings for board "baz" go here
    append:
      EXTRA_DTC_OVERLAY_FILE: baz.overlay
```

The above example uses `bar.overlay` when building for board `bar`, and `baz.overlay` when building for `baz`.

By regular expression You can enclose the board name in slashes (/) to match the name against a regular expression in the CMake syntax. The regular expression must match the entire board name.

For example:

```
name: foo
boards:
  /my_vendor_.*/:
    append:
      EXTRA_DTC_OVERLAY_FILE: my_vendor.overlay
```

The above example uses devicetree overlay `my_vendor.overlay` when building for either board `my_vendor_board1` or `my_vendor_board2`. It would not use the overlay when building for either `another_vendor_board` or `x_my_vendor_board`.

5.4.4 Snippets Design

This page documents design goals for the snippets feature. Further information can be found in Issue #51834.

- **extensible**: for example, it is possible to add board support for an existing built-in snippet without modifying the zephyr repository
- **composable**: it is possible to use multiple snippets at once, for example using:

  ```
  west build -S <snippet1> -S <snippet2> ...
  ```

- **able to combine multiple types of configuration**: snippets make it possible to store multiple different types of build system settings in one place, and apply them all together
- **specializable**: for example, it is possible to customize a snippet's behavior for a particular board, or board revision
- **future-proof and backwards-compatible**: arbitrary future changes to the snippets feature will be possible without breaking backwards compatibility for older snippets
- **applicable to purely "software" changes**: unlike the shields feature, snippets do not assume the presence of a "daughterboard", "shield", "hat", or any other type of external assembly which is connected to the main board
- **DRY** (don't repeat yourself): snippets allow you to skip unnecessary repetition; for example, you can apply the same board-specific configuration to boards `foo` and `bar` by specifying `/(foo|bar)/` as a regular expression for the settings, which will then apply to both boards

5.5 Zephyr CMake Package

The Zephyr CMake package is a convenient way to create a Zephyr-based application.

Note

The *Application types* section introduces the application types used in this page.

The Zephyr CMake package ensures that CMake can automatically select a Zephyr installation to use for building the application, whether it is a *Zephyr repository application*, a *Zephyr workspace application*, or a *Zephyr freestanding application*.

When developing a Zephyr-based application, then a developer simply needs to write `find_package(Zephyr)` in the beginning of the application `CMakeLists.txt` file.

To use the Zephyr CMake package it must first be exported to the CMake user package registry. This is means creating a reference to the current Zephyr installation inside the CMake user package registry.

Ubuntu

In Linux, the CMake user package registry is found in:

`~/.cmake/packages/Zephyr`

macOS

In macOS, the CMake user package registry is found in:

`~/.cmake/packages/Zephyr`

Windows

In Windows, the CMake user package registry is found in:

`HKEY_CURRENT_USER\Software\Kitware\CMake\Packages\Zephyr`

The Zephyr CMake package allows CMake to automatically find a Zephyr base. One or more Zephyr installations must be exported. Exporting multiple Zephyr installations may be useful when developing or testing Zephyr freestanding applications, Zephyr workspace application with vendor forks, etc..

5.5.1 Zephyr CMake package export (west)

When installing Zephyr using *west* then it is recommended to export Zephyr using `west zephyr-export`.

5.5.2 Zephyr CMake package export (without west)

Zephyr CMake package is exported to the CMake user package registry using the following commands:

```
cmake -P <PATH-TO-ZEPHYR>/share/zephyr-package/cmake/zephyr_export.cmake
```

This will export the current Zephyr to the CMake user package registry.

To also export the Zephyr Unittest CMake package, run the following command in addition:

```
cmake -P <PATH-TO-ZEPHYR>/share/zephyrunittest-package/cmake/zephyr_export.cmake
```

5.5.3 Zephyr Base Environment Setting

The Zephyr CMake package search functionality allows for explicitly specifying a Zephyr base using an environment variable.

To do this, use the following `find_package()` syntax:

```
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
```

This syntax instructs CMake to first search for Zephyr using the Zephyr base environment setting *ZEPHYR_BASE* and then use the normal search paths.

5.5.4 Zephyr CMake Package Search Order

When Zephyr base environment setting is not used for searching, the Zephyr installation matching the following criteria will be used:

- A Zephyr repository application will use the Zephyr in which it is located. For example:

  ```
  <projects>/zephyr-workspace/zephyr
  └── samples
      └── hello_world
  ```

  in this example, `hello_world` will use `<projects>/zephyr-workspace/zephyr`.

- Zephyr workspace application will use the Zephyr that share the same workspace. For example:

  ```
  <projects>/zephyr-workspace
  ├── zephyr
  ├── ...
  └── my_applications
       └── my_first_app
  ```

  in this example, `my_first_app` will use `<projects>/zephyr-workspace/zephyr` as this Zephyr is located in the same workspace as the Zephyr workspace application.

Note

The root of a Zephyr workspace is identical to `west topdir` if the workspace was installed using `west`

- Zephyr freestanding application will use the Zephyr registered in the CMake user package registry. For example:

```
<projects>/zephyr-workspace-1
└── zephyr                        (Not exported to CMake)

<projects>/zephyr-workspace-2
└── zephyr                        (Exported to CMake)

<home>/app
├── CMakeLists.txt
├── prj.conf
└── src
    └── main.c
```

in this example, only `<projects>/zephyr-workspace-2/zephyr` is exported to the CMake package registry and therefore this Zephyr will be used by the Zephyr freestanding application `<home>/app`.

If user wants to test the application with `<projects>/zephyr-workspace-1/zephyr`, this can be done by using the Zephyr Base environment setting, meaning set `ZEPHYR_BASE=<projects>/zephyr-workspace-1/zephyr`, before running CMake.

Note

The Zephyr package selected on the first CMake invocation will be used for all subsequent builds. To change the Zephyr package, for example to test the application using Zephyr base environment setting, then it is necessary to do a pristine build first (See *Rebuilding an Application*).

5.5.5 Zephyr CMake Package Version

When writing an application then it is possible to specify a Zephyr version number `x.y.z` that must be used in order to build the application.

Specifying a version is especially useful for a Zephyr freestanding application as it ensures the application is built with a minimal Zephyr version.

It also helps CMake to select the correct Zephyr to use for building, when there are multiple Zephyr installations in the system.

For example:

```
find_package(Zephyr 2.2.0)
project(app)
```

will require app to be built with Zephyr 2.2.0 as minimum. CMake will search all exported candidates to find a Zephyr installation which matches this version criteria.

Thus it is possible to have multiple Zephyr installations and have CMake automatically select between them based on the version number provided, see CMake package version for details.

For example:

```
<projects>/zephyr-workspace-2.a
└── zephyr                        (Exported to CMake)

<projects>/zephyr-workspace-2.b
└── zephyr                        (Exported to CMake)

<home>/app
├── CMakeLists.txt
├── prj.conf
└── src
    └── main.c
```

in this case, there are two released versions of Zephyr installed at their own workspaces. Workspace 2.a and 2.b, corresponding to the Zephyr version.

To ensure app is built with minimum version `2.a` the following `find_package` syntax may be used:

```
find_package(Zephyr 2.a)
project(app)
```

Note that both `2.a` and `2.b` fulfill this requirement.

CMake also supports the keyword `EXACT`, to ensure an exact version is used, if that is required. In this case, the application CMakeLists.txt could be written as:

```
find_package(Zephyr 2.a EXACT)
project(app)
```

In case no Zephyr is found which satisfies the version required, as example, the application specifies

```
find_package(Zephyr 2.z)
project(app)
```

then an error similar to below will be printed:

```
Could not find a configuration file for package "Zephyr" that is compatible
with requested version "2.z".

The following configuration files were considered but not accepted:

  <projects>/zephyr-workspace-2.a/zephyr/share/zephyr-package/cmake/ZephyrConfig.cmake,␣
↪version: 2.a.0
  <projects>/zephyr-workspace-2.b/zephyr/share/zephyr-package/cmake/ZephyrConfig.cmake,␣
↪version: 2.b.0
```

Note

It can also be beneficial to specify a version number for Zephyr repository applications and Zephyr workspace applications. Specifying a version in those cases ensures the application will only build if the Zephyr repository or workspace is matching. This can be useful to avoid accidental builds when only part of a workspace has been updated.

5.5.6 Multiple Zephyr Installations (Zephyr workspace)

Testing out a new Zephyr version, while at the same time keeping the existing Zephyr in the workspace untouched is sometimes beneficial.

Or having both an upstream Zephyr, Vendor specific, and a custom Zephyr in same workspace.

For example:

```
<projects>/zephyr-workspace
├── zephyr
├── zephyr-vendor
├── zephyr-custom
├── ...
└── my_applications
    └── my_first_app
```

in this setup, `find_package(Zephyr)` has the following order of precedence for selecting which Zephyr to use:

- Project name: `zephyr`
- First project, when Zephyr projects are ordered lexicographical, in this case.
  - `zephyr-custom`
  - `zephyr-vendor`

This means that `my_first_app` will use `<projects>/zephyr-workspace/zephyr`.

It is possible to specify a Zephyr preference list in the application.

A Zephyr preference list can be specified as:

```
set(ZEPHYR_PREFER "zephyr-custom" "zephyr-vendor")
find_package(Zephyr)

project(my_first_app)
```

the `ZEPHYR_PREFER` is a list, allowing for multiple Zephyrs. If a Zephyr is specified in the list, but not found in the system, it is simply ignored and `find_package(Zephyr)` will continue to the next candidate.

This allows for temporary creation of a new Zephyr release to be tested, without touching current Zephyr. When testing is done, the `zephyr-test` folder can simply be removed. Such a CMakeLists.txt could look as:

```
set(ZEPHYR_PREFER "zephyr-test")
find_package(Zephyr)

project(my_first_app)
```

5.5.7 Zephyr Build Configuration CMake packages

There are two Zephyr Build configuration packages which provide control over the build settings in Zephyr in a more generic way. These packages are:

- **ZephyrBuildConfiguration**: Applies to all Zephyr applications in the workspace
- **ZephyrAppConfiguration**: Applies only to the application you are currently building

They are similar to the per-user `.zephyrrc` file that can be used to set *Environment Variables*, but they set CMake variables instead. They also allow you to automatically share the build configuration among all users through the project repository. They also allow more advanced use cases, such as loading of additional CMake boilerplate code.

The Zephyr Build Configuration CMake packages will be loaded in the Zephyr boilerplate code after initial properties and `ZEPHYR_BASE` has been defined, but before CMake code execution. The ZephyrBuildConfiguration is included first and ZephyrAppConfiguration afterwards. That means the application-specific package could override the workspace settings, if needed. This

allows the Zephyr Build Configuration CMake packages to setup or extend properties such as: DTS_ROOT, BOARD_ROOT, TOOLCHAIN_ROOT / other toolchain setup, fixed overlays, and any other property that can be controlled. It also allows inclusion of additional boilerplate code.

To provide a ZephyrBuildConfiguration or ZephyrAppConfiguration, create ZephyrBuildConfig.cmake and/or ZephyrAppConfig.cmake respectively and place them in the appropriate location. The CMake find_package mechanism will search for these files with the steps below. Other default CMake package search paths and hints are disabled and there is no version checking implemented for these packages. This also means that these packages cannot be installed in the CMake package registry. The search steps are:

1. If ZephyrBuildConfiguration_ROOT, or ZephyrAppConfiguration_ROOT respectively, is set, search within this prefix path. If a matching file is found, execute this file. If no matching file is found, go to step 2.
2. Search within ${ZEPHYR_BASE}/../*, or ${APPLICATION_SOURCE_DIR} respectively. If a matching file is found, execute this file. If no matching file is found, abort the search.

It is recommended to place the files in the default paths from step 2, but with the <PackageName>_ROOT variables you have the flexibility to place them anywhere. This is especially necessary for freestanding applications, for which the default path to ZephyrBuildConfiguration usually does not work. In this case the <PackageName>_ROOT variables can be set on the CMake command line, **before** find_package(Zephyr ...), as environment variable or from a CMake cache initialization file with the -C command line option.

Note

The <PackageName>_ROOT variables, as well as the default paths, are just the prefixes to the search path. These prefixes get combined with additional path suffixes, which together form the actual search path. Any combination that honors the CMake package search procedure is valid and will work.

If you want to completely disable the search for these packages, you can use the special CMake CMAKE_DISABLE_FIND_PACKAGE_<PackageName> variable for that. Just set CMAKE_DISABLE_FIND_PACKAGE_ZephyrBuildConfiguration or CMAKE_DISABLE_FIND_PACKAGE_ZephyrAppConfiguration to TRUE to disable the package.

An example folder structure could look like this:

```
<projects>/zephyr-workspace
├── zephyr
├── ...
├── manifest repo (can be named anything)
|    └── cmake/ZephyrBuildConfig.cmake
├── ...
└── zephyr application
     └── share/zephyrapp-package/cmake/ZephyrAppConfig.cmake
```

A sample ZephyrBuildConfig.cmake can be seen below.

```
# ZephyrBuildConfig.cmake sample code

# To ensure final path is absolute and does not contain ../.. in variable.
get_filename_component(APPLICATION_PROJECT_DIR
                       ${CMAKE_CURRENT_LIST_DIR}/../../..
                       ABSOLUTE
)

# Add this project to list of board roots
list(APPEND BOARD_ROOT ${APPLICATION_PROJECT_DIR})
```

(continues on next page)

(continued from previous page)

```
# Default to GNU Arm Embedded toolchain if no toolchain is set
if(NOT ENV{ZEPHYR_TOOLCHAIN_VARIANT})
    set(ZEPHYR_TOOLCHAIN_VARIANT gnuarmemb)
    find_program(GNU_ARM_GCC arm-none-eabi-gcc)
    if(NOT ${GNU_ARM_GCC} STREQUAL GNU_ARM_GCC-NOTFOUND)
        # The toolchain root is located above the path to the compiler.
        get_filename_component(GNUARMEMB_TOOLCHAIN_PATH ${GNU_ARM_GCC}/../.. ABSOLUTE)
    endif()
endif()
```

5.5.8 Zephyr CMake package source code

The Zephyr CMake package source code in share/zephyr-package/cmake and share/zephyrunittest-package/cmake contains the CMake config package which is used by the CMake `find_package` function.

It also contains code for exporting Zephyr as a CMake config package.

The following is an overview of the files in these directories:

ZephyrConfigVersion.cmake
: The Zephyr package version file. This file is called by CMake to determine if this installation fulfils the requirements specified by user when calling `find_package(Zephyr ...)`. It is also responsible for detection of Zephyr repository or workspace only installations.

ZephyrUnittestConfigVersion.cmake
: Same responsibility as `ZephyrConfigVersion.cmake`, but for unit tests. Includes `ZephyrConfigVersion.cmake`.

ZephyrConfig.cmake
: The Zephyr package file. This file is called by CMake to for the package meeting which fulfils the requirements specified by user when calling `find_package(Zephyr ...)`. This file is responsible for sourcing of boilerplate code.

ZephyrUnittestConfig.cmake
: Same responsibility as `ZephyrConfig.cmake`, but for unit tests. Includes `ZephyrConfig.cmake`.

zephyr_package_search.cmake
: Common file used for detection of Zephyr repository and workspace candidates. Used by `ZephyrConfigVersion.cmake` and `ZephyrConfig.cmake` for common code.

zephyr_export.cmake
: See *Zephyr CMake package export (without west)*.

5.6 Sysbuild (System build)

Sysbuild is a higher-level build system that can be used to combine multiple other build systems together. It is a higher-level layer that combines one or more Zephyr build systems and optional additional build systems into a hierarchical build system.

For example, you can use sysbuild to build a Zephyr application together with the MCUboot bootloader, flash them both onto your device, and debug the results.

Sysbuild works by configuring and building at least a Zephyr application and, optionally, as many additional projects as you want. The additional projects can be either Zephyr applications or other types of builds you want to run.

Like Zephyr's *build system*, sysbuild is written in CMake and uses *Kconfig*.

5.6.1 Definitions

The following are some key concepts used in this document:

Single-image build
When sysbuild is used to create and manage just one Zephyr application's build system.

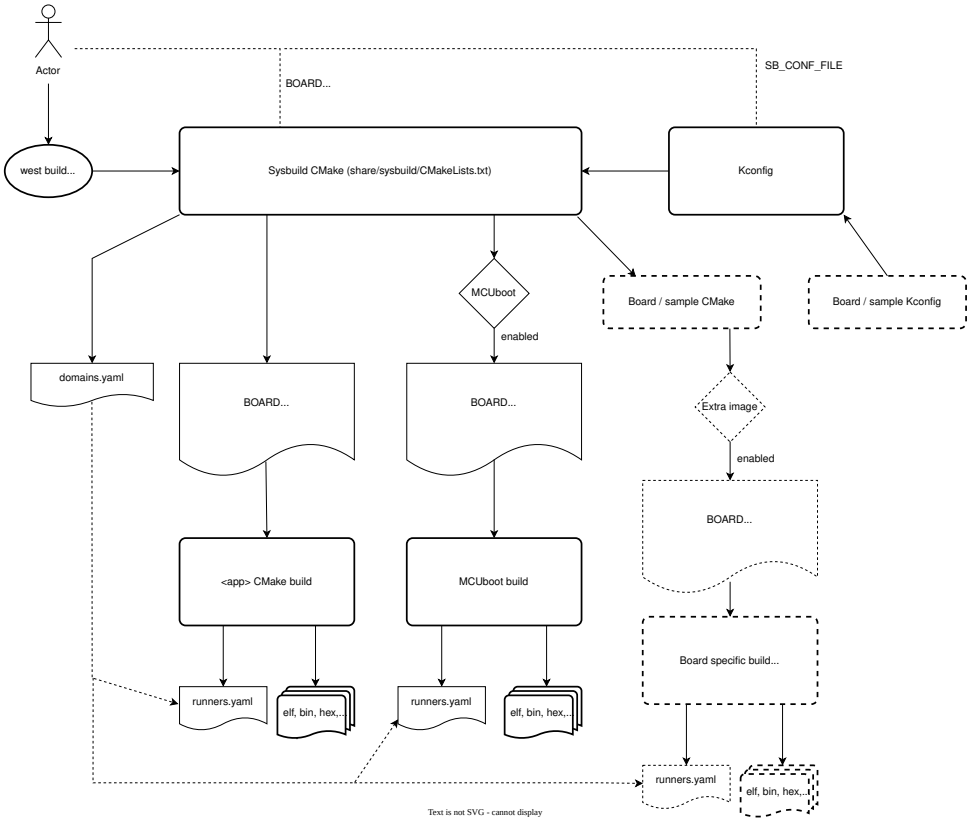
Multi-image build
When sysbuild is used to manage multiple build systems. The word "image" is used because your main goal is usually to generate the binaries of the firmware application images from each build system.

Domain
Every Zephyr CMake build system managed by sysbuild.

Multi-domain
When more than one Zephyr CMake build system (domain) is managed by sysbuild.

5.6.2 Architectural Overview

This figure is an overview of sysbuild's inputs, outputs, and user interfaces:



The following are some key sysbuild features indicated in this figure:

- You can run sysbuild either with *west build* or directly via `cmake`.
- You can use sysbuild to generate application images from each build system, shown above as ELF, BIN, and HEX files.
- You can configure sysbuild or any of the build systems it manages using various configuration variables. These variables are namespaced so that sysbuild can direct them to the right build system. In some cases, such as the `BOARD` variable, these are shared among multiple build systems.

- Sysbuild itself is also configured using Kconfig. For example, you can instruct sysbuild to build the MCUboot bootloader, as well as to build and link your main Zephyr application as an MCUboot-bootable image, using sysbuild's Kconfig files.
- Sysbuild integrates with west's *Building, Flashing and Debugging* commands. It does this by managing the *Flash and debug runners*, and specifically the `runners.yaml` files that each Zephyr build system will contain. These are packaged into a global view of how to flash and debug each build system in a `domains.yaml` file generated and managed by sysbuild.
- Build names are prefixed with the target name and an underscore, for example the sysbuild target is prefixed with `sysbuild_` and if MCUboot is enabled as part of sysbuild, it will be prefixed with `mcuboot_`. This also allows for running things like menuconfig with the prefix, for example (if using ninja) `ninja sysbuild_menuconfig` to configure sysbuild or (if using make) `make mcuboot_menuconfig`.

5.6.3 Building with sysbuild

As mentioned above, you can run sysbuild via `west build` or `cmake`.

`west build`

Here is an example. For details, see *Sysbuild (multi-domain builds)* in the `west build` documentation.

```
west build -b reel_board --sysbuild samples/hello_world
```

Tip

To configure `west build` to use `--sysbuild` by default from now on, run:

```
west config build.sysbuild True
```

Since sysbuild supports both single- and multi-image builds, this lets you use sysbuild all the time, without worrying about what type of build you are running.

To turn this off, run this before generating your build system:

```
west config build.sysbuild False
```

To turn this off for just one `west build` command, run:

```
west build --no-sysbuild ...
```

`cmake`

Here is an example using CMake and Ninja.

```
cmake -Bbuild -GNinja -DBOARD=reel_board -DAPP_DIR=samples/hello_world share/sysbuild
ninja -Cbuild
```

To use sysbuild directly with CMake, you must specify the sysbuild project as the source folder, and give `-DAPP_DIR=<path-to-sample>` as an extra CMake argument. `APP_DIR` is the path to the main Zephyr application managed by sysbuild.

5.6.4 Configuration namespacing

When building a single Zephyr application without sysbuild, all CMake cache settings and Kconfig build options given on the command line as `-D<var>=<value>` or `-DCONFIG_<var>=<value>` are handled by the Zephyr build system.

However, when sysbuild combines multiple Zephyr build systems, there could be Kconfig settings exclusive to sysbuild (and not used by any of the applications). To handle this, sysbuild has namespaces for configuration variables. You can use these namespaces to direct settings either to sysbuild itself or to a specific Zephyr application managed by sysbuild using the information in these sections.

The following example shows how to build hello_world with MCUboot enabled, applying to both images debug optimizations:

west build

```
west build -b reel_board --sysbuild samples/hello_world -- -DSB_CONFIG_BOOTLOADER_MCUBOOT=y␣
↪-DCONFIG_DEBUG_OPTIMIZATIONS=y -Dmcuboot_CONFIG_DEBUG_OPTIMIZATIONS=y
```

cmake

```
cmake -Bbuild -GNinja -DBOARD=reel_board -DAPP_DIR=samples/hello_world -DSB_CONFIG_
↪BOOTLOADER_MCUBOOT=y -DCONFIG_DEBUG_OPTIMIZATIONS=y -Dmcuboot_CONFIG_DEBUG_
↪OPTIMIZATIONS=y share/sysbuild
ninja -Cbuild
```

See the following subsections for more information.

CMake variable namespacing

CMake variable settings can be passed to CMake using `-D<var>=<value>` on the command line. You can also set Kconfig options via CMake as `-DCONFIG_<var>=<value>` or `-D<namespace>_CONFIG_<var>=<value>`.

Since sysbuild is the entry point for the build system, and sysbuild is written in CMake, all CMake variables are first processed by sysbuild.

Sysbuild creates a namespace for each domain. The namespace prefix is the domain's application name. See *Adding Zephyr applications to sysbuild* for more information.

To set the variable `<var>` in the namespace `<namespace>`, use this syntax:

```
-D<namespace>_<var>=<value>
```

For example, to set the CMake variable `FOO` in the `my_sample` application build system to the value `BAR`, run the following commands:

west build

```
west build --sysbuild ... -- -Dmy_sample_FOO=BAR
```

cmake

```
cmake -Dmy_sample_FOO=BAR ...
```

Kconfig namespacing

To set the sysbuild Kconfig option `<var>` to the value `<value>`, use this syntax:

```
-DSB_CONFIG_<var>=<value>
```

In the previous example, `SB_CONFIG` is the namespace prefix for sysbuild's Kconfig options.

To set a Zephyr application's Kconfig option instead, use this syntax:

```
-D<namespace>_CONFIG_<var>=<value>
```

In the previous example, <namespace> is the application name discussed above in *CMake variable namespacing*.

For example, to set the Kconfig option FOO in the my_sample application build system to the value BAR, run the following commands:

west build

```
west build --sysbuild ... -- -Dmy_sample_CONFIG_FOO=BAR
```

cmake

```
cmake -Dmy_sample_CONFIG_FOO=BAR ...
```

Tip

When no <namespace> is used, the Kconfig setting is passed to the main Zephyr application my_sample.

This means that passing -DCONFIG_<var>=<value> and -Dmy_sample_CONFIG_<var>=<value> are equivalent.

This allows you to build the same application with or without sysbuild using the same syntax for setting Kconfig values at CMake time. For example, the following commands will work in the same way:

```
west build -b <board> my_sample -- -DCONFIG_FOO=BAR
```

```
west build -b <board> --sysbuild my_sample -- -DCONFIG_FOO=BAR
```

5.6.5 Sysbuild flashing using west flash

You can use *west flash* to flash applications with sysbuild.

When invoking west flash on a build consisting of multiple images, each image is flashed in sequence. Extra arguments such as --runner jlink are passed to each invocation.

For more details, see *Multi-domain flashing*.

5.6.6 Sysbuild debugging using west debug

You can use west debug to debug the main application, whether you are using sysbuild or not. Just follow the existing *west debug* guide to debug the main sample.

To debug a different domain (Zephyr application), such as mcuboot, use the --domain argument, as follows:

```
west debug --domain mcuboot
```

For more details, see *Multi-domain debugging*.

5.6.7 Building a sample with MCUboot

Sysbuild supports MCUboot natively.

To build a sample like `hello_world` with MCUboot, enable MCUboot and build and flash the sample as follows:

`west build`

```
west build -b reel_board --sysbuild samples/hello_world -- -DSB_CONFIG_BOOTLOADER_MCUBOOT=y
```

`cmake`

```
cmake -Bbuild -GNinja -DBOARD=reel_board -DAPP_DIR=samples/hello_world -DSB_CONFIG_
↪BOOTLOADER_MCUBOOT=y share/sysbuild
ninja -Cbuild
```

This builds `hello_world` and `mcuboot` for the `reel_board`, and then flashes both the `mcuboot` and `hello_world` application images to the board.

More detailed information regarding the use of MCUboot with Zephyr can be found in the MCUboot with Zephyr documentation page on the MCUboot website.

Note

The deprecated MCUBoot Kconfig option `CONFIG_ZEPHYR_TRY_MASS_ERASE` will perform a full chip erase when flashed. If this option is enabled, then flashing only MCUBoot, for example using `west flash --domain mcuboot`, may erase the entire flash, including the main application image.

5.6.8 Sysbuild Kconfig file

You can set sysbuild's Kconfig options for a single application using configuration files. By default, sysbuild looks for a configuration file named `sysbuild.conf` in the application top-level directory.

In the following example, there is a `sysbuild.conf` file that enables building and flashing with MCUboot whenever sysbuild is used:

```
<home>/application
├── CMakeLists.txt
├── prj.conf
└── sysbuild.conf
```

```
SB_CONFIG_BOOTLOADER_MCUBOOT=y
```

You can set a configuration file to use with the `-DSB_CONF_FILE=<sysbuild-conf-file>` CMake build setting.

For example, you can create `sysbuild-mcuboot.conf` and then specify this file when building with sysbuild, as follows:

`west build`

```
west build -b reel_board --sysbuild samples/hello_world -- -DSB_CONF_FILE=sysbuild-mcuboot.
↪conf
```

`cmake`

```
cmake -Bbuild -GNinja -DBOARD=reel_board -DAPP_DIR=samples/hello_world -DSB_CONF_
↪FILE=sysbuild-mcuboot.conf share/sysbuild
ninja -Cbuild
```

5.6.9 Sysbuild targets

Sysbuild creates build targets for each image (including sysbuild itself) for the following modes:

- menuconfig
- hardenconfig
- guiconfig

For the main application (as is the same without using sysbuild) these can be ran normally without any prefix. For other images (including sysbuild), these are ran with a prefix of the image name and an underscore e.g. `sysbuild_` or `mcuboot_`, using ninja or make - for details on how to run image build targets that do not have mapped build targets in sysbuild, see the *Dedicated image build targets* section.

5.6.10 Dedicated image build targets

Not all build targets for images are given equivalent prefixed build targets when sysbuild is used, for example build targets like `ram_report`, `rom_report`, `footprint`, `puncover` and `pahole` are not exposed. When using *Trusted Firmware*, this includes build targets prefix with `tfm_` and `bl2_`, for example: `tfm_rom_report` and `bl2_ram_report`. To run these build targets, the build directory of the image can be provided to west/ninja/make along with the name of the build target to execute and it will run.

`west`

Assuming that a project has been configured and built using `west` using sysbuild with mcuboot enabled in the default `build` folder location, the `rom_report` build target for `mcuboot` can be ran with:

```
west build -d build/mcuboot -t rom_report
```

`ninja`

Assuming that a project has been configured using `cmake` and built using `ninja` using sysbuild with mcuboot enabled, the `rom_report` build target for `mcuboot` can be ran with:

```
ninja -C mcuboot rom_report
```

`make`

Assuming that a project has been configured using `cmake` and built using `make` using sysbuild with mcuboot enabled, the `rom_report` build target for `mcuboot` can be ran with:

```
make -C mcuboot rom_report
```

5.6.11 Adding Zephyr applications to sysbuild

You can use the `ExternalZephyrProject_Add()` function to add Zephyr applications as sysbuild domains. Call this CMake function from your application's `sysbuild.cmake` file, or any other CMake file you know will run as part sysbuild CMake invocation.

Targeting the same board

To include `my_sample` as another sysbuild domain, targeting the same board as the main image, use this example:

```
ExternalZephyrProject_Add(
  APPLICATION my_sample
  SOURCE_DIR <path-to>/my_sample
)
```

This could be useful, for example, if your board requires you to build and flash an SoC-specific bootloader along with your main application.

Targeting a different board

In sysbuild and Zephyr CMake build system a board may refer to:

- A physical board with a single core SoC.
- A specific core on a physical board with a multi-core SoC, such as nrf5340dk_nrf5340.
- A specific SoC on a physical board with multiple SoCs, such as nrf9160dk_nrf9160 and nrf9160dk_nrf52840.

If your main application, for example, is built for `mps2_an521`, and your helper application must target the `mps2_an521_remote` board (cpu1), add a CMake function call that is structured as follows:

```
ExternalZephyrProject_Add(
  APPLICATION my_sample
  SOURCE_DIR <path-to>/my_sample
  BOARD mps2_an521_remote
)
```

This could be useful, for example, if your main application requires another helper Zephyr application to be built and flashed alongside it, but the helper runs on another core in your SoC.

Targeting conditionally using Kconfig

You can control whether extra applications are included as sysbuild domains using Kconfig.

If the extra application image is specific to the board or an application, you can create two additional files: `sysbuild.cmake` and `Kconfig.sysbuild`.

For an application, this would look like this:

```
<home>/application
├── CMakeLists.txt
├── prj.conf
├── Kconfig.sysbuild
└── sysbuild.cmake
```

In the previous example, `sysbuild.cmake` would be structured as follows:

```
if(SB_CONFIG_SECOND_SAMPLE)
  ExternalZephyrProject_Add(
    APPLICATION second_sample
    SOURCE_DIR <path-to>/second_sample
  )
endif()
```

`Kconfig.sysbuild` would be structured as follows:

```
source "sysbuild/Kconfig"

config SECOND_SAMPLE
        bool "Second sample"
        default y
```

This will include `second_sample` by default, while still allowing you to disable it using the Kconfig option `SECOND_SAMPLE`.

For more information on setting sysbuild Kconfig options, see *Kconfig namespacing*.

Building without flashing

You can mark `my_sample` as a build-only application in this manner:

```
ExternalZephyrProject_Add(
  APPLICATION my_sample
  SOURCE_DIR <path-to>/my_sample
  BUILD_ONLY TRUE
)
```

As a result, `my_sample` will be built as part of the sysbuild build invocation, but it will be excluded from the default image sequence used by `west flash`. Instead, you may use the outputs of this domain for other purposes - for example, to produce a secondary image for DFU, or to merge multiple images together.

You can also replace `TRUE` with another boolean constant in CMake, such as a Kconfig option, which would make `my_sample` conditionally build-only.

Note

Applications marked as build-only can still be flashed manually, using `west flash --domain my_sample`. As such, the `BUILD_ONLY` option only controls the default behavior of `west flash`.

Zephyr application configuration

When adding a Zephyr application to sysbuild, such as MCUboot, then the configuration files from the application (MCUboot) itself will be used.

When integrating multiple applications with each other, then it is often necessary to make adjustments to the configuration of extra images.

Sysbuild gives users the ability of creating Kconfig fragments or devicetree overlays that will be used together with the application's default configuration. Sysbuild also allows users to change *Application Configuration Directory* in order to give users full control of an image's configuration.

Zephyr application Kconfig fragment and devicetree overlay In the folder of the main application, create a Kconfig fragment or a devicetree overlay under a sysbuild folder, where the name of the file is `<image>.conf` or `<image>.overlay`, for example if your main application includes `my_sample` then create a `sysbuild/my_sample.conf` file or a devicetree overlay `sysbuild/my_sample.overlay`.

A Kconfig fragment could look as:

```
# sysbuild/my_sample.conf
CONFIG_FOO=n
```

Zephyr application configuration directory In the folder of the main application, create a new folder under `sysbuild/<image>/`. This folder will then be used as `APPLICATION_CONFIG_DIR` when building `<image>`. As an example, if your main application includes `my_sample` then create a `sysbuild/my_sample/` folder and place any configuration files in there as you would normally do:

```
<home>/application
├── CMakeLists.txt
├── prj.conf
└── sysbuild
    └── my_sample
        ├── prj.conf
        ├── app.overlay
        └── boards
            ├── <board_A>.conf
            ├── <board_A>.overlay
            ├── <board_B>.conf
            └── <board_B>.overlay
```

All configuration files under the `sysbuild/my_sample/` folder will now be used when `my_sample` is included in the build, and the default configuration files for `my_sample` will be ignored.

This give you full control on how images are configured when integrating those with `application`.

Sysbuild file suffix support File suffix support through the makevar:*FILE_SUFFIX* is supported in sysbuild (see *File Suffixes* for details on this feature in applications). For sysbuild, a globally provided option will be passed down to all images. In addition, the image configuration file will have this value applied and used (instead of the build type) if the file exists.

Given the example project:

```
<home>/application
├── CMakeLists.txt
├── prj.conf
├── sysbuild.conf
├── sysbuild_test_key.conf
└── sysbuild
    ├── mcuboot.conf
    ├── mcuboot_max_log.conf
    └── my_sample.conf
```

- If `FILE_SUFFIX` is not defined and both `mcuboot` and `my_sample` images are included, `mcuboot` will use the `mcuboot.conf` Kconfig fragment file and `my_sample` will use the `my_sample.conf` Kconfig fragment file. Sysbuild itself will use the `sysbuild.conf` Kconfig fragment file.
- If `FILE_SUFFIX` is set to `max_log` and both `mcuboot` and `my_sample` images are included, `mcuboot` will use the `mcuboot_max_log.conf` Kconfig fragment file and `my_sample` will use the `my_sample.conf` Kconfig fragment file (as it will fallback to the file without the suffix). Sysbuild itself will use the `sysbuild.conf` Kconfig fragment file (as it will fallback to the file without the suffix).
- If `FILE_SUFFIX` is set to `test_key` and both `mcuboot` and `my_sample` images are included, `mcuboot` will use the `mcuboot.conf` Kconfig fragment file and `my_sample` will use the `my_sample.conf` Kconfig fragment file (as it will fallback to the files without the suffix). Sysbuild itself will use the `sysbuild_test_key.conf` Kconfig fragment file. This can be used to apply a different sysbuild configuration, for example to use a different signing key in MCUboot and when signing the main application.

The `FILE_SUFFIX` can also be applied only to single images by prefixing the variable with the image name:

```
west build
```

```
west build -b reel_board --sysbuild file_suffix_example -- -DSB_CONFIG_BOOTLOADER_MCUBOOT=y␣
↪-Dmcuboot_FILE_SUFFIX="max_log"
```

cmake

```
cmake -Bbuild -GNinja -DBOARD=reel_board -DAPP_DIR=<app_dir> -DSB_CONFIG_BOOTLOADER_
↪MCUBOOT=y -Dmcuboot_FILE_SUFFIX="max_log" share/sysbuild
ninja -Cbuild
```

Adding dependencies among Zephyr applications

Sometimes, in a multi-image build, you may want certain Zephyr applications to be configured or flashed in a specific order. For example, if you need some information from one application's build system to be available to another's, then the first thing to do is to add a configuration dependency between them. Separately, you can also add flashing dependencies to control the sequence of images used by `west flash`; this could be used if a specific flashing order is required by an SoC, a _runner_, or something else.

By default, sysbuild will configure and flash applications in the order that they are added, as `ExternalZephyrProject_Add()` calls are processed by CMake. You can use the `sysbuild_add_dependencies()` function to make adjustments to this order, according to your needs. Its usage is similar to the standard `add_dependencies()` function in CMake.

Here is an example of adding configuration dependencies for `my_sample`:

```
sysbuild_add_dependencies(IMAGE CONFIGURE my_sample sample_a sample_b)
```

This will ensure that sysbuild will run CMake for `sample_a` and `sample_b` (in some order) before doing the same for `my_sample`, when building these domains in a single invocation.

If you want to add flashing dependencies instead, then do it like this:

```
sysbuild_add_dependencies(IMAGE FLASH my_sample sample_a sample_b)
```

As a result, `my_sample` will be flashed after `sample_a` and `sample_b` (in some order), when flashing these domains in a single invocation.

Note

Adding flashing dependencies is not allowed for build-only applications. If `my_sample` had been created with `BUILD_ONLY TRUE`, then the above call to `sysbuild_add_dependencies()` would have produced an error.

5.6.12 Adding non-Zephyr applications to sysbuild

You can include non-Zephyr applications in a multi-image build using the standard CMake module ExternalProject. Please refer to the CMake documentation for usage details.

When using `ExternalProject`, the non-Zephyr application will be built as part of the sysbuild build invocation, but `west flash` or `west debug` will not be aware of the application. Instead, you must manually flash and debug the application.

5.6.13 Extending sysbuild

Sysbuild can be extended by other modules to give it additional functionality or include other configuration or images, an example could be to add support for another bootloader or external

signing method.

Modules can be extended by adding custom CMake or Kconfig files as normal *modules* do, this will cause the files to be included in each image that is part of a project. Alternatively, there are *sysbuild-specific module extension* files which can be used to include CMake and Kconfig files for the overall sysbuild image itself, this is where e.g. a custom image for a particular board or SoC can be added.

5.7 Application version management

Zephyr supports an application version management system for applications which is built around the system that Zephyr uses for its own version system management. This allows applications to define a version file and have application (or module) code include the auto-generated file and be able to access it, just as they can with the kernel version. This version information is available from multiple scopes, including:

- Code (C/C++)
- Kconfig
- CMake

which makes it a very versatile system for lifecycle management of applications. In addition, it can be used when building applications which target supported bootloaders (e.g. MCUboot) allowing images to be signed with correct version of the application automatically - no manual signing steps are required.

5.7.1 VERSION file

Application version information is set on a per-application basis in a file named `VERSION`, which must be placed at the base directory of the application, where the CMakeLists.txt file is located. This is a simple text file which contains the various version information fields, each on a newline. The basic `VERSION` file has the following structure:

```
VERSION_MAJOR =
VERSION_MINOR =
PATCHLEVEL =
VERSION_TWEAK =
EXTRAVERSION =
```

Each field and the values it supports is described below. Zephyr limits the value of each numeric field to a single byte (note that there may be further restrictions depending upon what the version is used for, e.g. bootloaders might only support some of these fields or might place limits on the maximum values of fields):

Field	Data type
VERSION_MAJOR	Numerical (0-255)
VERSION_MINOR	Numerical (0-255)
PATCHLEVEL	Numerical (0-255)
VERSION_TWEAK	Numerical (0-255)
EXTRAVERSION	Alphanumerical (Lowercase a-z and 0-9)

When an application is configured using CMake, the version file will be automatically processed, and will be checked automatically each time the version is changed, so CMake does not need to be manually re-ran for changes to this file.

For the sections below, examples are provided for the following `VERSION` file:

```
VERSION_MAJOR = 1
VERSION_MINOR = 2
PATCHLEVEL = 3
VERSION_TWEAK = 4
EXTRAVERSION = unstable
```

5.7.2 Use in code

To use the version information in application code, the version file must be included, then the fields can be freely used. The include file name is `app_version.h` (no path is needed), the following defines are available:

Define	Type	Field(s)	Example
APPVERSION	Numerical	`VERSION_MAJOR` (left shifted by 24 bits), `VERSION_MINOR` (left shifted by 16 bits), `PATCHLEVEL` (left shifted by 8 bits), `VERSION_TWEAK`	0x1020304
APP_VERSION_N	Numerical	`VERSION_MAJOR` (left shifted by 16 bits), `VERSION_MINOR` (left shifted by 8 bits), `PATCHLEVEL`	0x10203
APP_VERSION_N	Numerical	`VERSION_MAJOR`	1
APP_VERSION_N	Numerical	`VERSION_MINOR`	2
APP_PATCHLEV	Numerical	`PATCHLEVEL`	3
APP_TWEAK	Numerical	`VERSION_TWEAK`	4
APP_VERSION_S	String (quoted)	`VERSION_MAJOR`, `VERSION_MINOR`, `PATCHLEVEL`, `EXTRAVERSION`	“1.2.3-unstable”
APP_VERSION_F	String (quoted)	`VERSION_MAJOR`, `VERSION_MINOR`, `PATCHLEVEL`, `EXTRAVERSION VERSION_TWEAK`	“1.2.3-unstable+4”
APP_VERSION_T	String (quoted)	`VERSION_MAJOR`, `VERSION_MINOR`, `PATCHLEVEL`, `VERSION_TWEAK`	“1.2.3+4”
APP_BUILD_VEF	String (unquoted)	None (value of `git describe --abbrev=12 --always` from application repository)	v3.3.0-18-g2c85d9224fca

5.7.3 Use in Kconfig

The following variables are available for usage in Kconfig files:

Variable	Type	Field(s)	Example
$(VERSION_MAJOR)	Numerical	VERSION_MAJOR	1
$(VERSION_MINOR)	Numerical	VERSION_MINOR	2
$(PATCHLEVEL)	Numerical	PATCHLEVEL	3
$(VERSION_TWEAK)	Numerical	VERSION_TWEAK	4
$(APPVERSION)	String	VERSION_MAJOR, VERSION_MINOR, PATCHLEVEL, EXTRAVERSION	1.2.3-unstable
$(APP_VERSION_EXTEN]	String	VERSION_MAJOR, VERSION_MINOR, PATCHLEVEL, EXTRAVERSION, VERSION_TWEAK	1.2.3-unstable+4
$(APP_VERSION_TWEAK	String	VERSION_MAJOR, VERSION_MINOR, PATCHLEVEL, VERSION_TWEAK	1.2.3+4

5.7.4 Use in CMake

The following variable are available for usage in CMake files:

Variable	Type	Field(s)	Example
APPVERSION	Numerical (hex)	VERSION_MAJOR (left shifted by 24 bits), VERSION_MINOR (left shifted by 16 bits), PATCHLEVEL (left shifted by 8 bits), VERSION_TWEAK	0x1020304
APP_VERSION_N	Numerical (hex)	VERSION_MAJOR (left shifted by 16 bits), VERSION_MINOR (left shifted by 8 bits), PATCHLEVEL	0x10203
APP_VERSION_M	Numerical	VERSION_MAJOR	1
APP_VERSION_M	Numerical	VERSION_MINOR	2
APP_PATCHLEVE	Numerical	PATCHLEVEL	3
APP_VERSION_T	Numerical	VERSION_TWEAK	4
APP_VERSION_S	String	VERSION_MAJOR, VERSION_MINOR, PATCHLEVEL, EXTRAVERSION	1.2.3-unstable
APP_VERSION_E	String	VERSION_MAJOR, VERSION_MINOR, PATCHLEVEL, EXTRAVERSION, VERSION_TWEAK	1.2.3-unstable+4
APP_VERSION_T	String	VERSION_MAJOR, VERSION_MINOR, PATCHLEVEL, VERSION_TWEAK	1.2.3+4

5.7.5 Use in MCUboot-supported applications

No additional configuration needs to be done to the target application so long as it is configured to support MCUboot and a signed image is generated, the version information will be automatically included in the image data.

5.8 Flashing

5.8.1 Flashing configuration

Zephyr supports setting up configuration for flash runners (invoked from *west flash*) which allows for customising how commands are used when programming boards. This configuration is used for *Sysbuild (System build)* projects and allows for configuring when commands are ran for groups of board targets. As an example: a multi-core SoC might want to only allow the `--erase` argument to be used once for all of the cores in the SoC which would prevent multiple erase tasks running in a single `west flash` invocation, which could wrongly clear the memory which is used by the other images being programmed.

Priority

Flashing configuration is singular, it will only be read from a single location, this configuration can reside in the following files starting with the highest priority:

- `soc.yml` (in soc folder)
- `board.yml` (in board folder)

Configuration

Configuration is applied in the yml file by using a `runners` map with a single `run_once` child, this then contains a map of commands as they would be provided to the flash runner e.g. `--reset` followed by a list which specifies the settings for each of these commands (these are grouped by flash runner, and by qualifiers/boards). Commands have associated runners that they apply to using a `runners` list value, this can contain `all` if it applies to all runners, otherwise must contain each runner that it applies to using the runner-specific name. Groups of board targets can be specified using the `groups` key which has a list of board target sets. Board targets are regular expression matches, for `soc.yml` files each set of board targets must be in a `qualifiers` key (only regular expression matches for board qualifiers are allowed, board names must be omitted from these entries). For `board.yml` files each set of board targets must be in a `boards` key, these are lists containing the matches which form a singular group. A final parameter `run` can be set to `first` which means that the command will be ran once with the first image flashing process per set of board targets, or to `last` which will be ran once for the final image flash per set of board targets.

An example flashing configuration for a `soc.yml` is shown below in which the `--recover` command will only be used once for any board targets which used the nRF5340 SoC application or network CPU cores, and will only reset the network or application core after all images for the respective core have been flashed.

```
runners:
  run_once:
    '--recover':
      - run: first
        runners:
          - nrfjprog
```

(continues on next page)

(continued from previous page)

```
        groups:
          - qualifiers:
              - nrf5340/cpunet
              - nrf5340/cpuapp
              - nrf5340/cpuapp/ns
    '--reset':
      - run: last
        runners:
          - nrfjprog
          - jlink
        groups:
          - qualifiers:
              - nrf5340/cpunet
          - qualifiers:
              - nrf5340/cpuapp
              - nrf5340/cpuapp/ns
      # Made up non-real world example to show how to specify different options for␣
↪different
      # flash runners
      - run: first
        runners:
          - some_other_runner
        groups:
          - qualifiers:
              - nrf5340/cpunet
          - qualifiers:
              - nrf5340/cpuapp
              - nrf5340/cpuapp/ns
```

Usage

Commands that are supported by flash runners can be used as normal when flashing non-sysbuild applications, the run once configuration will not be used. When flashing a sysbuild project with multiple images, the flash runner run once configuration will be applied.

For example, building the smp-svr sample for the nrf5340dk which will include MCUboot as a secondary image:

```
cmake -GNinja -Sshare/sysbuild/ -Bbuild -DBOARD=nrf5340dk/nrf5340/cpuapp -DAPP_DIR=samples/
↪subsys/mgmt/mcumgr/smp_svr
cmake --build build
```

Once built with an nrf5340dk connected, the following command can be used to flash the board with both applications and will only perform a single device recovery operation when programming the first image:

```
west flash --recover
```

If the above was ran without the flashing configuration, the recovery process would be ran twice and the device would be unbootable.

Chapter 6

Connectivity

6.1 Bluetooth

This section contains information regarding the Bluetooth stack of the Zephyr OS. You can use this information to understand the principles behind the operation of the layers and how they were implemented.

Zephyr includes a complete Bluetooth Low Energy stack from application to radio hardware, as well as portions of a Classical Bluetooth (BR/EDR) Host layer.

6.1.1 Supported features

Since its inception, Zephyr has had a strong focus on Bluetooth and, in particular, on Bluetooth Low Energy (BLE). Through the contributions of several companies and individuals involved in existing open source implementations of the Bluetooth specification (Linux's BlueZ) as well as the design and development of BLE radio hardware, the protocol stack in Zephyr has grown to be mature and feature-rich, as can be seen in the section below.

- Bluetooth v5.3 compliant
  - Highly configurable
    - Features, buffer sizes/counts, stack sizes, etc.
  - Portable to all architectures supported by Zephyr (including big and little endian, alignment flavors and more)
  - Support for *all combinations* of Host and Controller builds:
    - Controller-only (HCI) over UART, SPI, USB and IPC physical transports
    - Host-only over UART, SPI, and IPC (shared memory)
    - Combined (Host + Controller)
- *Bluetooth-SIG qualifiable*
  - Conformance tests run regularly on all layers (Controller and Host, except BT Classic) on Nordic Semiconductor hardware.
- *Bluetooth Low Energy Controller* (LE Link Layer)
  - Unlimited role and connection count, all roles supported
  - All v5.3 specification features supported (except a few minor items)
  - Concurrent multi-protocol support ready
  - Intelligent scheduling of roles to minimize overlap

  - Portable design to any open BLE radio, currently supports Nordic Semiconductor nRF52x and nRF53x SoC Series, as well as proprietary radios
  - Supports little and big endian architectures, and abstracts the hard real-time specifics so that they can be encapsulated in a hardware-specific module
  - Support for Controller (HCI) builds over different physical transports
  - Isochronous channels
- *Bluetooth Host*
  - Generic Access Profile (GAP) with all possible LE roles
    * Peripheral & Central
    * Observer & Broadcaster
    * Multiple PHY support (2Mbit/s, Coded)
    * Extended Advertising
    * Periodic Advertising (including Sync Transfer)
  - GATT (Generic Attribute Profile)
    * Server (to be a sensor)
    * Client (to connect to sensors)
    * Enhanced ATT (EATT)
    * GATT Database Hash
    * GATT Multiple Notifications
  - Pairing support, including the Secure Connections feature from Bluetooth 4.2
  - Non-volatile storage support for permanent storage of Bluetooth-specific settings and data
  - Bluetooth Mesh support
    * Relay, Friend Node, Low-Power Node (LPN) and GATT Proxy features
    * Both Provisioning roles and bearers supported (PB-ADV & PB-GATT)
    * Foundation Models included
    * Highly configurable, fits as small as 16k RAM devices
  - Basic Bluetooth BR/EDR (Classic) support
    * Generic Access Profile (GAP)
    * Logical Link Control and Adaptation Protocol (L2CAP)
    * Serial Port emulation (RFCOMM protocol)
    * Service Discovery Protocol (SDP)
  - Clean HCI driver abstraction
    * 3-Wire (H:5) & 5-Wire (H:4) UART
    * SPI
    * Local controller support as a virtual HCI driver
  - Verified with multiple popular controllers
  - Isochronous channels
  - *LE Audio*

6.1.2 Qualification

Qualification setup

The Zephyr Bluetooth host can be qualified using Bluetooth's PTS (Profile Tuning Suite) software. It is originally a manual process, but is automated by using the AutoPTS automation software.

The setup is described in more details in the pages linked below.

AutoPTS on Windows 10 with nRF52 board This tutorial shows how to setup AutoPTS client and server to run both on Windows 10. We use WSL1 with Ubuntu only to build a Zephyr project to an elf file, because Zephyr SDK is not available on Windows yet. Tutorial covers only nrf52840dk.

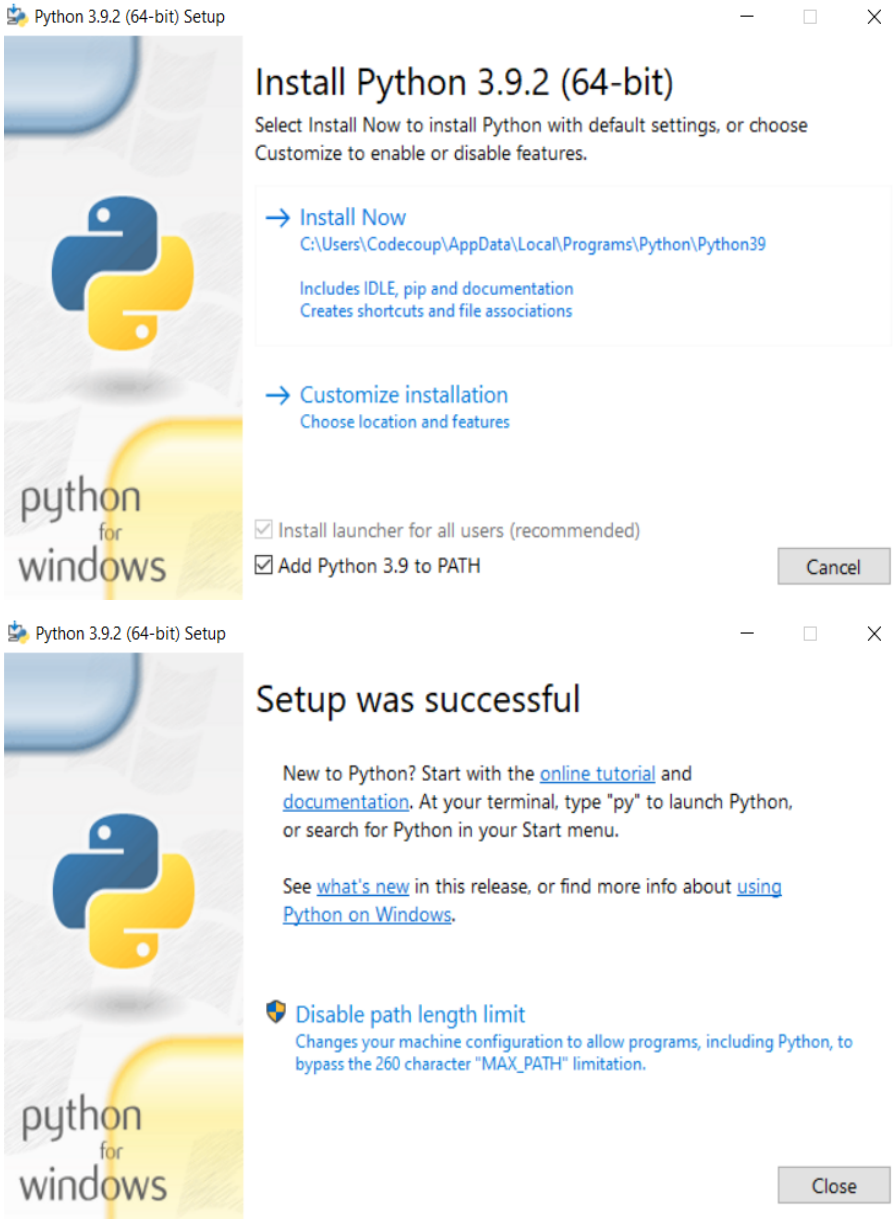
- *Update Windows and drivers*
- *Install Python 3*
- *Install Git*
- *Install PTS 8*
- *Setup Zephyr project for Windows*
- *Install nrftools*
- *Connect devices*
- *Flash board*
- *Setup auto-pts project*
- *Install socat.exe*
- *Running AutoPTS*
- *Troubleshooting*

Update Windows and drivers Update Windows in:

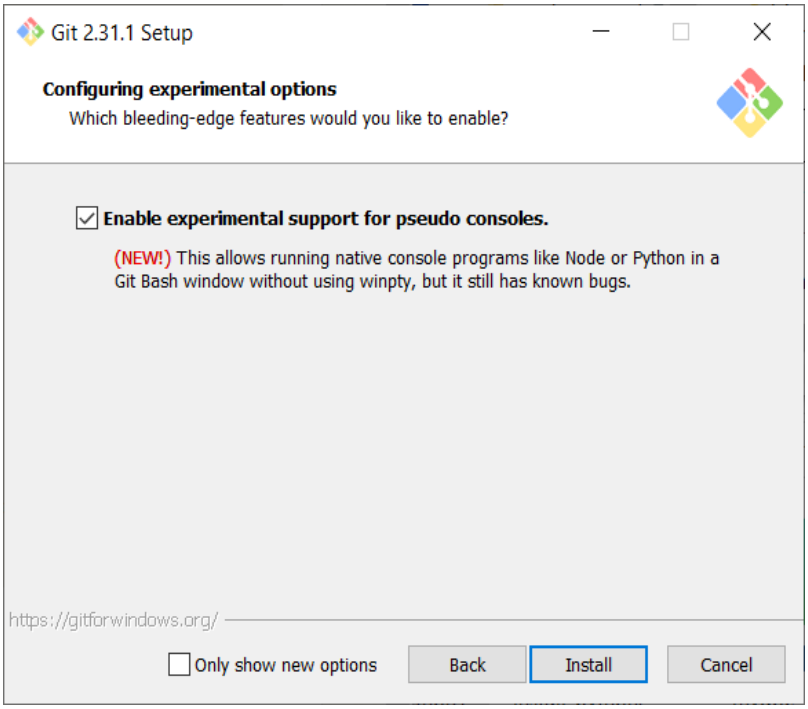
Start -> Settings -> Update & Security -> Windows Update

Update drivers, following the instructions from your hardware vendor.

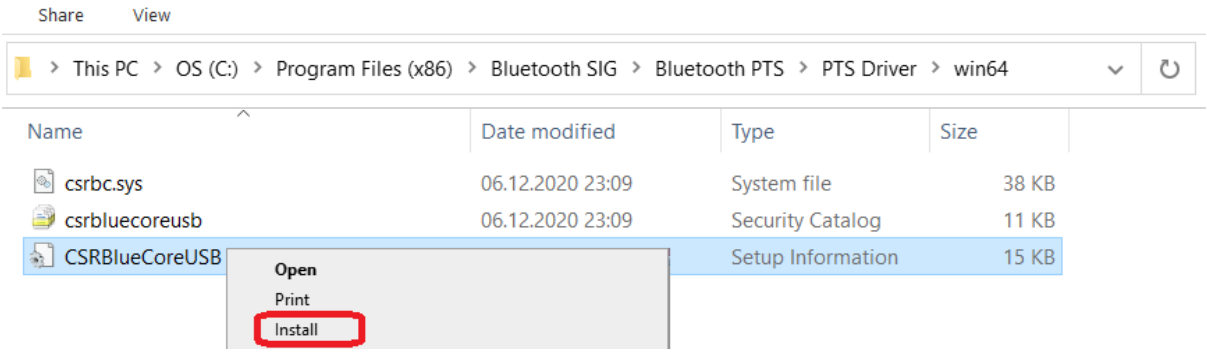
Install Python 3 Download and install Python 3. Setup was tested with versions >=3.8. Let the installer add the Python installation directory to the PATH and disable the path length limitation.



Install Git Download and install Git. During installation enable option: Enable experimental support for pseudo consoles. We will use Git Bash as Windows terminal.



Install PTS 8 Install latest PTS from https://www.bluetooth.org. Remember to install drivers from installation directory "C:/Program Files (x86)/Bluetooth SIG/Bluetooth PTS/PTS Driver/win64/CSRBlueCoreUSB.inf"

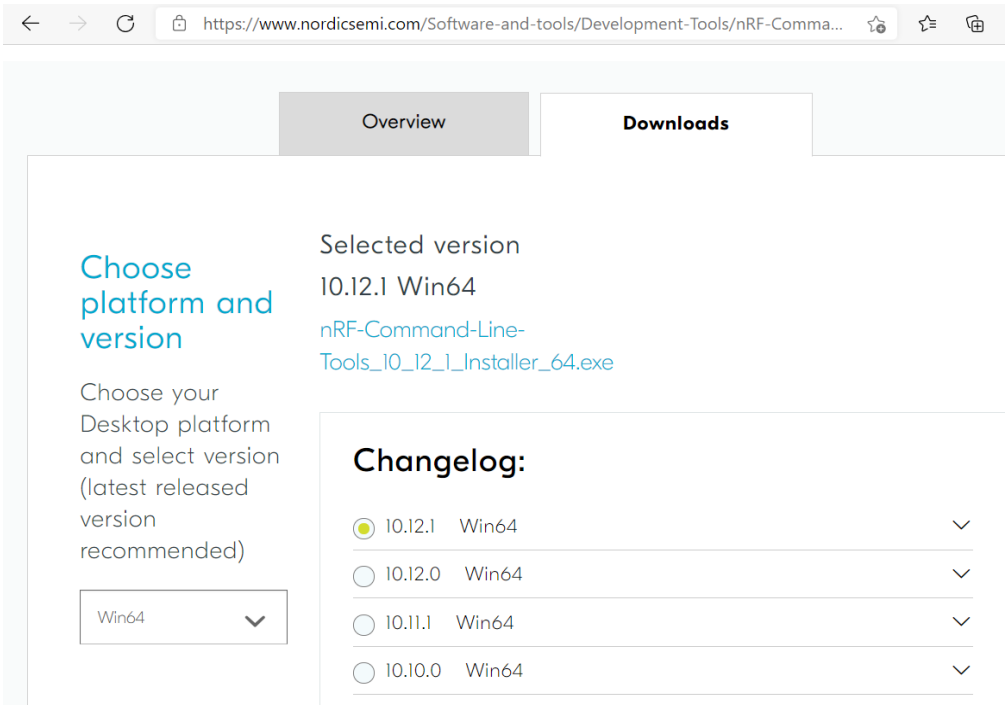


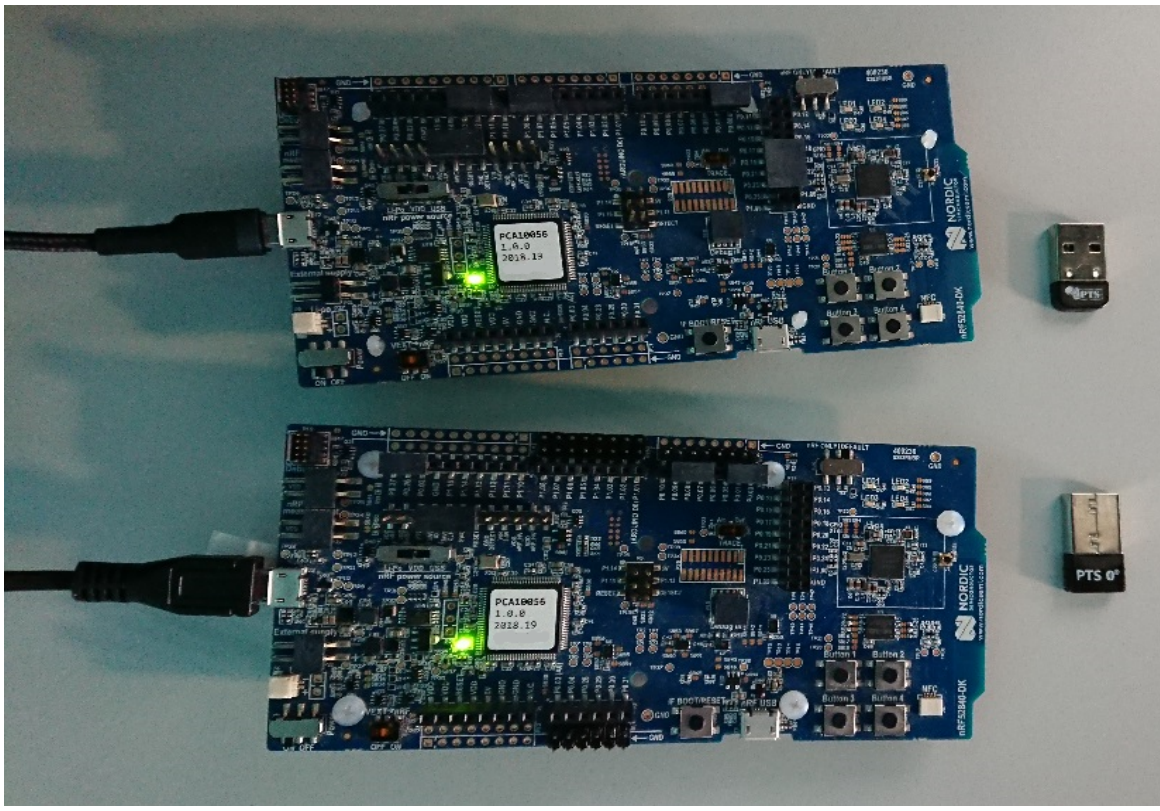
Note

Starting with PTS 8.0.1 the Bluetooth Protocol Viewer is no longer included. So to capture Bluetooth events, you have to download it separately.

Setup Zephyr project for Windows Perform Windows setup from Getting Started Guide.

Install nrftools On Windows download latest nrftools (version >= 10.12.1) from site https://www.nordicsemi.com/Software-and-tools/Development-Tools/nRF-Command-Line-Tools/Download and run default install.

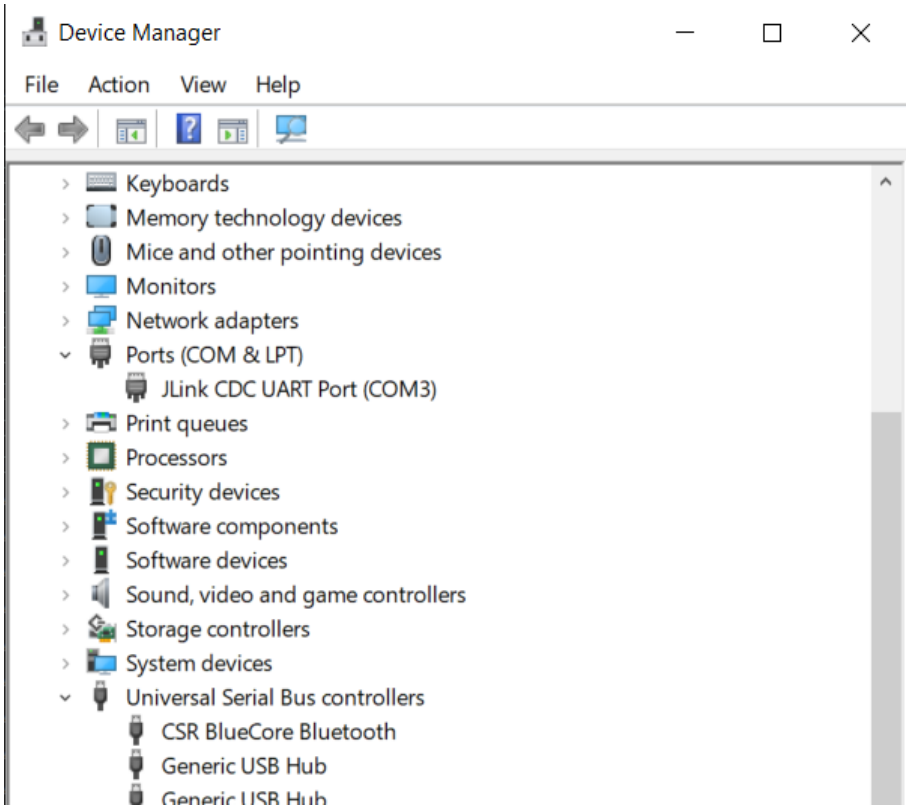




Connect devices



Flash board In Device Manager find COM port of your nrf board. In my case it is COM3.



In Git Bash, go to zephyrproject

```
cd ~/zephyrproject
```

Build the auto-pts tester app

```
west build -p auto -b nrf52840dk/nrf52840 zephyr/tests/bluetooth/tester/
```

You can display flashing options with:

```
west flash --help
```

and flash board with built earlier elf file:

```
west flash --skip-rebuild --board-dir /dev/ttyS2 --elf-file ~/zephyrproject/build/zephyr/
↪zephyr.elf
```

Note that west does not accept COMs, so use /dev/ttyS2 as the COM3 equivalent, /dev/ttyS2 as the COM3 equivalent, etc.(/dev/ttyS + decremented COM number).

Setup auto-pts project In Git Bash, clone project repo:

```
git clone https://github.com/intel/auto-pts.git
```

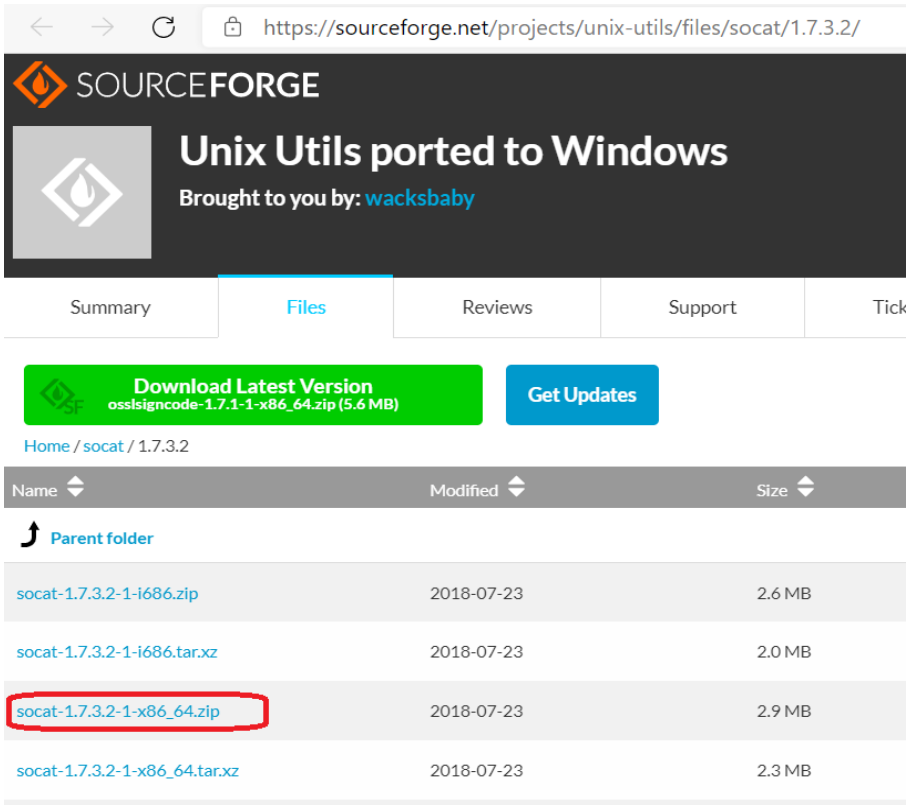
Go into the project folder:

```
cd auto-pts
```

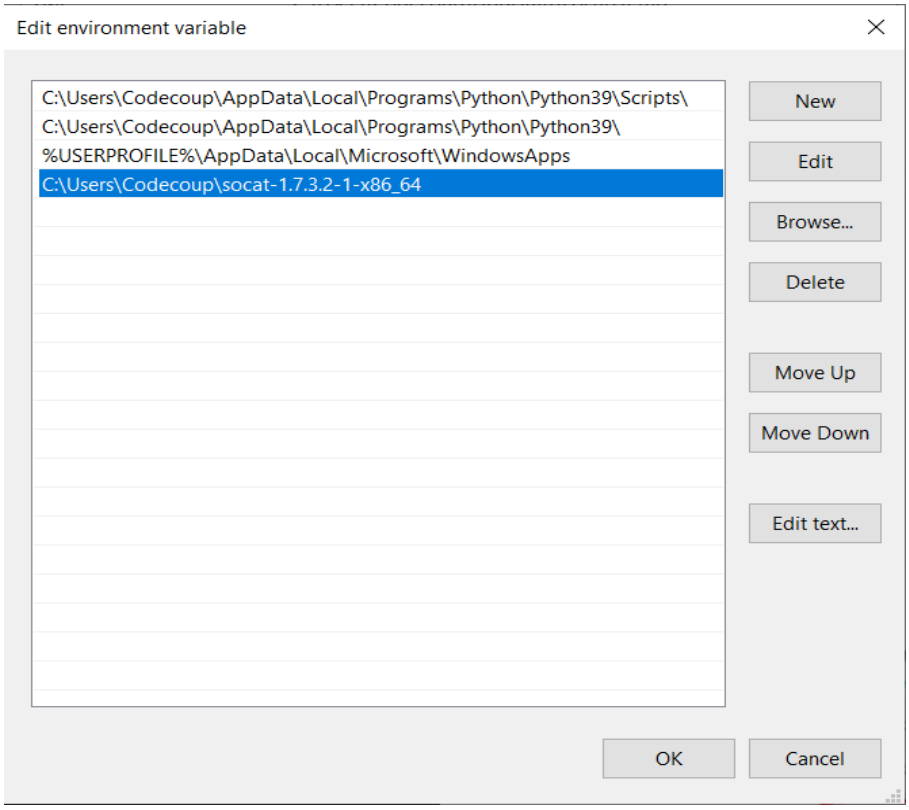
Install required python modules:

```
pip3 install --user wheel
pip3 install --user -r autoptsserver_requirements.txt
pip3 install --user -r autoptsclient_requirements.txt
```

Install socat.exe Download and extract socat.exe from https://sourceforge.net/projects/unix-utils/files/socat/1.7.3.2/ into folder ~/socat-1.7.3.2-1-x86_64/.

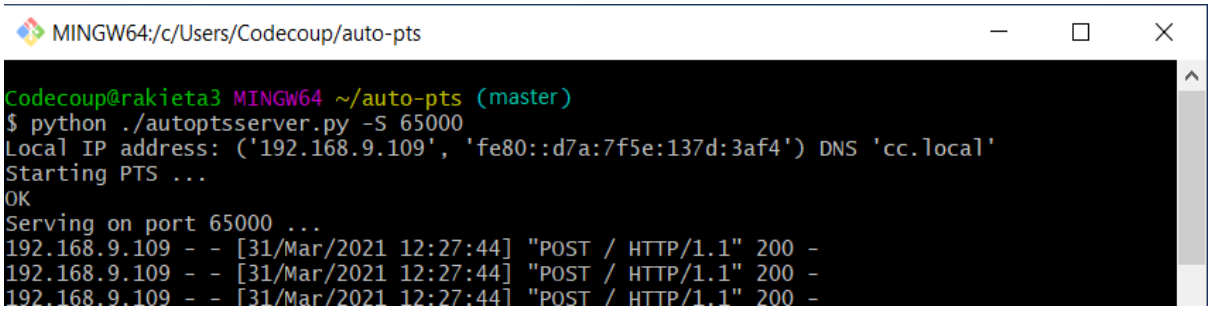


Add path to directory of socat.exe to PATH:



Running AutoPTS Server and client by default will run on localhost address. Run server:

```
python ./autoptsserver.py -S 65000
```



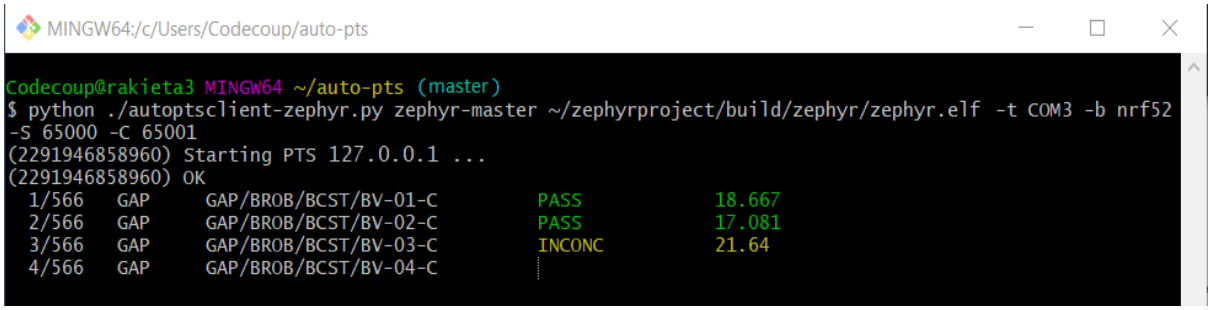
Note

If the error "ImportError: No module named pywintypes" appeared after the fresh setup, uninstall and install the pywin32 module:

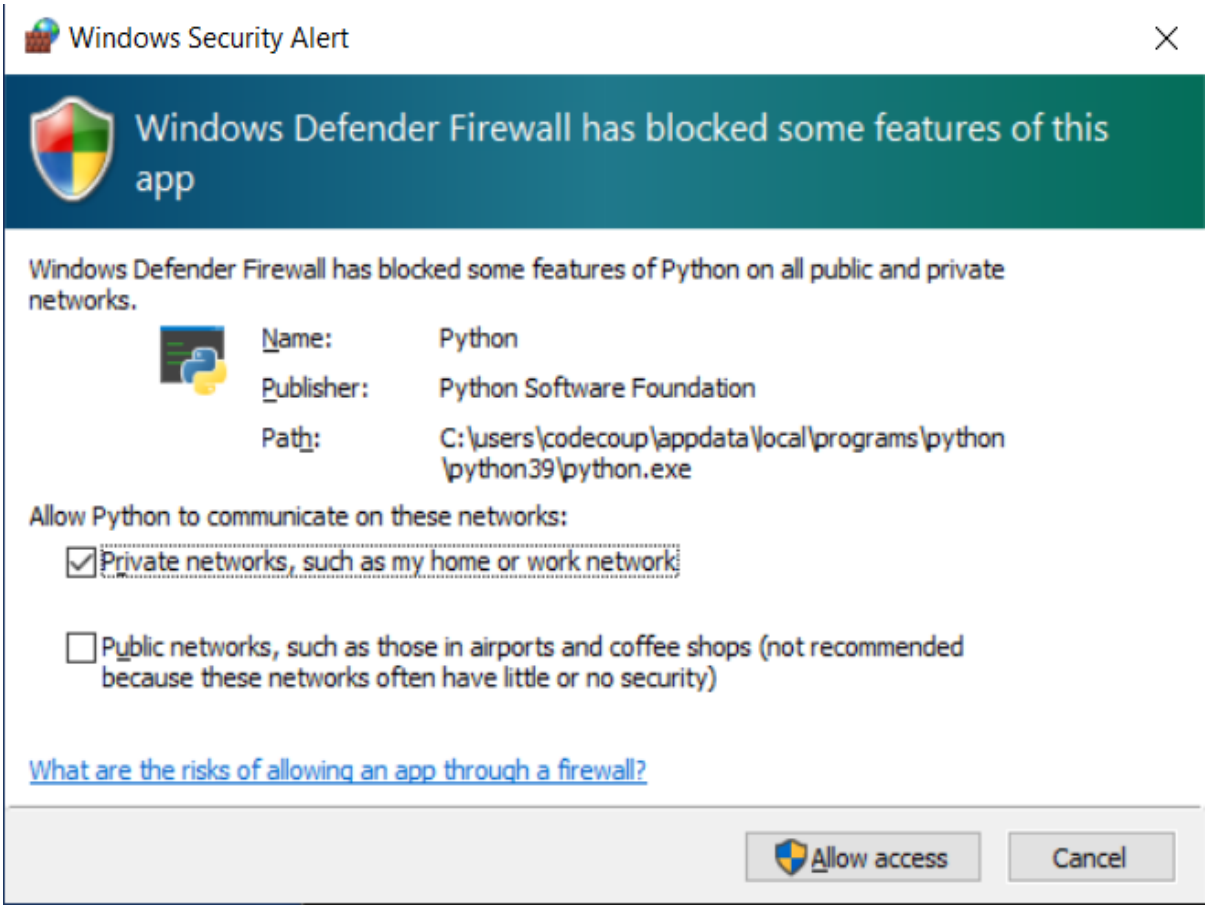
```
pip install --upgrade --force-reinstall pywin32
```

Run client:

```
python ./autoptsclient-zephyr.py zephyr-master ~/zephyrproject/build/zephyr/zephyr.elf -t␣
↪COM3 -b nrf52 -S 65000 -C 65001
```



At the first run, when Windows asks, enable connection through firewall:



Troubleshooting

- "When running actual hardware test mode, I have only BTP TIMEOUTs."

This is a problem with connection between auto-pts client and board. There are many possible causes. Try:

- Clean your auto-pts and zephyr repos with

Warning

This command will force the irreversible removal of all uncommitted files in the repo.

```
git clean -fdx
```

then build and flash tester elf again.

- If you have set up Windows on virtual machine, check if guest extensions are installed properly or change USB compatibility mode in VM settings to USB 2.0.
- Check, if firewall in not blocking python.exe or socat.exe.
- Check if board sends ready event after restart (hex 00 00 80 ff 00 00). Open serial connection to board with e.g. PuTTy with proper COM and baud rate. After board reset you should see some strings in console.
- Check if socat.exe creates tunnel to board. Run in console

```
socat.exe -x -v tcp-listen:65123 /dev/ttyS2,raw,b115200
```

where /dev/ttyS2 is the COM3 equivalent. Open PuTTY, set connection type to Raw, IP to 127.0.0.1, port to 65123. After board reset you should see some strings in console.

AutoPTS on Linux This tutorial shows how to setup AutoPTS client on Linux with AutoPTS server running on Windows 10 virtual machine. Tested with Ubuntu 20.4 and Linux Mint 20.4.

You must have a Zephyr development environment set up. See *Getting Started Guide* for details.

Supported methods to test zephyr bluetooth host:

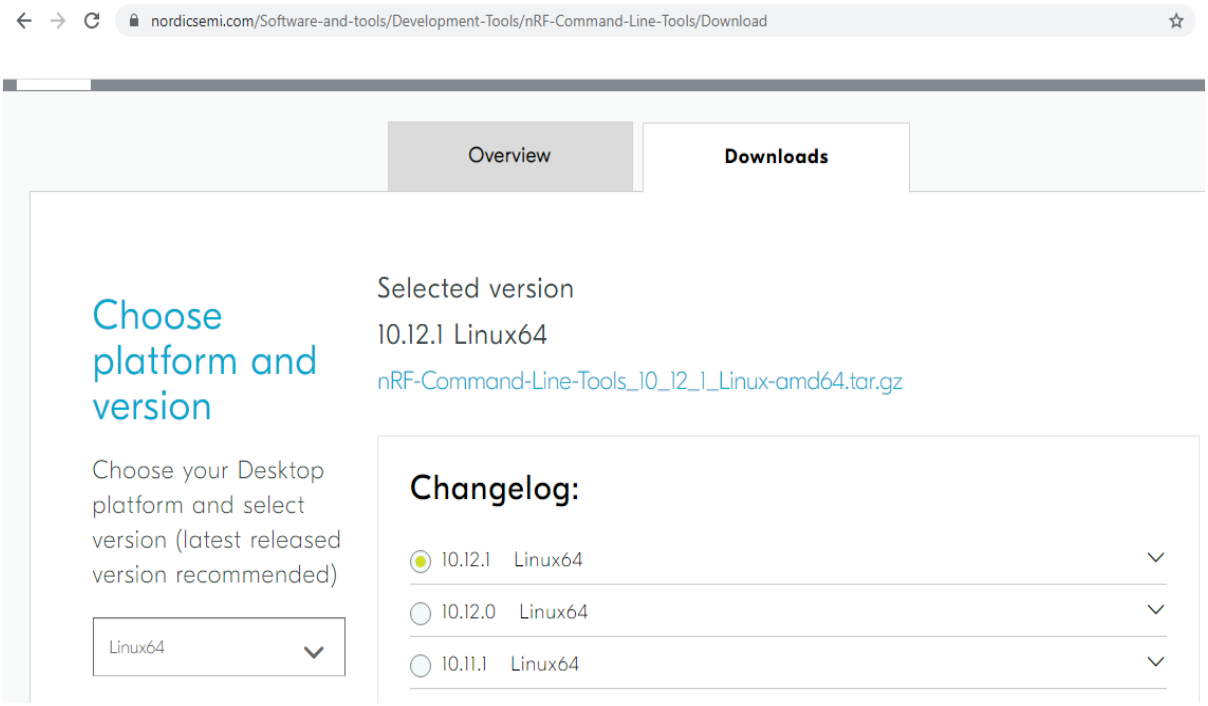
- Testing Zephyr Host Stack on QEMU
- Testing Zephyr Host Stack on native_sim
- Testing Zephyr combined (controller + host) build on Real hardware (such as nRF52)

For running with QEMU or native_sim, see *Running on QEMU or native_sim*.

- *Setup Linux*
- *Install nrftools (only required in the actual hardware test mode)*
- *Setup Windows 10 virtual machine*
  - *Update Windows*
  - *Setup static IP*
  - *Install Python 3*
  - *Install Git*
  - *Install PTS 8*
  - *Connect PTS dongle*
- *Connect devices (only required in the actual hardware test mode)*
- *Flash board (only required in the actual hardware test mode)*
- *Setup auto-pts project*
  - *AutoPTS client on Linux*
  - *Autopts server on Windows virtual machine*
- *Running AutoPTS*
- *Troubleshooting*

Setup Linux

Install nrftools (only required in the actual hardware test mode) Download latest nrftools (version >= 10.12.1) from site https://www.nordicsemi.com/Software-and-tools/Development-Tools/nRF-Command-Line-Tools/Download.



After you extract archive, you will see 2 .deb files, e.g.:

- JLink_Linux_V688a_x86_64.deb
- nRF-Command-Line-Tools_10_12_1_Linux-amd64.deb

and README.md. To install the tools, double click on each .deb file or follow instructions from README.md.

Setup Windows 10 virtual machine Choose and install your hypervisor like VMWare Workstation(preferred) or VirtualBox. On VirtualBox could be some issues, if your host has fewer than 6 CPU.

Create Windows virtual machine instance. Make sure it has at least 2 cores and installed guest extensions.

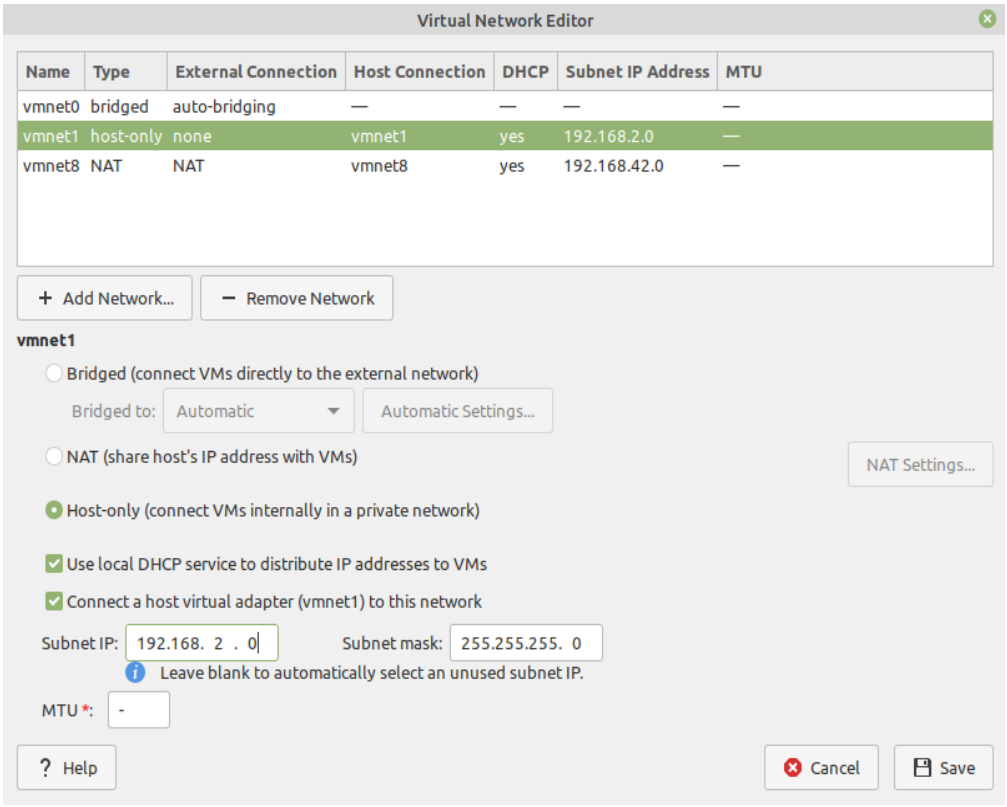
Setup tested with VirtualBox 6.1.18 and VMWare Workstation 16.1.1 Pro.

Update Windows Update Windows in:

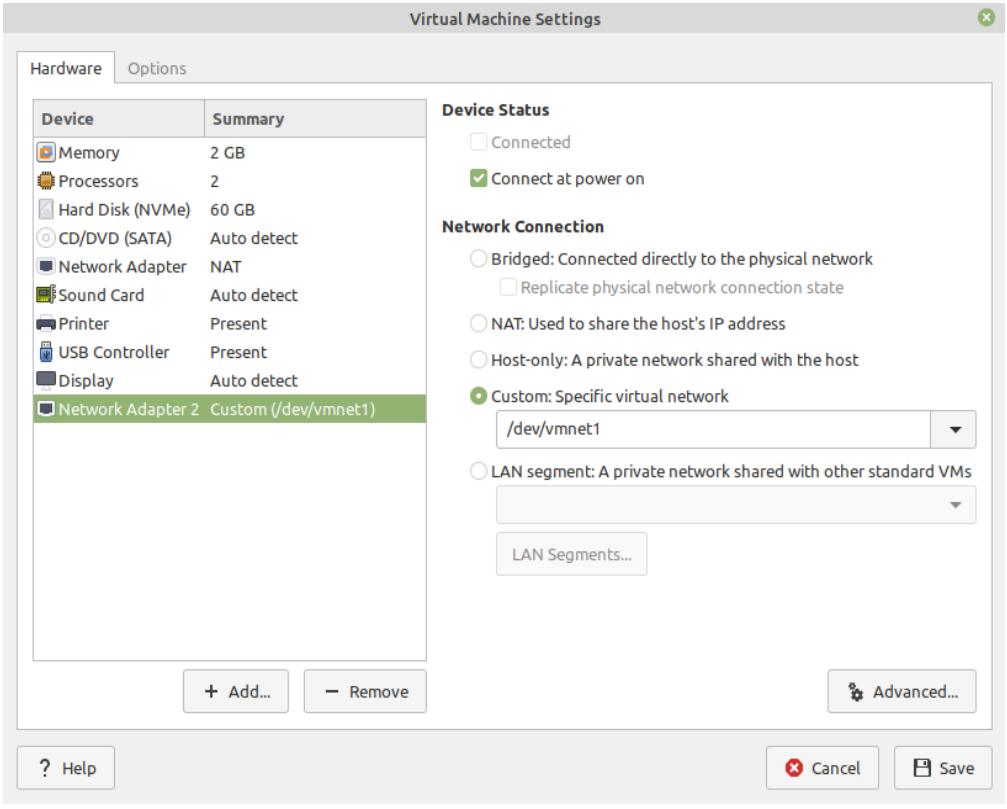
Start -> Settings -> Update & Security -> Windows Update

Setup static IP

WMWare Works On Linux, open Virtual Network Editor app and create network:



Open virtual machine network settings. Add custom adapter:



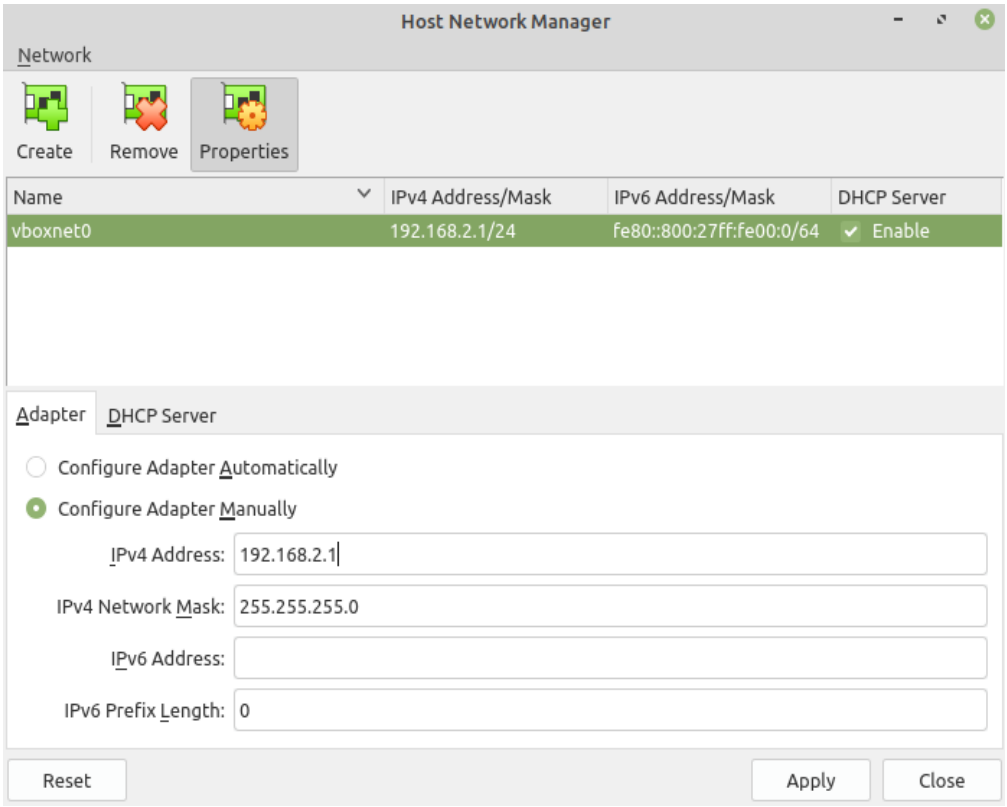
If you type 'ifconfig' in terminal, you should be able to find your host IP:

```
vmnet1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.2.1  netmask 255.255.255.0  broadcast 192.168.2.255
        inet6 fe80::250:56ff:fec0:1  prefixlen 64  scopeid 0x20<link>
        ether 00:50:56:c0:00:01  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 42  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

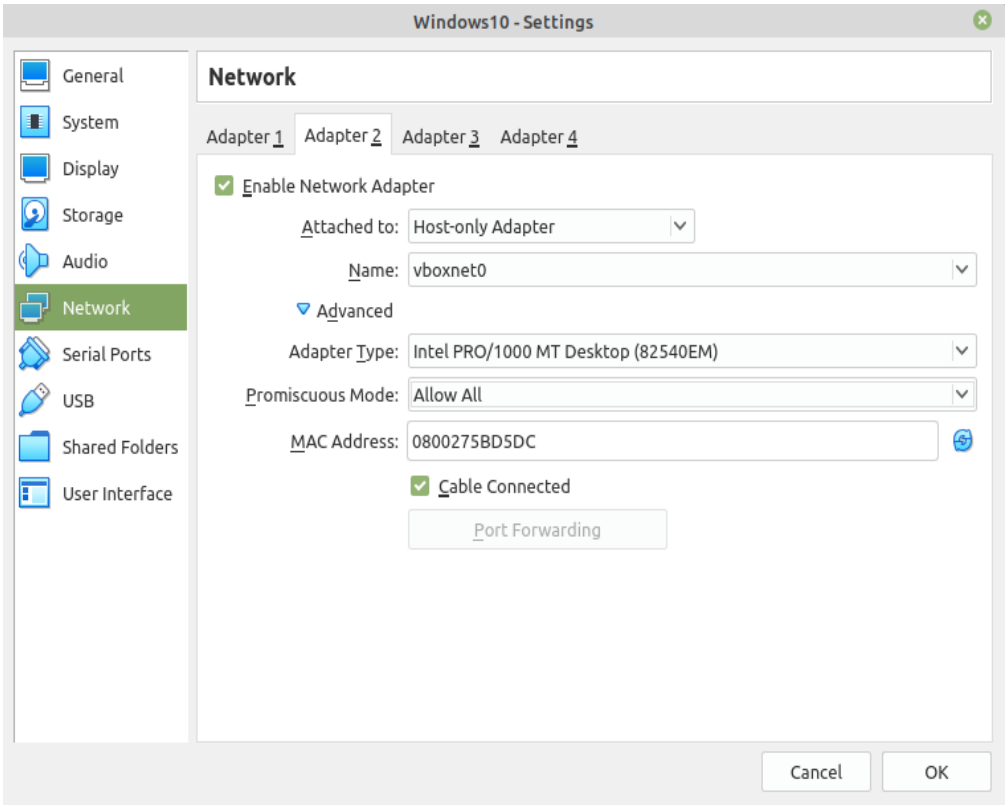
VirtualBox Go to:

File -> Host Network Manager

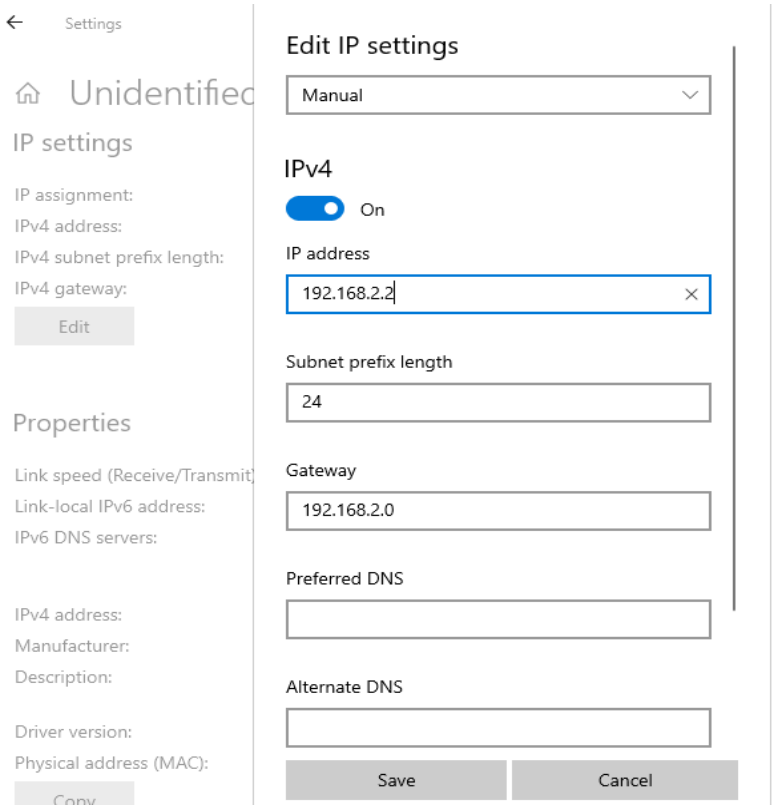
and create network:



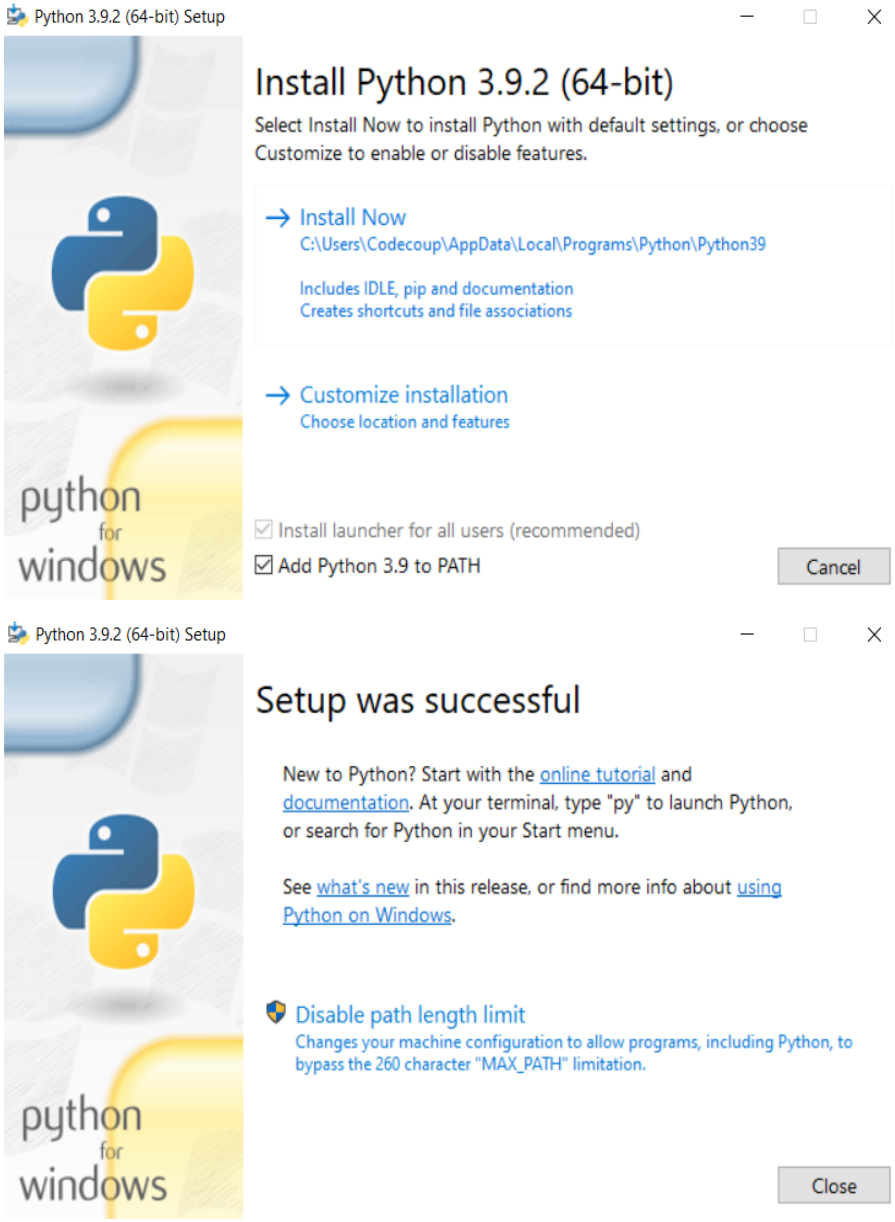
Open virtual machine network settings. On adapter 1 you will have created by default NAT. Add adapter 2:



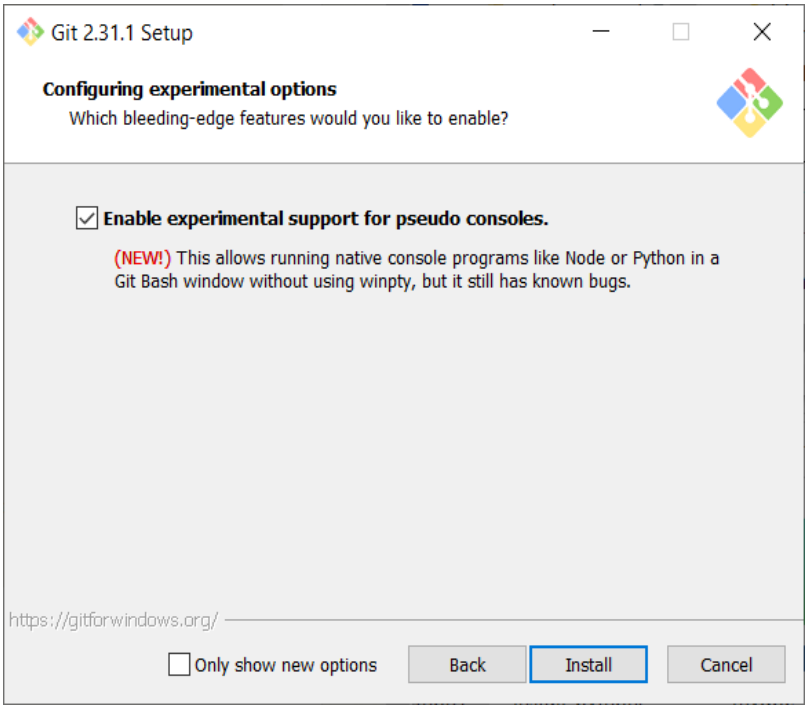
Windows Setup static IP on Windows virtual machine. Go to Settings -> Network & Internet -> Ethernet -> Unidentified network -> Edit and set:



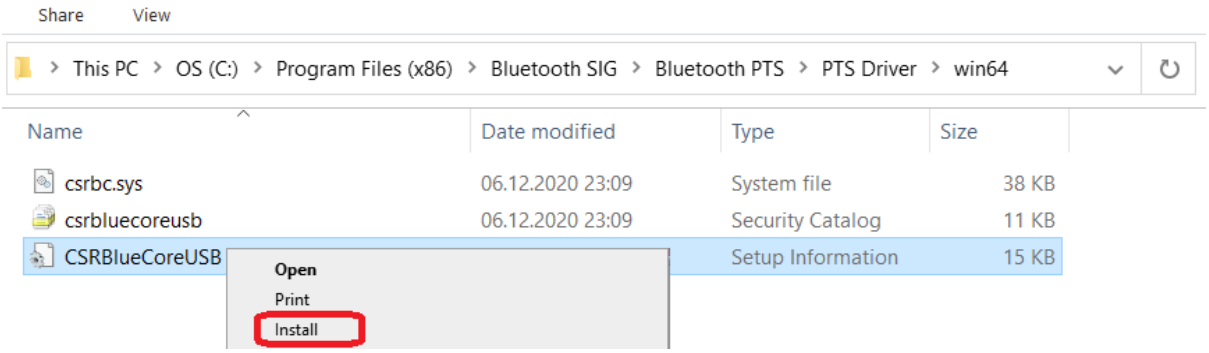
Install Python 3 Download and install latest Python 3 on Windows. Let the installer add the Python installation directory to the PATH and disable the path length limitation.



Install Git Download and install Git. During installation enable option: Enable experimental support for pseudo consoles. We will use Git Bash as Windows terminal.



Install PTS 8 On Windows virtual machine, install latest PTS from https://www.bluetooth.org. Remember to install drivers from installation directory "C:/Program Files (x86)/Bluetooth SIG/Bluetooth PTS/PTS Driver/win64/CSRBlueCoreUSB.inf"



Note

Starting with PTS 8.0.1 the Bluetooth Protocol Viewer is no longer included. So to capture Bluetooth events, you have to download it separately.

Connect PTS dongle With VirtualBox there should be no problem. Just find dongle in Devices -> USB and connect.

With VMWare you might need to use some trick, if you cannot find dongle in VM -> Removable Devices. Type in Linux terminal:

```
usb-devices
```

and find in output your PTS Bluetooth USB dongle

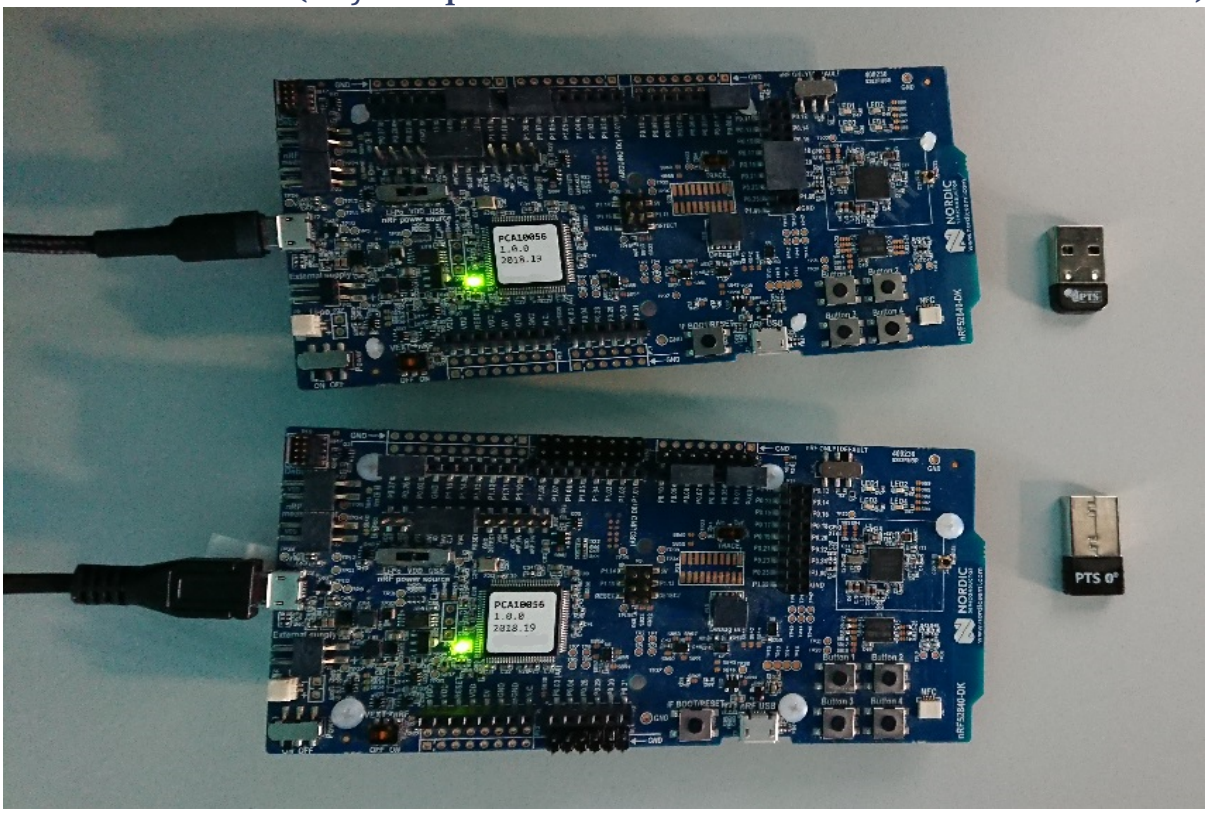
```
T:  Bus=01 Lev=01 Prnt=01 Port=01 Cnt=01 Dev#=  6 Spd=12  MxCh= 0
D:  Ver= 2.00 Cls=e0(wlcon) Sub=01 Prot=01 MxPS=64 #Cfgs=  1
P:  Vendor=0a12 ProdID=0001 Rev=25.20
S:  Product=CSR rck PTS Dongle
C:  #Ifs= 3 Cfg#= 1 Atr=c0 MxPwr=0mA
```

Note Vendor and ProdID number. Close VMWare Workstation and open .vmx of your virtual machine (path similar to /home/codecoup/vmware/Windows 10/Windows 10.vmx) in text editor. Write anywhere in the file following line:

```
usb.autoConnect.device0 = "0x0a12:0x0001"
```

just replace 0x0a12 with Vendor number and 0x0001 with ProdID number you found earlier.

Connect devices (only required in the actual hardware test mode)





Flash board (only required in the actual hardware test mode) On Linux, go to ~/zephyrproject. There should be already ~/zephyrproject/build directory. Flash board:

```
west flash
```

Setup auto-pts project

AutoPTS client on Linux Clone auto-pts project:

```
git clone https://github.com/intel/auto-pts.git
```

Install socat, that is used to transfer BTP data stream from UART's tty file:

```
sudo apt-get install python-setuptools socat
```

Install required python modules:

```
cd auto-pts
pip3 install --user wheel
pip3 install --user -r autoptsclient_requirements.txt
```

Autopts server on Windows virtual machine In Git Bash, clone auto-pts project repo:

```
git clone https://github.com/intel/auto-pts.git
```

Install required python modules:

```
cd auto-pts
pip3 install --user wheel
pip3 install --user -r autoptsserver_requirements.txt
```

Restart virtual machine.

Running AutoPTS Server and client by default will run on localhost address. Run server:

```
python ./autoptsserver.py
```

```
codecoup@DESKTOP-5SFTBGB MINGW64 ~/auto-pts (master)
$ python ./autoptsserver.py
Local IP address: ('10.0.2.15', 'fe80::ccb2:6ca1:6368:342a') DNS 'home'
Local IP address: ('192.168.2.2', 'fe80::fd12:15d5:2ddb:9147') DNS None
Starting PTS ...
OK
Serving on port 65000 ...
```

Testing Zephyr Host Stack on QEMU:

```
# A Bluetooth controller needs to be mounted.
# For running with HCI UART, please visit: https://docs.zephyrproject.org/latest/samples/
↪bluetooth/hci_uart/README.html#bluetooth-hci-uart

python ./autoptsclient-zephyr.py "C:\Users\USER_NAME\Documents\Profile Tuning Suite\PTS_
↪PROJECT\PTS_PROJECT.pqw6" \
    ~/zephyrproject/build/zephyr/zephyr.elf -i SERVER_IP -l LOCAL_IP
```

Testing Zephyr Host Stack on native_sim:

```
# A Bluetooth controller needs to be mounted.
# For running with HCI UART, please visit: https://docs.zephyrproject.org/latest/samples/
↪bluetooth/hci_uart/README.html#bluetooth-hci-uart

west build -b native_sim zephyr/tests/bluetooth/tester/ -DEXTRA_CONF_FILE=overlay-native.
↪conf

sudo python ./autoptsclient-zephyr.py "C:\Users\USER_NAME\Documents\Profile Tuning Suite\
↪PTS_PROJECT\PTS_PROJECT.pqw6" \
    ~/zephyrproject/build/zephyr/zephyr.exe -i SERVER_IP -l LOCAL_IP --hci 0
```

Testing Zephyr combined (controller + host) build on nRF52:

Note

If the error "ImportError: No module named pywintypes" appeared after the fresh setup, uninstall and install the pywin32 module:

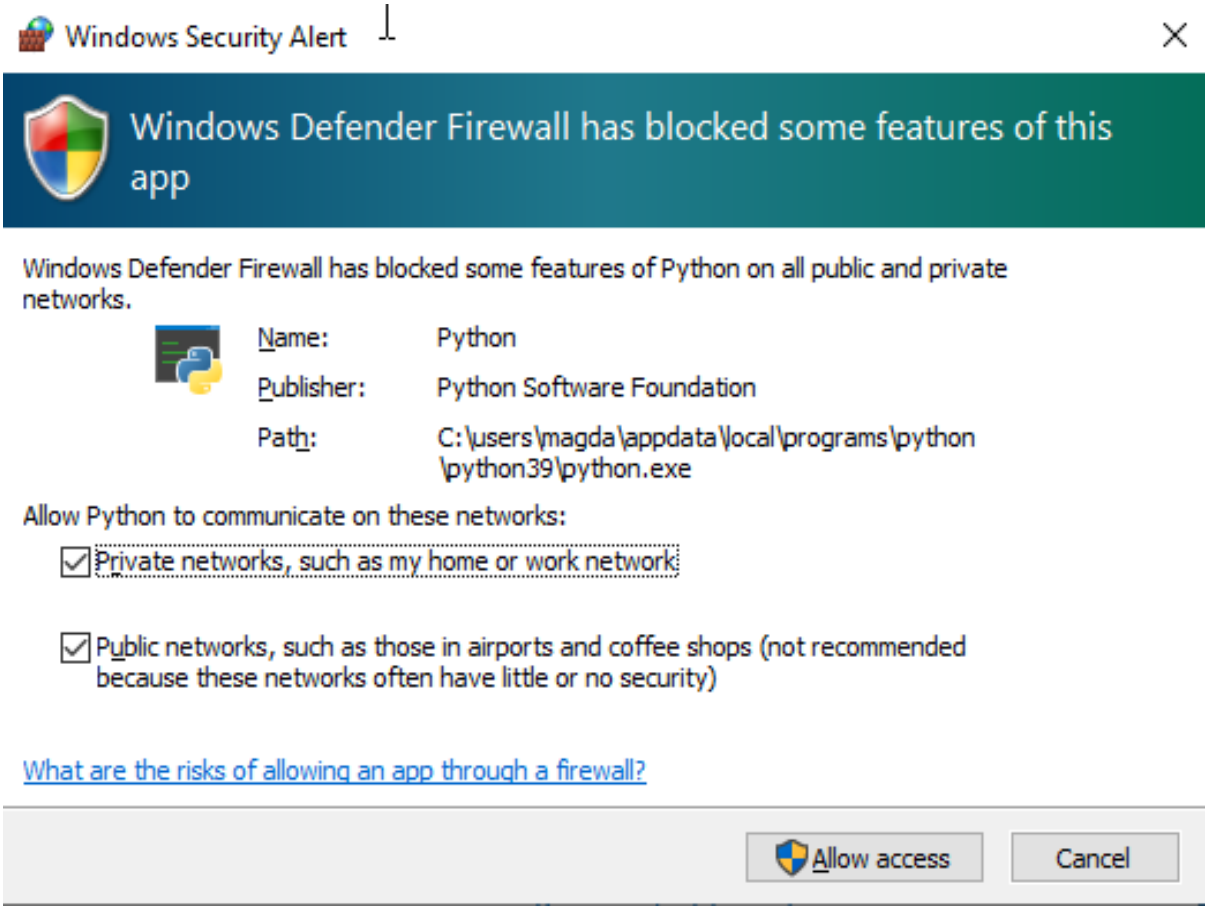
```
pip install --upgrade --force-reinstall pywin32
```

Run client:

```
python ./autoptsclient-zephyr.py zephyr-master ~/zephyrproject/build/zephyr/zephyr.elf -t /
↪dev/ACM0 \
    -b nrf52 -l 192.168.2.1 -i 192.168.2.2
```

```
codecoup@:~/auto-pts$ ./autoptsclient-zephyr.py zephyr-master ~/zephyrproject/build/zephyr/zephyr.elf
-t /dev/ttyACM0 -b nrf52 -l 192.168.2.1 -i 192.168.2.2
(140092026955280) Starting PTS 192.168.2.2 ...
(140092026955280) OK
  1/629   DIS     DIS/SR/SD/BV-01-C             PASS            23.477
  2/629   DIS     DIS/SR/DEC/BV-01-C            PASS            22.659
```

At the first run, when Windows asks, enable connection through firewall:



Troubleshooting

- "After running one test, I need to restart my Windows virtual machine to run another, because of fail verdict from APICOM in PTS logs."

It means your virtual machine has not enough processor cores or memory. Try to add more in settings. Note that a host with 4 CPUs could be not enough with VirtualBox as hypervisor. In this

case, choose rather VMWare Workstation.

- "I cannot start autoptsserver-zephyr.py. I always got error:"

```
codecoup@DESKTOP-5SFTBGB MINGW64 ~/auto-pts (master)
$ python ./autoptsserver.py
Local IP address: ('10.0.2.15', 'fe80::ccb2:6ca1:6368:342a') DNS 'home'
Local IP address: ('192.168.2.2', 'fe80::fd12:15d5:2ddb:9147') DNS None
Starting PTS ...
Traceback (most recent call last):
  File "C:\Users\codecoup\auto-pts\autoptsserver.py", line 127, in <module>
    main()
  File "C:\Users\codecoup\auto-pts\autoptsserver.py", line 115, in main
    pts = PyPTSWithXmlRpcCallback()
  File "C:\Users\codecoup\auto-pts\autoptsserver.py", line 52, in __init__
    ptscontrol.PyPTS.__init__(self)
  File "C:\Users\codecoup\auto-pts\ptscontrol.py", line 267, in __init__
    self.restart_pts()
  File "C:\Users\codecoup\auto-pts\ptscontrol.py", line 391, in restart_pts
    self.start_pts()
  File "C:\Users\codecoup\auto-pts\ptscontrol.py", line 440, in start_pts
    log("PTS Bluetooth Address: %s", self.get_bluetooth_address())
  File "C:\Users\codecoup\auto-pts\ptscontrol.py", line 762, in get_bluetooth_address
    raise e
  File "C:\Users\codecoup\auto-pts\ptscontrol.py", line 759, in get_bluetooth_address
    address = self._pts.GetPTSBluetoothAddress()
  File "<COMObject ProfileTuningSuite_6.PTSControlServer>", line 3, in GetPTSBluetoothAddress
pywintypes.com_error: (-2147352567, 'Exception occurred.', (0, 'PTS', '(HRESULT:0x849C0043) Error HResult!', None
, 0, -2070151101), None)
```

One or more of the following steps should help:

- Close all PTS Windows.
- Replug PTS bluetooth dongle.
- Delete temporary workspace. You will find it in auto-pts-code/workspaces/zephyr/zephyr-master/ as temp_zephyr-master. Be careful, do not remove the original one zephyr-master.pqw6.
- Restart Windows virtual machine.

ICS Features

The Zephyr ICS file for the Host features can be downloaded here: `ICS_Zephyr_Bluetooth_Host.pts`.

Use the Bluetooth Qualification website to view and edit the ICS.

6.1.3 Stack Architecture

Overview

This page describes the software architecture of Zephyr's Bluetooth protocol stack.

> **Note**
>
> Zephyr supports mainly Bluetooth Low Energy (BLE), the low-power version of the Bluetooth specification. Zephyr also has limited support for portions of the BR/EDR Host. Throughout this architecture document we use BLE interchangeably for Bluetooth except when noted.

BLE Layers There are 3 main layers that together constitute a full Bluetooth Low Energy protocol stack:

- **Host**: This layer sits right below the application, and is comprised of multiple (non real-time) network and transport protocols enabling applications to communicate with peer devices in a standard and interoperable way.
- **Controller**: The Controller implements the Link Layer (LE LL), the low-level, real-time protocol which provides, in conjunction with the Radio Hardware, standard-interoperable over-the-air communication. The LL schedules packet reception and transmission, guarantees the delivery of data, and handles all the LL control procedures.
- **Radio Hardware**: Hardware implements the required analog and digital baseband functional blocks that permit the Link Layer firmware to send and receive in the 2.4GHz band of the spectrum.

Host Controller Interface The Bluetooth Specification describes the format in which a Host must communicate with a Controller. This is called the Host Controller Interface (HCI) protocol. HCI can be implemented over a range of different physical transports like UART, SPI, or USB. This protocol defines the commands that a Host can send to a Controller and the events that it can expect in return, and also the format for user and protocol data that needs to go over the air. The HCI ensures that different Host and Controller implementations can communicate in a standard way making it possible to combine Hosts and Controllers from different vendors.

Configurations The three separate layers of the protocol and the standardized interface make it possible to implement the Host and Controller on different platforms. The two following configurations are commonly used:

- **Single-chip configuration**: In this configuration, a single microcontroller implements all three layers and the application itself. This can also be called a system-on-chip (SoC) implementation. In this case the BLE Host and the BLE Controller communicate directly through function calls and queues in RAM. The Bluetooth specification does not specify how HCI is implemented in this single-chip configuration and so how HCI commands, events, and data flows between the two can be implementation-specific. This configuration is well suited for those applications and designs that require a small footprint and the lowest possible power consumption, since everything runs on a single IC.
- **Dual-chip configuration**: This configuration uses two separate ICs, one running the Application and the Host, and a second one with the Controller and the Radio Hardware. This is sometimes also called a connectivity-chip configuration. This configuration allows for a wider variety of combinations of Hosts when using the Zephyr OS as a Controller. Since HCI ensures interoperability among Host and Controller implementations, including of course Zephyr's very own BLE Host and Controller, users of the Zephyr Controller can choose to use whatever Host running on any platform they prefer. For example, the host can be the Linux BLE Host stack (BlueZ) running on any processor capable of supporting Linux. The Host processor may of course also run Zephyr and the Zephyr OS BLE Host. Conversely, combining an IC running the Zephyr Host with an external Controller that does not run Zephyr is also supported.

Build Types The Zephyr software stack as an RTOS is highly configurable, and in particular, the BLE subsystem can be configured in multiple ways during the build process to include only the features and layers that are required to reduce RAM and ROM footprint as well as power consumption. Here's a short list of the different BLE-enabled builds that can be produced from the Zephyr project codebase:

- **Controller-only build**: When built as a BLE Controller, Zephyr includes the Link Layer and a special application. This application is different depending on the physical transport chosen for HCI:
  - hci_uart
  - hci_usb

  - hci_spi

  This application acts as a bridge between the UART, SPI or USB peripherals and the Controller subsystem, listening for HCI commands, sending application data and responding with events and received data. A build of this type sets the following Kconfig option values:

  - `CONFIG_BT=y`
  - `CONFIG_BT_HCI=y`
  - `CONFIG_BT_HCI_RAW=y`
  - `CONFIG_BT_CTLR=y`
  - `CONFIG_BT_LL_SW_SPLIT=y` (if using the open source Link Layer)

- **Host-only build**: A Zephyr OS Host build will contain the Application and the BLE Host, along with an HCI driver (UART or SPI) to interface with an external Controller chip. A build of this type sets the following Kconfig option values:

  - `CONFIG_BT=y`
  - `CONFIG_BT_HCI=y`
  - `CONFIG_BT_CTLR=n`

  All of the samples located in `samples/bluetooth` except for the ones used for Controller-only builds can be built as Host-only

- **Combined build**: This includes the Application, the Host and the Controller, and it is used exclusively for single-chip (SoC) configurations. A build of this type sets the following Kconfig option values:

  - `CONFIG_BT=y`
  - `CONFIG_BT_HCI=y`
  - `CONFIG_BT_CTLR=y`
  - `CONFIG_BT_LL_SW_SPLIT=y` (if using the open source Link Layer)

  All of the samples located in `samples/bluetooth` except for the ones used for Controller-only builds can be built as Combined

The picture below shows the SoC or single-chip configuration when using a Zephyr combined build (a build that includes both a BLE Host and a Controller in the same firmware image that is programmed onto the chip):

When using connectivity or dual-chip configurations, several Host and Controller combinations are possible, some of which are depicted below:

When using a Zephyr Host (left side of image), two instances of Zephyr OS must be built with different configurations, yielding two separate images that must be programmed into each of the chips respectively. The Host build image contains the application, the BLE Host and the selected HCI driver (UART or SPI), while the Controller build runs either the hci_uart, or the hci_spi app to provide an interface to the BLE Controller.

This configuration is not limited to using a Zephyr OS Host, as the right side of the image shows. One can indeed take one of the many existing GNU/Linux distributions, most of which include Linux's own BLE Host (BlueZ), to connect it via UART or USB to one or more instances of the Zephyr OS Controller build. BlueZ as a Host supports multiple Controllers simultaneously for applications that require more than one BLE radio operating at the same time but sharing the same Host stack.

Source tree layout

The stack is split up as follows in the source tree:

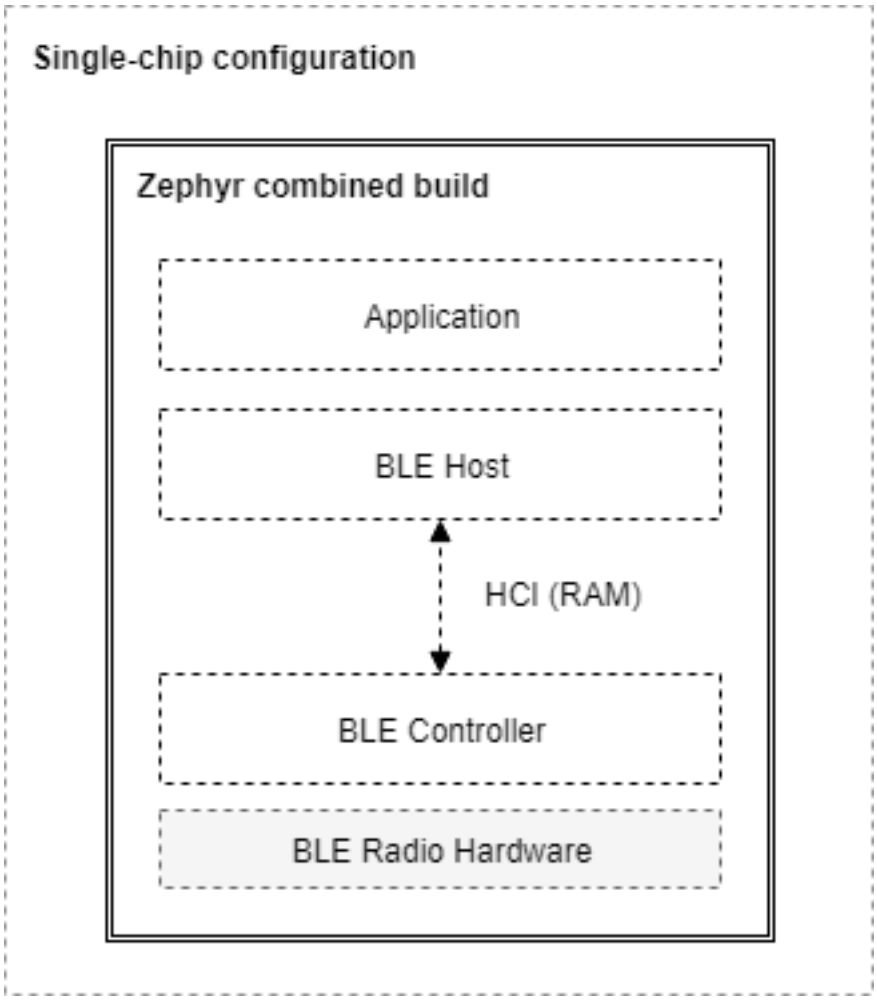


Fig. 1: A Combined build on a Single-Chip configuration

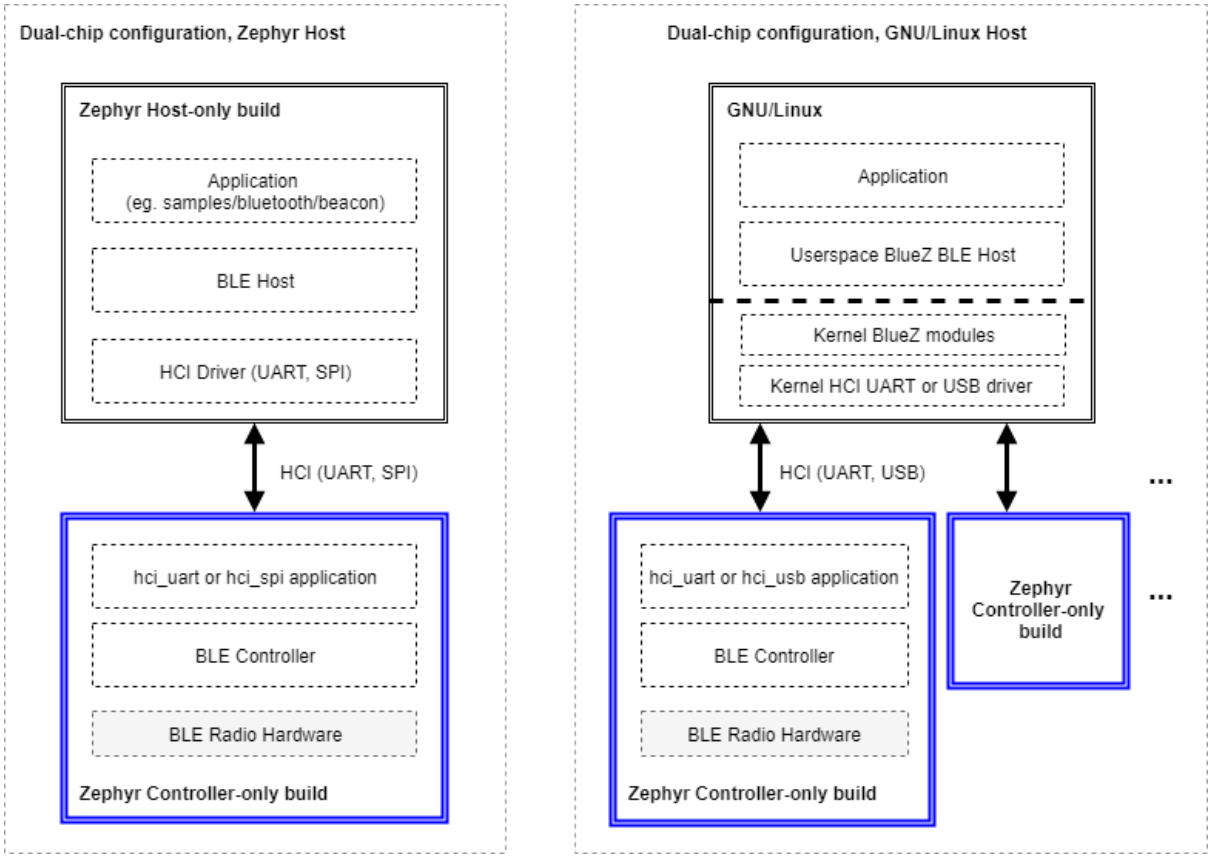


Fig. 2: Host-only and Controller-only builds on dual-chip configurations

`subsys/bluetooth/host`
: *The host stack*. This is where the HCI command and event handling as well as connection tracking happens. The implementation of the core protocols such as L2CAP, ATT, and SMP is also here.

`subsys/bluetooth/controller`
: *Bluetooth LE Controller* implementation. Implements the controller-side of HCI, the Link Layer as well as access to the radio transceiver.

`include/bluetooth/`
: *Public API* header files. These are the header files applications need to include in order to use Bluetooth functionality.

`drivers/bluetooth/`
: HCI transport drivers. Every HCI transport needs its own driver. For example, the two common types of UART transport protocols (3-Wire and 5-Wire) have their own drivers.

`samples/bluetooth/`
: Sample Bluetooth code. This is a good reference to get started with Bluetooth application development.

`tests/bluetooth/`
: Test applications. These applications are used to verify the functionality of the Bluetooth stack, but are not necessary the best source for sample code (see `samples/bluetooth` instead).

`doc/connectivity/bluetooth/`
: Extra documentation, such as PICS documents.

6.1.4 LE Host

The Bluetooth Host implements all the higher-level protocols and profiles, and most importantly, provides a high-level API for applications. The following diagram depicts the main protocol & profile layers of the host.

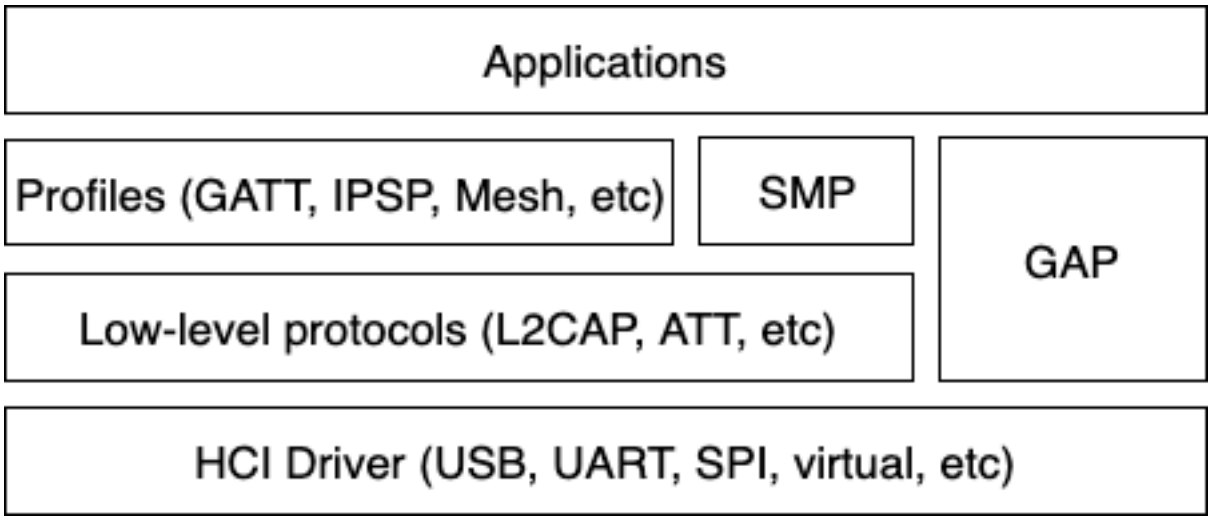


Fig. 3: Bluetooth Host protocol & profile layers.

Lowest down in the host stack sits a so-called HCI driver, which is responsible for abstracting away the details of the HCI transport. It provides a basic API for delivering data from the controller to the host, and vice-versa.

Perhaps the most important block above the HCI handling is the Generic Access Profile (GAP). GAP simplifies Bluetooth LE access by defining four distinct roles of BLE usage:

- Connection-oriented roles
  - Peripheral (e.g. a smart sensor, often with a limited user interface)
  - Central (typically a mobile phone or a PC)
- Connection-less roles
  - Broadcaster (sending out BLE advertisements, e.g. a smart beacon)
  - Observer (scanning for BLE advertisements)

Each role comes with its own build-time configuration option: `CONFIG_BT_PERIPHERAL`, `CONFIG_BT_CENTRAL`, `CONFIG_BT_BROADCASTER` & `CONFIG_BT_OBSERVER`. Of the connection-oriented roles central implicitly enables observer role, and peripheral implicitly enables broadcaster role. Usually the first step when creating an application is to decide which roles are needed and go from there. Bluetooth Mesh is a slightly special case, requiring at least the observer and broadcaster roles, and possibly also the Peripheral role. This will be described in more detail in a later section.

Peripheral role

Most Zephyr-based BLE devices will most likely be peripheral-role devices. This means that they perform connectable advertising and expose one or more GATT services. After registering services using the *`bt_gatt_service_register()`* API the application will typically start connectable advertising using the *`bt_le_adv_start()`* API.

There are several peripheral sample applications available in the tree, such as samples/bluetooth/peripheral_hr.

Central role

Central role may not be as common for Zephyr-based devices as peripheral role, but it is still a plausible one and equally well supported in Zephyr. Rather than accepting connections from other devices a central role device will scan for available peripheral device and choose one to connect to. Once connected, a central will typically act as a GATT client, first performing discovery of available services and then accessing one or more supported services.

To initially discover a device to connect to the application will likely use the *bt_le_scan_start()* API, wait for an appropriate device to be found (using the scan callback), stop scanning using *bt_le_scan_stop()* and then connect to the device using *bt_conn_le_create()*. If the central wants to keep automatically reconnecting to the peripheral it should use the *bt_le_set_auto_conn()* API.

There are some sample applications for the central role available in the tree, such as samples/bluetooth/central_hr.

Observer role

An observer role device will use the *bt_le_scan_start()* API to scan for device, but it will not connect to any of them. Instead it will simply utilize the advertising data of found devices, combining it optionally with the received signal strength (RSSI).

Broadcaster role

A broadcaster role device will use the *bt_le_adv_start()* API to advertise specific advertising data, but the type of advertising will be non-connectable, i.e. other device will not be able to connect to it.

Connections

Connection handling and the related APIs can be found in the *Connection Management* section.

Security

To achieve a secure relationship between two Bluetooth devices a process called pairing is used. This process can either be triggered implicitly through the security properties of GATT services, or explicitly using the `bt_conn_security()` API on a connection object.

To achieve a higher security level, and protect against Man-In-The-Middle (MITM) attacks, it is recommended to use some out-of-band channel during the pairing. If the devices have a sufficient user interface this "channel" is the user itself. The capabilities of the device are registered using the *bt_conn_auth_cb_register()* API. The *bt_conn_auth_cb* struct that's passed to this API has a set of optional callbacks that can be used during the pairing - if the device lacks some feature the corresponding callback may be set to NULL. For example, if the device does not have an input method but does have a display, the `passkey_entry` and `passkey_confirm` callbacks would be set to NULL, but the `passkey_display` would be set to a callback capable of displaying a passkey to the user.

Depending on the local and remote security requirements & capabilities, there are four possible security levels that can be reached:

BT_SECURITY_L1
: No encryption and no authentication.

BT_SECURITY_L2
: Encryption but no authentication (no MITM protection).

BT_SECURITY_L3
: Encryption and authentication using the legacy pairing method from Bluetooth 4.0 and 4.1.

BT_SECURITY_L4
: Encryption and authentication using the LE Secure Connections feature available since Bluetooth 4.2.

Note

Mesh has its own security solution through a process called provisioning. It follows a similar procedure as pairing, but is done using separate mesh-specific APIs.

L2CAP

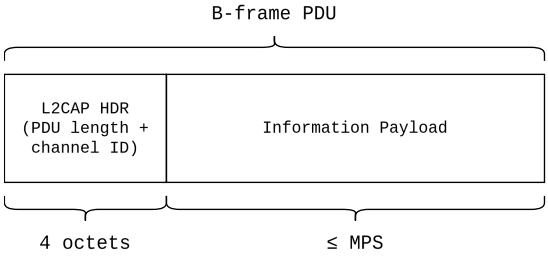
L2CAP stands for the Logical Link Control and Adaptation Protocol. It is a common layer for all communication over Bluetooth connections, however an application comes in direct contact with it only when using it in the so-called Connection-oriented Channels (CoC) mode. More information on this can be found in the *L2CAP API section*.

Terminology The definitions are from the Core Specification version 5.4, volume 3, part A 1.4.

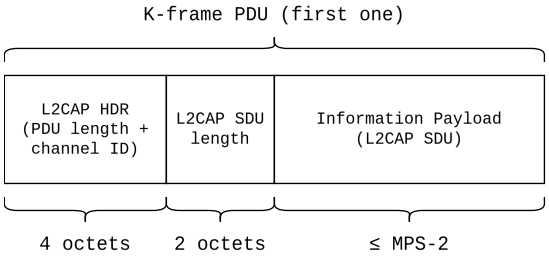
Term	Description
Upper layer	Layer above L2CAP, it exchanges data in form of SDUs. It may be an application or a higher level protocol.
Lower layer	Layer below L2CAP, it exchanges data in form of PDUs (or fragments). It is usually the HCI.
Service Data Unit (SDU)	Packet of data that L2CAP exchanges with the upper layer. This term is relevant only in Enhanced Retransmission mode, Streaming mode, Retransmission mode and Flow Control Mode, not in Basic L2CAP mode.
Protocol Data Unit (PDU)	Packet of data containing L2CAP data. PDUs always start with Basic L2CAP header. Types of PDUs for LE: *B-frames* and *K-frames*. Types of PDUs for BR/EDR: I-frames, S-frames, C-frames and G-frames.
Maximum Transmission Unit (MTU)	Maximum size of an SDU that the upper layer is capable of accepting.
Maximum Payload Size (MPS)	Maximum payload size that the L2CAP layer is capable of accepting. In Basic L2CAP mode, the MTU size is equal to MPS. In credit-based channels without segmentation, the MTU is MPS minus 2.
Basic L2CAP header	Present at the beginning of each PDU. It contains two fields, the PDU length and the Channel Identifier (CID).

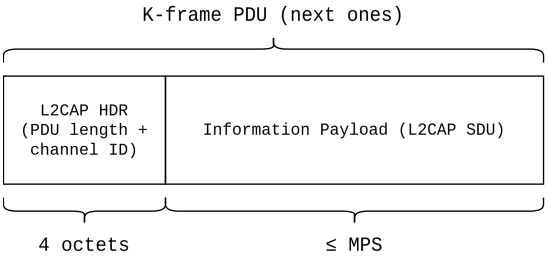
PDU Types

B-frame: Basic information frame PDU used in Basic L2CAP mode. It contains the payload received from the upper layer or delivered to the upper layer as its payload.



K-frame: Credit-based frame PDU used in LE Credit Based Flow Control mode and Enhanced Credit Based Flow Control mode. It contains a SDU segment and additional protocol information.





Relevant Kconfig

Kconfig symbol	Description
CONFIG_BT_BUF_ACL_RX_SIZE	Represents the MPS
CONFIG_BT_L2CAP_TX_MTU	Represents the L2CAP MTU
CONFIG_BT_L2CAP_DYNAMIC_CHAN	Enables LE Credit Based Flow Control and thus the stack may use *K-frame* PDUs

GATT

The Generic Attribute Profile is the most common means of communication over LE connections. A more detailed description of this layer and the API reference can be found in the *GATT API reference section*.

Mesh

Mesh is a little bit special when it comes to the needed GAP roles. By default, mesh requires both observer and broadcaster role to be enabled. If the optional GATT Proxy feature is desired, then peripheral role should also be enabled.

The API reference for mesh can be found in the *Mesh API reference section*.

LE Audio

The LE audio is a set of profiles and services that utilizes GATT and Isochronous Channel to provide audio over Bluetooth Low Energy. The architecture and API references can be found in *Bluetooth Audio Architecture*.

Persistent storage

The Bluetooth host stack uses the settings subsystem to implement persistent storage to flash. This requires the presence of a flash driver and a designated "storage" partition on flash. A typical set of configuration options needed will look something like the following:

```
CONFIG_BT_SETTINGS=y
CONFIG_FLASH=y
CONFIG_FLASH_PAGE_LAYOUT=y
CONFIG_FLASH_MAP=y
CONFIG_NVS=y
CONFIG_SETTINGS=y
```

Once enabled, it is the responsibility of the application to call settings_load() after having initialized Bluetooth (using the *bt_enable()* API).

6.1.5 LE Audio Stack

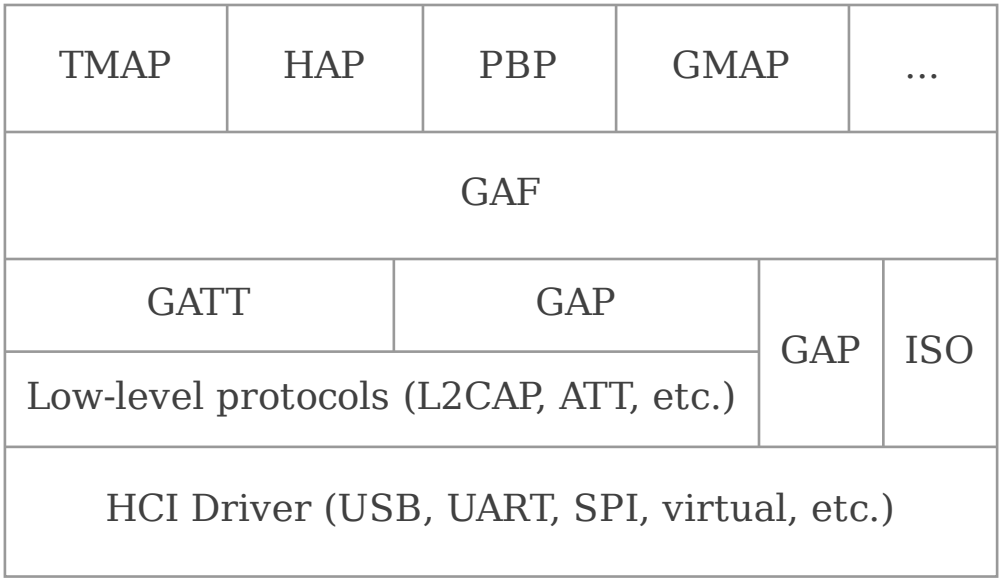


Fig. 4: Bluetooth Audio Architecture

Overall design

The overall design of the LE Audio stack is that the implementation follows the specifications as closely as possible, both in terms of structure but also naming. Most API functions are prefixed by the specification acronym (e.g. *bt_bap* for the Basic Audio Profile (BAP) and *bt_vcp* for the Volume Control Profile (VCP)). The functions are then further prefixed with the specific role from each profile where applicable (e.g. `bt_bap_unicast_client_discover()` and `bt_vcp_vol_rend_set_vol()`). There are usually a function per procedure defined by the profile or service specifications, and additional helper or meta functions that do not correspond to procedures.

The structure of the files generally also follow this, where BAP related files are prefixed with *bap* and VCP related files are prefixed with *vcp*. If the file is specific for a profile role, the role is also embedded in the file name.

Generic Audio Framework (GAF)

The Generic Audio Framework (GAF) is considered the middleware of the Bluetooth LE Audio architecture. The GAF contains the profiles and services that allows higher layer applications and profiles to set up streams, change volume, control media and telephony and more. The GAF builds on GATT, GAP and isochronous channels (ISO).

GAF uses GAP to connect, advertise and synchronize to other devices. GAF uses GATT to configure streams, associate streams with content (e.g. media or telephony), control volume and more. GAF uses ISO for the audio streams themselves, both as unicast (connected) audio streams or broadcast (unconnected) audio streams.

GAF mandates the use of the LC3 codec, but also supports other codecs.

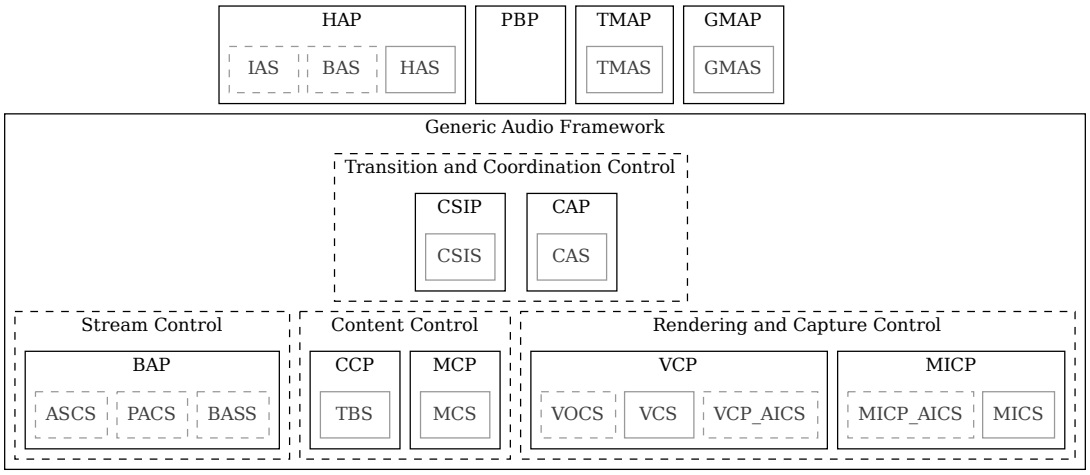


Fig. 5: Generic Audio Framework (GAF)

The top-level profiles TMAP and HAP are not part of the GAF, but rather provide top-level requirements for how to use the GAF.

GAF and the top layer profiles gave been implemented in Zephyr with the following structure.

Profile Dependencies The LE Audio profiles depend on other profiles and services, as outlined in the following tables. In these tables 'Server' refers to acting in the GATT server role, and 'Client' refers to acting in the GATT client role for the specific service. If a profile role depends on another

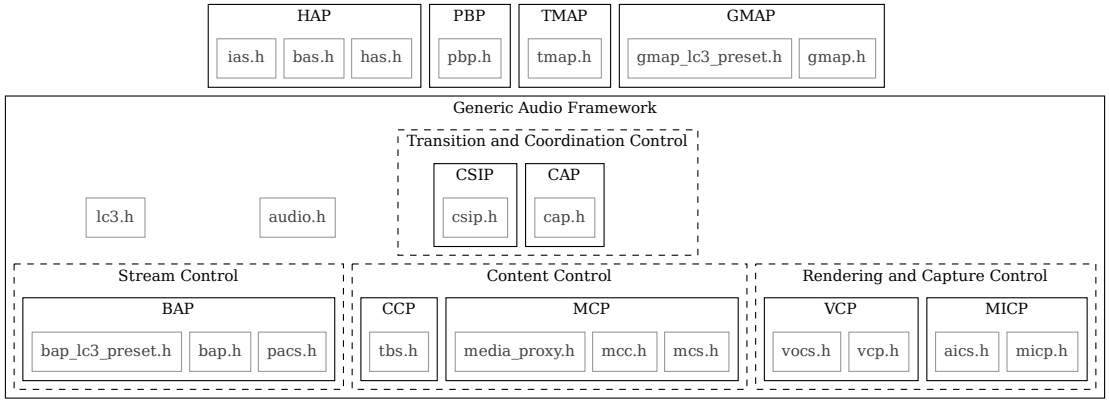


Fig. 6: Zephyr Generic Audio Framework

profile that depends on a service, then that dependency is implicitly also applied to that profile. For example, if the CAP Acceptor uses the BAP Unicast Server role, then the requirements on the ASCS Server and PACS Server also apply to the CAP Acceptor.

The dependencies for Stream Control (BAP) are in the following table.

Table 1: BAP dependencies

	Unicast Server	Unicast Client	Broadcast Source	Broadcast Sink	Scan Delegator	Broadcast Assistant
BAP Scan Delegator				M		
ASCS Client		M				
ASCS Server	M					
PACS Client		M				O
PACS Server	M			M		
BASS Client						M
BASS Server					M	

Note:

- As the table shows, the Broadcast Source role has no dependencies on other LE Audio profiles or services

The dependencies for Content Control (MCP and CCP) are in the following tables.

Table 2: MCP dependencies

	Media Control Server	Media Control Client
GMCS Server	M	
GMCS Client		M
MCS Server	O	
MCS Client		O
OTS Server	O	
OTS Client		O

Table 3: CCP dependencies

	Call Control Server	Call Control Client
GTBS Server	M	
GTBS Client		M
TBS Server	M	
TBS Client		M

The dependencies for Rendering Control (MICP and VCP) are in the following tables.

Table 4: MICP dependencies

	Microphone Controller	Microphone Device
MICS Server	M	
MICS Client		M
AICS Server	O	
AICS Client		O

Table 5: VCP dependencies

	Volume Renderer	Volume Controller
VCS Server	M	
VCS Client		M
VOCS Server	O	
VOCS Client		O
AICS Server	O	
AICS Client		O

The last element in GAF is Transition and Coordination Control (CAP and CSIP) with the dependencies from the following tables.

Table 6: CAP dependencies

	Acceptor	Initiator	Commander
CAS Server	M		C.8
CAS Client		M	M
BAP Unicast Client		C.1	
BAP Unicast Server	C.2		
BAP Broadcast Source		C.1	
BAP Broadcast Sink	C.2		
BAP Broadcast Assistant			C.4, C.6
BAP Scan Delegator	C.3		C.6
VCP Volume Controller			C.6
VCP Volume Renderer	O		
MICP Microphone Controller			C.6
MICP Microphone Device	O		
CCP Call Control Server		O	
CCP Call Control Client	O		C.6
MCP Media Control Server		O	
MCP Media Control Client	O		C.6
CSIP Set Coordinator		C.5	M
CSIP Set Member	C.7		

Notes:

- C.1: Support at least one of BAP Unicast Client or BAP Broadcast Source
- C.2: Support at least one of BAP Unicast Server or BAP Broadcast Sink
- C.3: Mandatory if BAP Broadcast Sink
- C.4: Mandatory if BAP Scan Delegator
- C.5: Mandatory if BAP Unicast Client
- C.6: Support at least one
- C.7: Mandatory if part of a coordinated set
- C.8: Mandatory if the Commander transmits CAP announcements

Table 7: CSIP dependencies

	Set Member	Set Coordinator
CSIS Server	M	
CSIS Client		M

The dependencies of the higher level profiles (GMAP, HAP, PBP and TMAP) are listed in the following tables.

Table 8: GMAP dependencies

	Unicast Game Gateway	Unicast Game Terminal	Broadcast Game Sender	Broadcast Game Receiver
GMAS Server	M	M	O	M
GMAS Client	M	O	O	O
CAP Initiator	M		M	
CAP Acceptor		M		M
CAP Commander	M		M	
BAP Broadcast Source			M	
BAP Broadcast Sink				M
BAP Unicast Client	M			
BAP Unicast Server		M		
VCP Volume Controller	M			
VCP Volume Renderer		C.1		M
MICP Microphone Controller	O			
MICP Microphone Device		C.2		

Notes:

- C.1 Mandatory if the UGT supports the UGT Sink feature
- C.2 Optional if the UGT supports the UGT Source feature

Table 9: HAP dependencies

	Hearing Aid	Hearing Aid Unicast Client	Hearing Aid Remote Controller
HAS Client			M
HAS Server	M		
CAP Initiator		M	
CAP Acceptor	M		
CAP Commander			M
BAP Unicast Client		M	
BAP Unicast Server	M		
VCP Volume Controller			M
VCP Volume Renderer	M		
VOCS Server	C.1		
AICS Server	O		
MICP Microphone Controller			O
MICP Microphone Device	C.2		
CCP Call Control Client	O		
CCP Call Control Server		O	
CSIP Set Coordinator		M	M
CSIP Set Member	C.3		
BAS Server	C.4		
IAS Server	O		

Notes:

- C.1 Mandatory if the HA supports the Volume Baslance feature and is part of a Binaural Hearing Aid Set
- C.2 Mandatory if the HA supports the BAP Audio Source Role
- C.3 Mandatory if the HA is capable of being part of a Binaural Hearing Aid set
- C.4 If equipped with batteries
- C.5 If CCP Call Control Server is supported

Table 10: PBP dependencies

	Public Broadcast Source	Public Broadcast sink	Public Broadcast Assistant
CAP Initiator	M		
CAP Acceptor		M	
CAP Commander			M
BAP Broadcast Assistant			M

Table 11: TMAP dependencies

	Call Gateway	Call Terminal	Unicast Media Sender	Unicast Media Receiver	Broadcast Media Sender	Broadcast Media Receiver
TMAS Server	M	M	M	M	O	M
TMAS Client	O	O	O	O	O	O
CAP Initiator	M		M		M	
CAP Acceptor		M		M		M
CAP Commander	M	O	M	O	O	O
BAP Broadcast Source					M	
BAP Broadcast Sink						M
BAP Unicast Client	M		M			
BAP Unicast Server		M		M		
VCP Volume Controller	M		M			
VCP Volume Renderer		C.1		M		M
MCP Media Control Server			M			
CCP Call Control Server	M					

Notes:

- C.1 Mandatory to support if the BAP Unicast Server is acting as an Audio Sink

Bluetooth Audio Stack Status The following table shows the current status and support of the profiles in the Bluetooth Audio Stack.

Table 12: Bluetooth Audio Profile status

Module	Role	Version	Added in Release	Status	Remaining
VCP	Volume Renderer	1.0	2.6	• Feature complete • Shell Module • BSIM test	• Sample Application

continues on next page

Table 12 – continued from previous page

Module	Role	Version	Added in Release	Status	Remaining
	Volume Controller	1.0	2.6	• Feature complete • Shell Module • BSIM test	• Sample Application
MICP	Microphone Device	1.0	2.7	• Feature complete • Shell Module • BSIM test	• Sample Application
	Microphone Controller	1.0	2.7	• Feature complete • Shell Module • BSIM test	• Sample Application
CSIP	Set Member	1.0.1	3.0	• Feature complete • Shell Module • BSIM test	• Sample Application
	Set Coordinator	1.0.1	3.0	• Feature complete • Shell Module • BSIM test	• Sample Application
CCP	Call Control Server	1.0	3.0	• Feature complete • Shell Module • BSIM test	• API refactor • Sample Application

continues on next page

Table 12 – continued from previous page

Module	Role	Version	Added in Release	Status	Remaining
	Call Control Client	1.0	3.0	• Feature complete • Shell Module • BSIM test	• API refactor • Sample Application
MCP	Media Control Server	1.0	3.0	• Feature complete • Shell Module • BSIM test	• API refactor • Support for multiple instances and connections • Sample Application
	Media Control Client	1.0	3.0	• Feature complete • Shell Module • BSIM test	• API refactor • Sample Application
BAP	Unicast Server	1.0.1	3.0	• Feature complete • Shell Module • BSIM test • Sample Application	

continues on next page

Table 12 – continued from previous page

Module	Role	Version	Added in Release	Status	Remaining
	Unicast Client	1.0.1	3.0	• Feature complete • Shell Module • BSIM test • Sample Application	
	Broadcast Source	1.0.1	3.0	• Feature complete • Shell Module • BSIM test • Sample Application	
	Broadcast Sink	1.0.1	3.0	• Feature complete • Shell Module • BSIM test • Sample Application	
	Scan Delegator	1.0.1	3.3	• Feature complete • Shell Module • BSIM test • Sample Application	

continues on next page

Table 12 – continued from previous page

Module	Role	Version	Added in Release	Status	Remaining
	Broadcast Assistant	1.0.1	3.3	• Feature complete • Shell Module • BSIM test • Sample Application	
CAP	Acceptor	1.0	3.2	• Feature complete • Shell Module • BSIM test • Sample Application	
	Initiator	1.0	3.3	• Feature complete • Shell Module • BSIM test • Sample Application	
	Commander			• WIP	• Feature complete • Shell Module • BSIM test • Sample Application

continues on next page

Table 12 – continued from previous page

Module	Role	Version	Added in Release	Status	Remaining
HAP	Hearing Aid	1.0	3.1	• Feature complete • Shell Module • BSIM test • Sample Application	
	Hearing Aid Unicast Client	1.0	3.1	• Feature complete • Shell Module • BSIM test • Sample Application	
	Hearing Aid Remote Controller			• WIP	• Feature complete • Shell Module • BSIM test • Sample Application
TMAP	Call Gateway	1.0	3.4	• Feature complete • Shell Module • BSIM test • Sample Application	

continues on next page

Table 12 – continued from previous page

Module	Role	Version	Added in Release	Status	Remaining
	Call Terminal	1.0	3.4	• Feature complete • Shell Module • BSIM test • Sample Application	
	Unicast Media Sender	1.0	3.4	• Feature complete • Shell Module • BSIM test • Sample Application	
	Unicast Media Receiver	1.0	3.4	• Feature complete • Shell Module • BSIM test • Sample Application	
	Broadcast Media Sender	1.0	3.4	• Feature complete • Shell Module • BSIM test • Sample Application	

continues on next page

Table 12 – continued from previous page

Module	Role	Version	Added in Release	Status	Remaining
	Broadcast Media Receiver	1.0	3.4	• Feature complete • Shell Module • BSIM test • Sample Application	
PBP	Public Broadcast Source		3.5	• Feature complete • Shell Module • BSIM test • Sample Application	
	Public Broadcast Sink		3.5	• Feature complete • Shell Module • BSIM test • Sample Application	
	Public Broadcast Assistant				• Feature complete • Shell Module • BSIM test • Sample Application

continues on next page

Table 12 – continued from previous page

Module	Role	Version	Added in Release	Status	Remaining
GMAP	Unicast Game Gateway		3.5	• Feature complete • Shell Module • BSIM test	• Sample Application
	Unicast Game Terminal		3.5	• Feature complete • Shell Module • BSIM test	• Sample Application
	Broadcast Game Sender		3.5	• Feature complete • Shell Module • BSIM test	• Sample Application
	Broadcast Game Receiver		3.5	• Feature complete • Shell Module • BSIM test	• Sample Application

Using the Bluetooth Audio Stack To use any of the profiles in the Bluetooth Audio Stack, including the top-level profiles outside of GAF, `CONFIG_BT_AUDIO` shall be enabled. This Kconfig option allows the enabling of the individual profiles inside of the Bluetooth Audio Stack. Each profile can generally be enabled on its own, but enabling higher-layer profiles (such as CAP, TMAP and HAP) will typically require enabling some of the lower layer profiles.

It is, however, possible to create a device that uses e.g. only Stream Control (with just the BAP), without using any of the content control or rendering/capture control profiles, or vice versa. Using the higher layer profiles will however typically provide a better user experience and better interoperability with other devices.

Common Audio Profile (CAP) The Common Audio Profile introduces restrictions and requirements on the lower layer profiles. The procedures in CAP works on one or more streams for one or more devices. Is it thus possible via CAP to do a single function call to setup multiple streams across multiple devices.

The figure below shows a complete structure of the procedures in CAP and how they correspond to procedures from the other profiles. The circles with I, A and C show whether the procedure

has active involvement or requirements from the CAP Initiator, CAP Accept and CAP Commander roles respectively.

The API reference for CAP can be found in *Common Audio Profile*.

Stream Control (BAP) Stream control is implemented by the Basic Audio Profile. This profile defines multiple roles:

- Unicast Client
- Unicast Server
- Broadcast Source
- Broadcast Sink
- Scan Delegator
- Broadcast Assistant

Each role can be enabled individually, and it is possible to support more than one role.

Notes about the stream control services There are 3 services primarily used by stream control using the Basic Audio Profile.

Audio Stream Control Service (ASCS) ASCS is a service used exclusively for setting up unicast streams, and is located on the BAP Unicast Server device. The service exposes one or more endpoints that can either be a sink or source endpoint, from the perspective of the Unicast Server. That means a sink endpoint is always audio from the Unicast Client to the Unicast Server, and a source endpoint is always from the Unicast Server to the Unicast Client.

Unlike most other GATT services, ASCS require that each characteristic in the service has unique data per client. This means that if a Unicast Server is connected to multiple Unicast Clients, the Unicast Clients are not able to see or control the endpoints configured by the other clients. For example if a person's smartphone is streaming audio to a headset, then the same person will not be able to see or control that stream from their smartwatch.

Broadcast Audio Scan Service (BASS) BASS is a service that is exclusively used by the Scan Delegator and Broadcast Assistant. The main purpose of the service is to offload scanning from low power peripherals to e.g. phones and PCs. Unlike ASCS where the data is required to be unique per client, the data in BASS (called receive states) are (usually) shared among all connected clients. That means it is possible for a person to tell their headphones to synchronize to a Broadcast Source using their phone, and then later tell their headphones to stop synchronizing using their smartwatch.

A Broadcast Assistant can be any device, and may only support this one role without any audio capabilities. This allows legacy devices that do not support periodic advertisements or isochronous channels to still provide an interface and scan offloading for peripherals. The Bluetooth SIG have provided a guide on how to develop such legacy Broadcast Assistants that can be found at https://www.bluetooth.com/bluetooth-resources/developing-auracast-receivers-with-an-assistant-application-for-legacy-smartphones/. An important note about this guide is that many operating systems (especially on phones), do not allow generic usage of the BASS UUID, effectively making it impossible to implement your own Broadcast Assistant, because you cannot access the BASS.

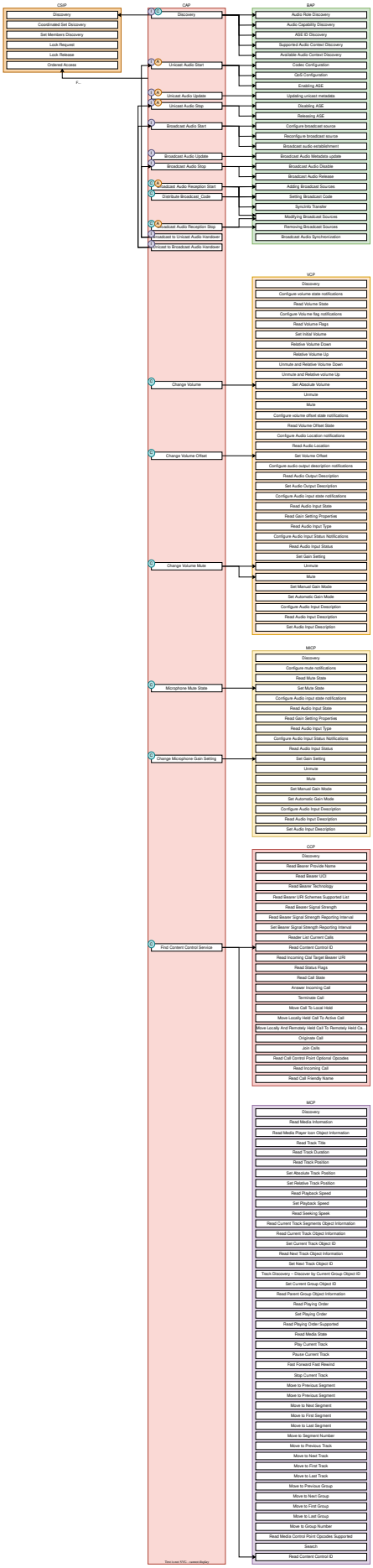


Fig. 7: Common Audio Profile Procedures

Published Audio Capabilities Service (PACS) PACS is used to expose a device's audio capabilities in Published Audio Capabilities (PAC) records. PACS is used by nearly all roles, where the Unicast Client and Broadcast Assistant will act as PACS clients, and Unicast Server and Broadcast Sink will act as PACS servers. These records contain information about the codec, and which values are supported by each codec. The values for the LC3 codec are defined by the Bluetooth Assigned numbers (https://www.bluetooth.com/specifications/assigned-numbers/), and the values for other codecs such as SBC are left undefined/implementation specific for BAP.

PACS also usually share the same data between each connected client, but by using functions such as *bt_pacs_conn_set_available_contexts_for_conn()*, it is possible to set specific values for specific clients.

The API reference for stream control can be found in *Bluetooth Audio*.

Rendering and Capture Control Rendering and capture control is implemented by the Volume Control Profile (VCP) and Microphone Control Profile (MICP).

The VCP implementation supports the following roles

- Volume Control Service (VCS) Server
- Volume Control Service (VCS) Client

The MICP implementation supports the following roles

- Microphone Control Profile (MICP) Microphone Device (server)
- Microphone Control Profile (MICP) Microphone Controller (client)

The API reference for volume control can be found in *Bluetooth Volume Control*.

The API reference for Microphone Control can be found in *Bluetooth Microphone Control*.

Content Control Content control is implemented by the Call Control Profile (CCP) and Media Control Profile (MCP).

The CCP implementation is not yet implemented in Zephyr.

The MCP implementation supports the following roles

- Media Control Service (MCS) Server via the Media Proxy module
- Media Control Client (MCC)

The API reference for media control can be found in *Bluetooth Media Control*.

Generic TBS and Generic MCS Both the Telephone Bearer Service (TBS) used by CCP and the Media Control Service (MCS) used by MCP have the concept of generic instances of the services called Generic TBS (GTBS) and Generic MCS (GMCS).

While these share a common name prefix, the behavior of these two may be significantly different.

Generic TBS The TBS spec defines GTBS as

> GTBS provides a single point of access and exposes a representation of its internal telephone bearers into a single telephone bearer. This service provides telephone status and control of the device as a single unit with a single set of characteristics. It is left up to the implementation to determine what telephone bearer a characteristic of GTBS represents at any time. There is no specified manner of representing a characteristic from each individual TBS that resides on the device to the same characteristic of the GTBS.

> For example, if there is more than one TBS on a device and each has a unique telephone bearer name (e.g., Name1 and Name2), the way the GTBS represents the telephone bearer name is left up to the implementation. GTBS is suited for clients that do not need to access or control all the information available on specific telephone bearers.

This means that a GTBS instance represents one or more telephone bearers. A telephone bearer could be any application on a device that can handle (telephone) calls, such as the default Call application on a smartphone, but also other applications such as Signal, Discord, Teams, Slack, etc.

GTBS may be standalone (i.e.the device only has a GTBS instance without any TBS instances), and the behavior of the GTBS is mostly left up to the implementation. In Zephyr the implementation of GBTS is that it contains some generic information, such as the provider name which is defined to simply be "Generic TBS", but the majority of the information in the GTBS instance in Zephyr has been implemented to be a union of the data of the other bearers. For example if you have a bearer for regular phone calls and Teams and have an active call in both bearers, then each of those bearers will report a single call, but the GTBS instance will report 2 calls, making it possible for a simple Call Control Client to control all calls from a single bearer. Similarly the supported URIs for each bearer are also made into a union in GTBS, and when placing a call using the GTBS the server will pick the most suited bearer depending on the URI. For example calls with URI *tel* would go to the regular phone application, and calls with the URI *skype* would go to the Teams application.

In conclusion the GTBS implementation in Zephyr is a union of the non-generic telephone bearers.

Generic MCS The MCS spec defines GMCS as

> The GMCS provides status and control of media playback for the device as a single unit. An MCS instance describes and controls the media playback for a specific media player within the device. A device implements MCS instances to allow clients to access the separate internal media player entities.

and where the behavior of GMCS is defined as

> ... the behavior of MCS and GMCS is identical, and all the characteristics and the characteristics' behaviors are the same. The term "MCS" is used throughout the document. Unless otherwise specifically stated in this specification, the same meaning applies to GMCS as well.

This means that a GMCS instance works the same way as an MCS instance, and it follows that GMCS

> controls the media playback for a specific media player within the device

A media player on a device could be anything that plays media, such as a Spotify or Youtube application on a smartphone. Thus if a device has multiple MCS instances, then each of these control media for that specific application, but the GMCS also controls media playback for a specific media player. GMCS can thus be considered a pointer to a specific MCS instance, and control either e.g. Spotify or Youtube, but not both.

The MCS spec does however provide an example of GMCS where a device can

> Implement a GMCS that provides status and control of media playback for the device as a whole.

Which may indicate that an implementation may use GMCS to represent all media players with GMCS and not a specific media player as stated above. In the case where a device does not have any MCS instances and only GMCS, then GMCS will point to a generic instance.

The Zephyr implementation of MCS and GMCS is incomplete, and currently only supports instantiating a single instance that can either be an MCS or GMCS. This means that the implementation is neither complete nor spec-compliant.

Difference between GTBS and GMCS The definitions and implementations of GTBS and GMCS as stated above are notably different. GTBS works as a union between the other TBS instances (if any), and GMCS works as a pointer to a specific MCS instance (if any). This effectively means that a simple Call Control Client can control all calls just using GTBS, but a Media Control Client may only be able to control a single player using GMCS.

Coordinated Sets Coordinated Sets is implemented by the Coordinated Sets Identification Profile (CSIP).

The CSIP implementation supports the following roles

- Coordinated Set Identification Service (CSIP) Set Member
- Coordinated Set Identification Service (CSIP) Set Coordinator

The API reference for media control can be found in *Bluetooth Coordinated Sets*.

Specification correctness and data location The implementations are designed to ensure specification compliance as much as possible. When a specification introduces a requirement with e.g. a **shall** then the implementation should attempt to ensure that this requirement is always followed. Depending on the context of this, the implementation ensures this by rejecting invalid parameters from the application, or from the remote devices.

Some requirements from the specifications are not or can not be handled by the stack itself for various reasons. One reason when the stack cannot handle a requirement is if the data related to the requirement is exclusively controlled by the application. An example of this is the advertising data, where multiple service have requirements for what to advertise and when, but where both the advertising state and data is exclusively controlled by the application.

Oppositely there are also requirements from the specification, where the data related to the requirement is exclusively controlled by the stack. An example of this is the Volume Control Service (VCS) state, where the specifications mandata that the VCP Volume Renderer (VCS server) modify the values without a choice, e.g. when setting the absolutely volume. In cases like this the application is only notified about the change with a callback, but cannot reject the request (the stack will reject any invalid requests).

Generally when the data is simple (like the VCS state which only take up a few bytes), the data is kept in and controlled by the stack, as this can ensure that the requirements can be handled by the stack, making it easier to use a profile role correctly. When the data is more complex (e.g. the PAC records), the data may be kept by the application and the stack only contains a reference to it. When the data is very application specific (e.g. advertising data), the data is kept in and controlled by the application.

As a rule of thumb, the return types of the callbacks for each profile implementation indicate whether the data is controlled by the stack or the application. For example all the callbacks for the VCP Volume Renderer have the return type of *void*, but the return type of the BAP Unicast Server callbacks are *int*, indicating that the application not only controls a lot of the Unicast Server data, but can also reject the requests. The choice of what the return type of the callbacks often depend on the specifications, and how much control the role has in a given context.

Things worth knowing or considering when using LE Audio This section describes a few tings to consider when contributing to or using LE Audio in Zephyr. The things described by this section are not unique to Zephyr as they are defined by the specifications.

Security requirements All LE Audio services require Security Level 2 but where the key must be 128-bit and derived via an OOB method or via LE Secure connections. There is no Core-spec defined way of reporting this in GATT, as ATT does not have a specific error code for missing OOB method or LE Secure Connections (although there is a way to report wrong key size).

In Zephyr we do not force the device to always use these, as a device that uses LE Audio may also use other profiles and services that do not require such security. We guard all access to services using a custom security check implemented in subsys/bluetooth/audio/audio.c, where all LE Audio services must use the internal *BT_AUDIO_CHRC* macro for proper security verification.

Access to the LTK for encrypted SIRKs in CSIS The Coordinated Set Identification Service (CSIS) may encrypt the SIRK (set identity resolving key). The process of encrypting the SIRK requires the LTK as the encryption key, which is typically not exposed to higher layer implementations such as CSIS. This does not have any effect on the security though.

MTU requirements The Basic Audio Profile (BAP) has a requirement that both sides shall support a minimum ATT_MTU of at least 64 on the unenhanced ATT bearer or at least one enhanced ATT bearer. The requirement comes from the preferred (or sometimes mandatory) use of GATT Write Without Response, and where support for Write Long Characteristic Value is optional in most cases.

If a ASCS device supports values larger than the minimum ATT_MTU of 64 octets, then it shall support Read long Characteristic Value by setting `CONFIG_BT_ATT_PREPARE_COUNT` to a non-zero value.

6.1.6 LE Audio resources

This section contains some links and reference to resources that are useful for either contributors to the LE Audio Stack in Zephyr, LE Audio application developers or both.

The LE audio channel on Discord

Zephyr has a specific Discord channel for LE Audio development, which is open to all. Find it here at https://discordapp.com/channels/720317445772017664/1207326649591271434 or simply search for *ble-audio* from within Discord. Since the *ble-audio* channel is open for all, we cannot discuss any specifications that are in development in that channel. For discussions that require a Bluetooth SIG membership we refer to the *bluetooth-sig* Discord channel found at https://discordapp.com/channels/720317445772017664/869172014018097162.

Zephyr weekly meetings

Anyone who is a Bluetooth SIG member and a Zephyr member can join the weekly meetings where we discuss and plan the development of LE Audio in Zephyr. You can find the time of the meetings by joining the Bluetooth-sig group at https://lists.zephyrproject.org/g/Bluetooth-sig.

Github project

LE Audio in Zephyr has its own Github project available at https://github.com/orgs/zephyrproject-rtos/projects/26. The project is mostly automated, and the LE Audio contributors almost only rely on the automated workflows to present the state of development. Anyone is able to pick any of the open issues and work on it. If you cannot assign the issue to yourself, please leave a comment in the issue itself or ping the Discord channel for help.

Bluetooth SIG errata for LE Audio

There are many specifications for LE Audio, and several of them are still being updated and developed. To get an overview of the errata for the LE Audio specifications you can visit

- Generic Audio (GA) errata https://bluetooth.atlassian.net/wiki/spaces/GA/pages/1634402349/GAWG+Errata+Lists
- Hearing Aid (HA) errata https://bluetooth.atlassian.net/wiki/spaces/HA/pages/1634140216/HA+WG+Errata+List
- Audio, Telephony and Automotive (ATA) errata https://bluetooth.atlassian.net/wiki/spaces/ATA/pages/1668481034/ATA+Errata+Lists

Access to errata requires a Bluetooth SIG membership.

Bluetooth SIG working groups for LE Audio

There are 3 working groups in the Bluetooth SIG related to LE Audio:

- Generic Audio (GA) https://www.bluetooth.org/groups/group.aspx?gId=665
- Hearing Aid (HA) https://www.bluetooth.org/groups/group.aspx?gId=605
- Audio, Telephony, and Automotive (ATA) https://www.bluetooth.org/groups/group.aspx?gId=659

By joining these groups you will also get emails from their respective mailing lists, where multiple questions and discussions are handled. The working groups also have scheduled weekly meetings, where issues and the development of the specifications are handled.

Access to the Bluetooth SIG working groups requires a Bluetooth SIG membership.

The LE Audio Book

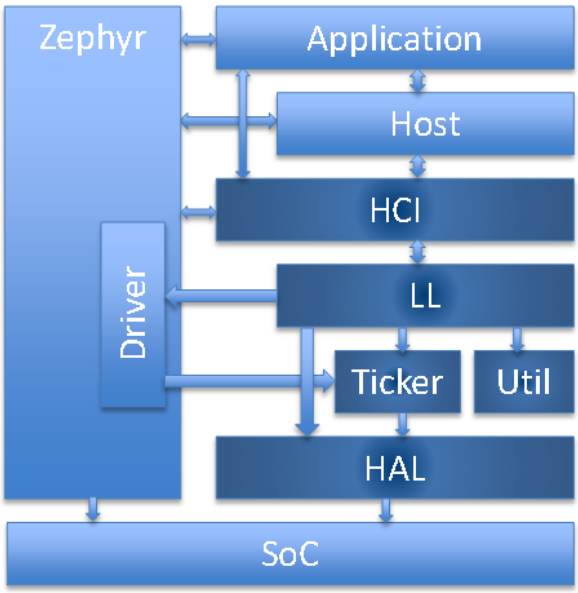
There is a free ebook on LE Audio at https://www.bluetooth.com/bluetooth-resources/le-audio-book/. The book was released in January 2022, and thus before some of the specifications were finalized, but also before some of the released updates to the specifications. Nevertheless the book still provides a good explanation for many of the concepts and ideas, but please refer to the individual specifications for technical information.

Bluetooth SIG informational papers, reports and guides

The Bluetooth SIG occasionally release new informational papers, report and guides. These can be found at https://www.bluetooth.com/bluetooth-resources/?tags=le-audio&keyword. Here you will also find the aforementioned LE Audio book, among many other good resources.

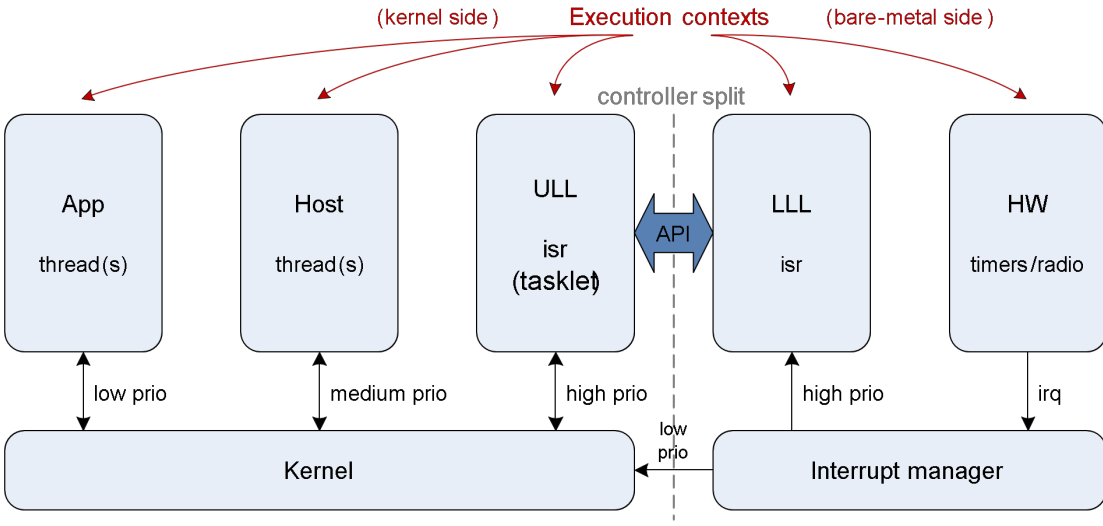
6.1.7 LE Controller

Overview

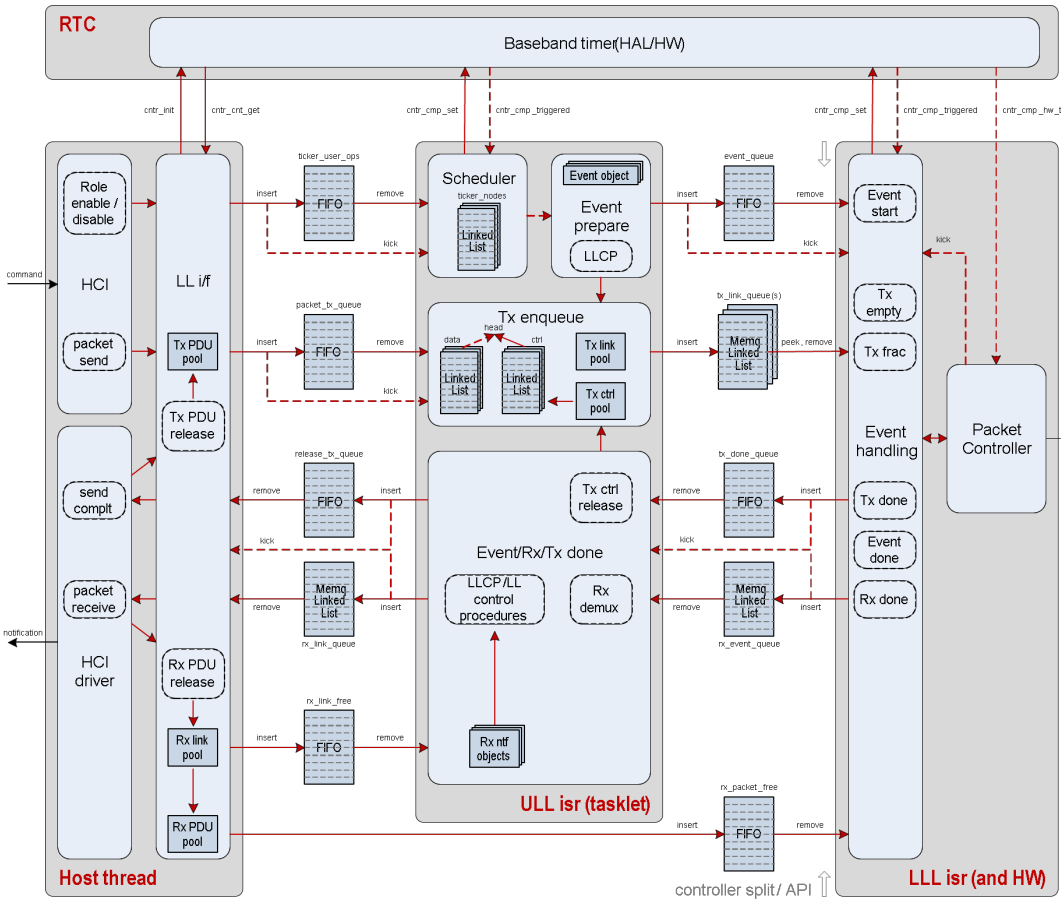


1. HCI
   - Host Controller Interface, Bluetooth standard
   - Provides Zephyr Bluetooth HCI Driver
2. HAL
   - Hardware Abstraction Layer
   - Vendor Specific, and Zephyr Driver usage
3. Ticker
   - Soft real time radio/resource scheduling
4. LL_SW
   - Software-based Link Layer implementation
   - States and roles, control procedures, packet controller
5. Util
   - Bare metal memory pool management
   - Queues of variable count, lockless usage
   - FIFO of fixed count, lockless usage
   - Mayfly concept based deferred ISR executions

Architecture

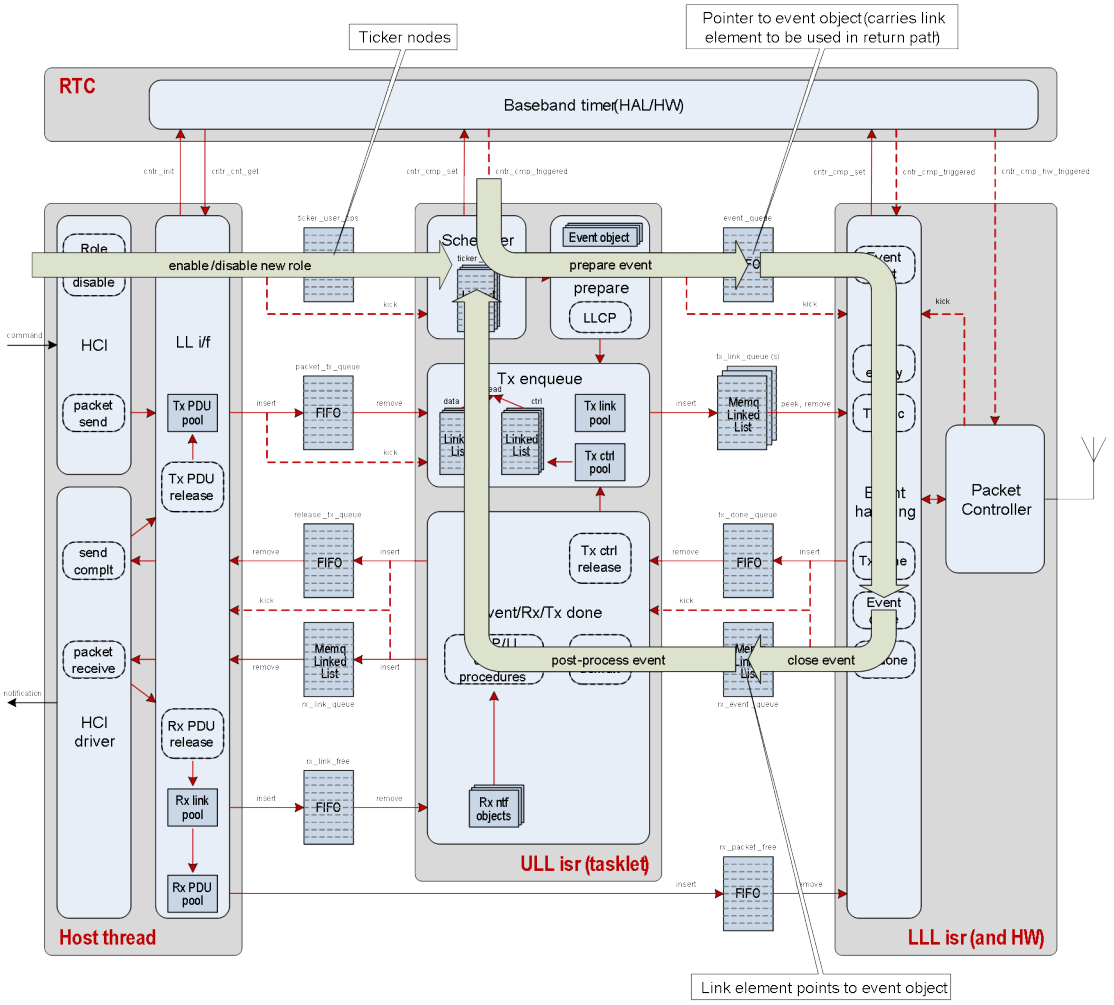


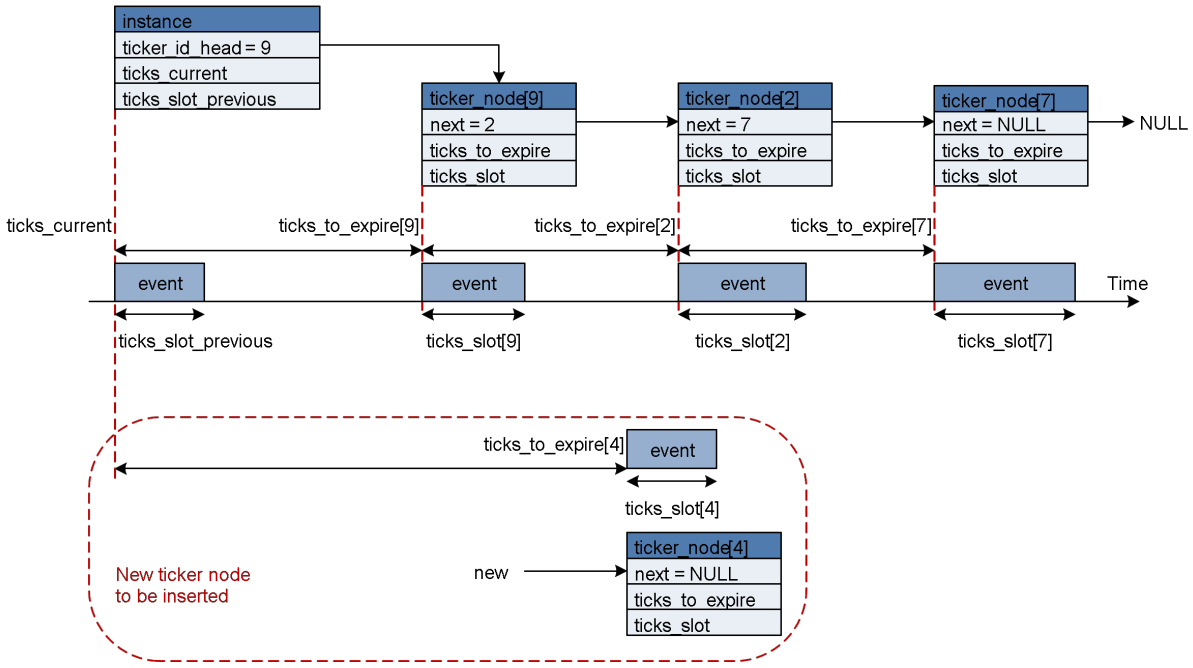
Execution Overview



Architecture Overview

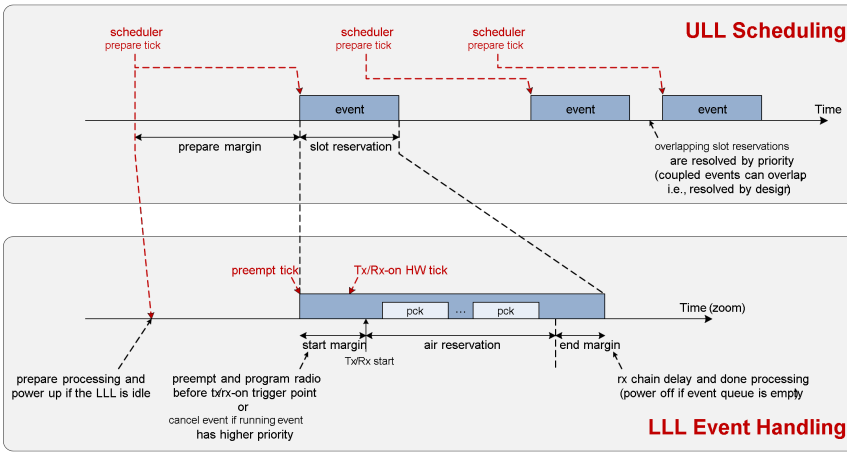
Scheduling



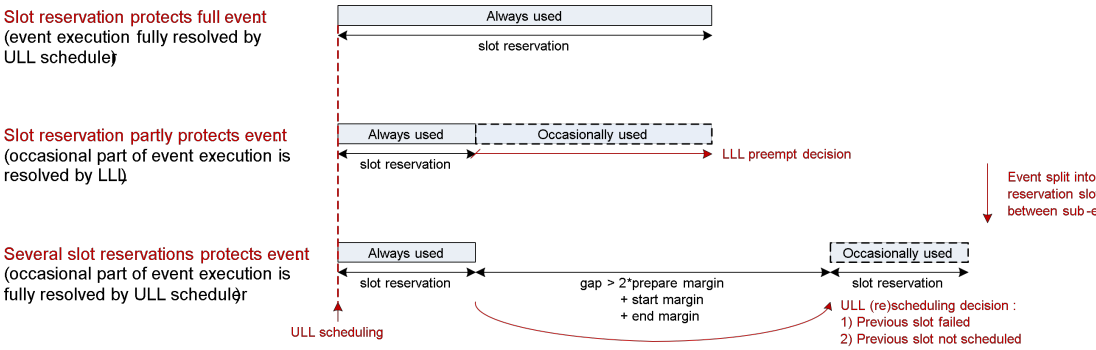


Ticker

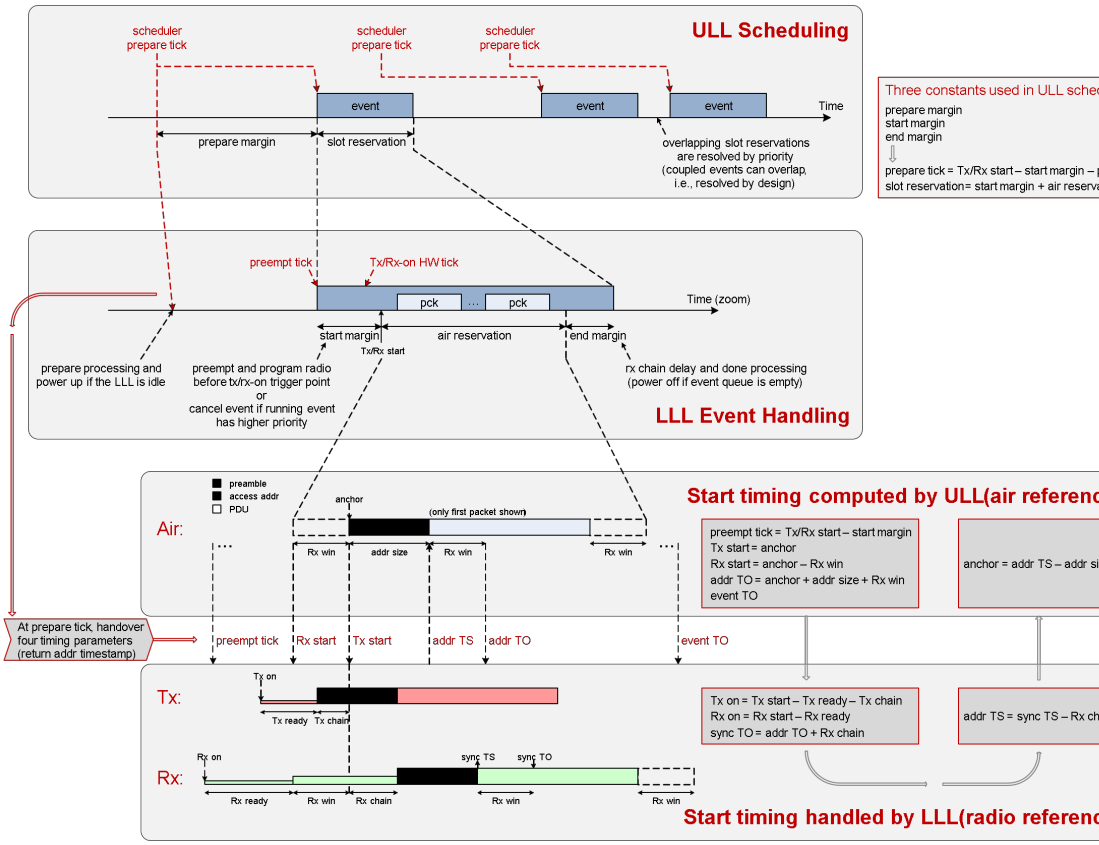
Upper Link Layer and Lower Link Layer



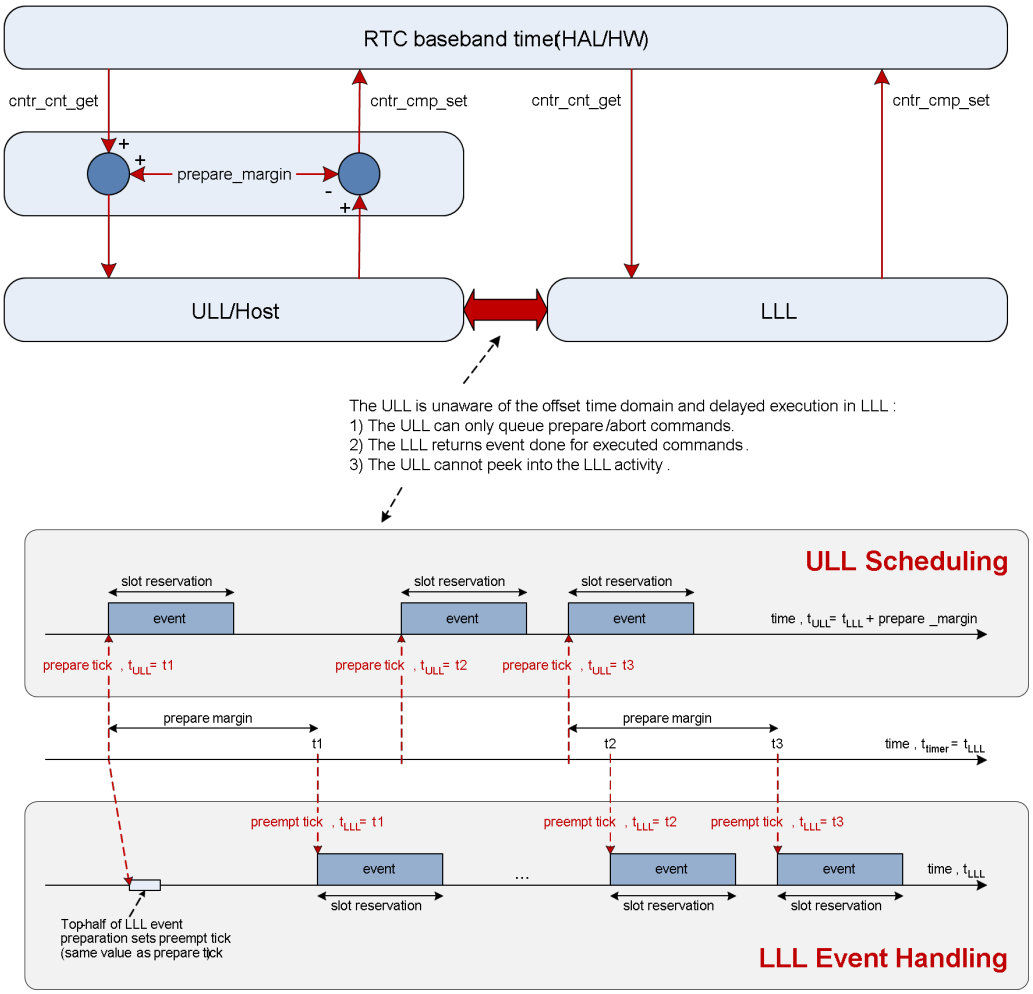
Scheduling Variants

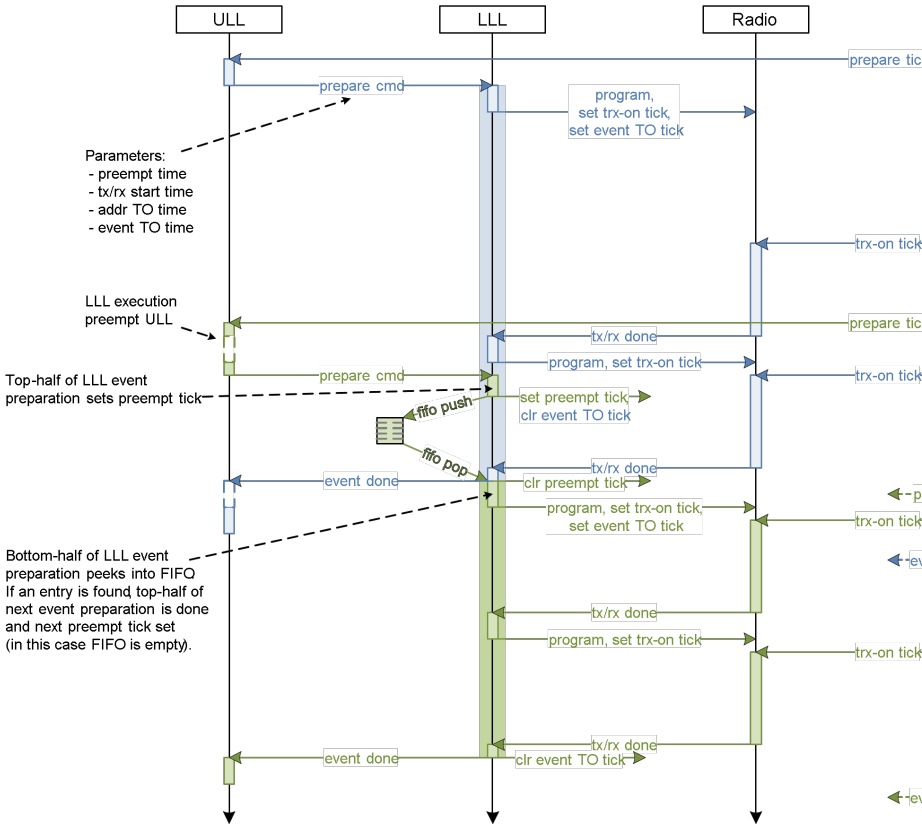


ULL and LLL Timing

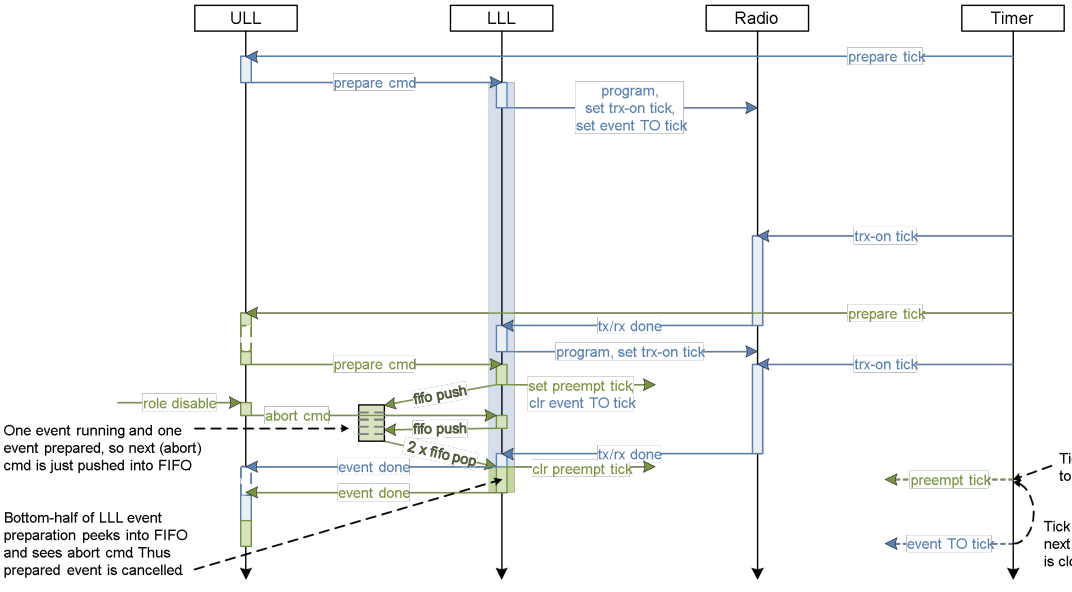


Event Handling

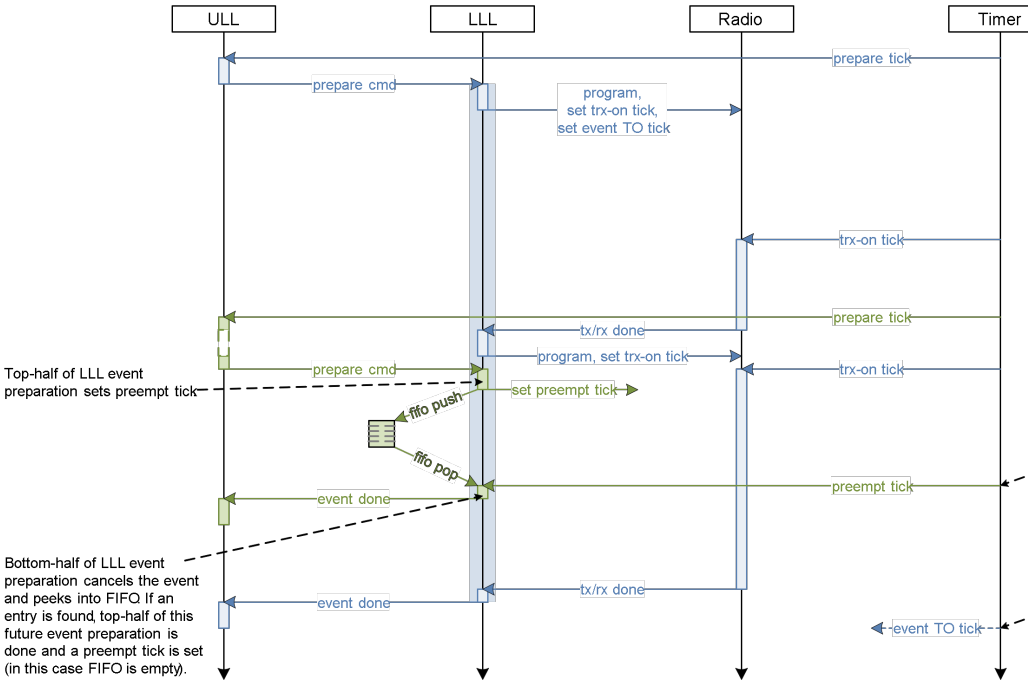




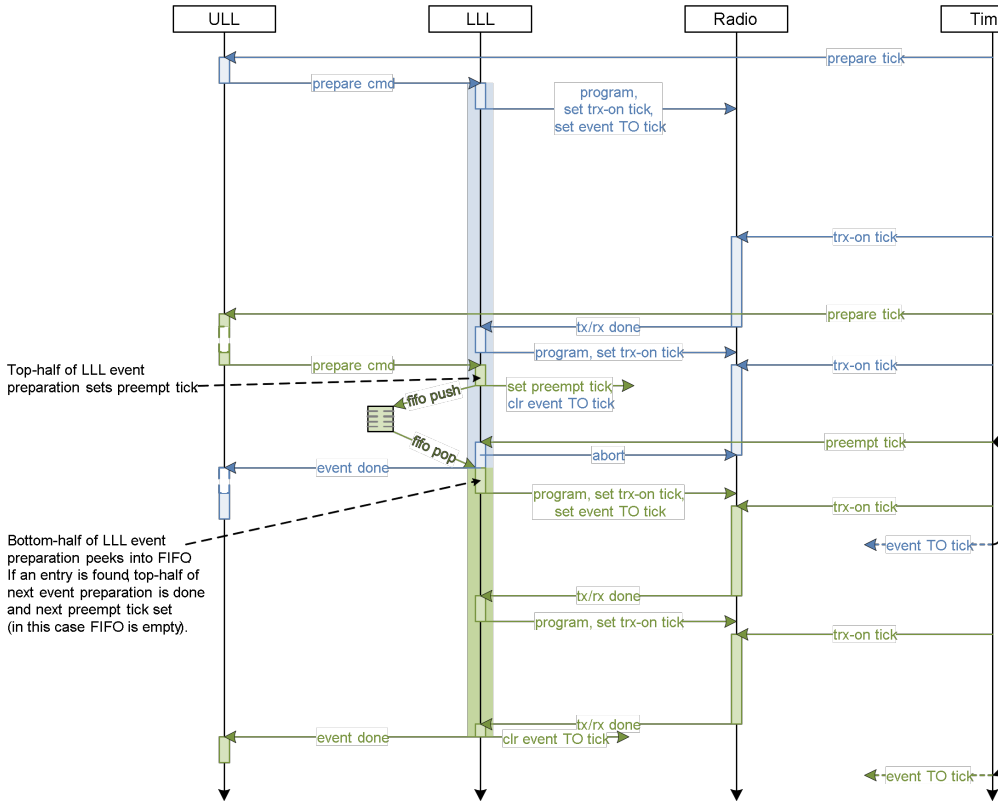
Scheduling Closely Spaced Events



Aborting Active Event

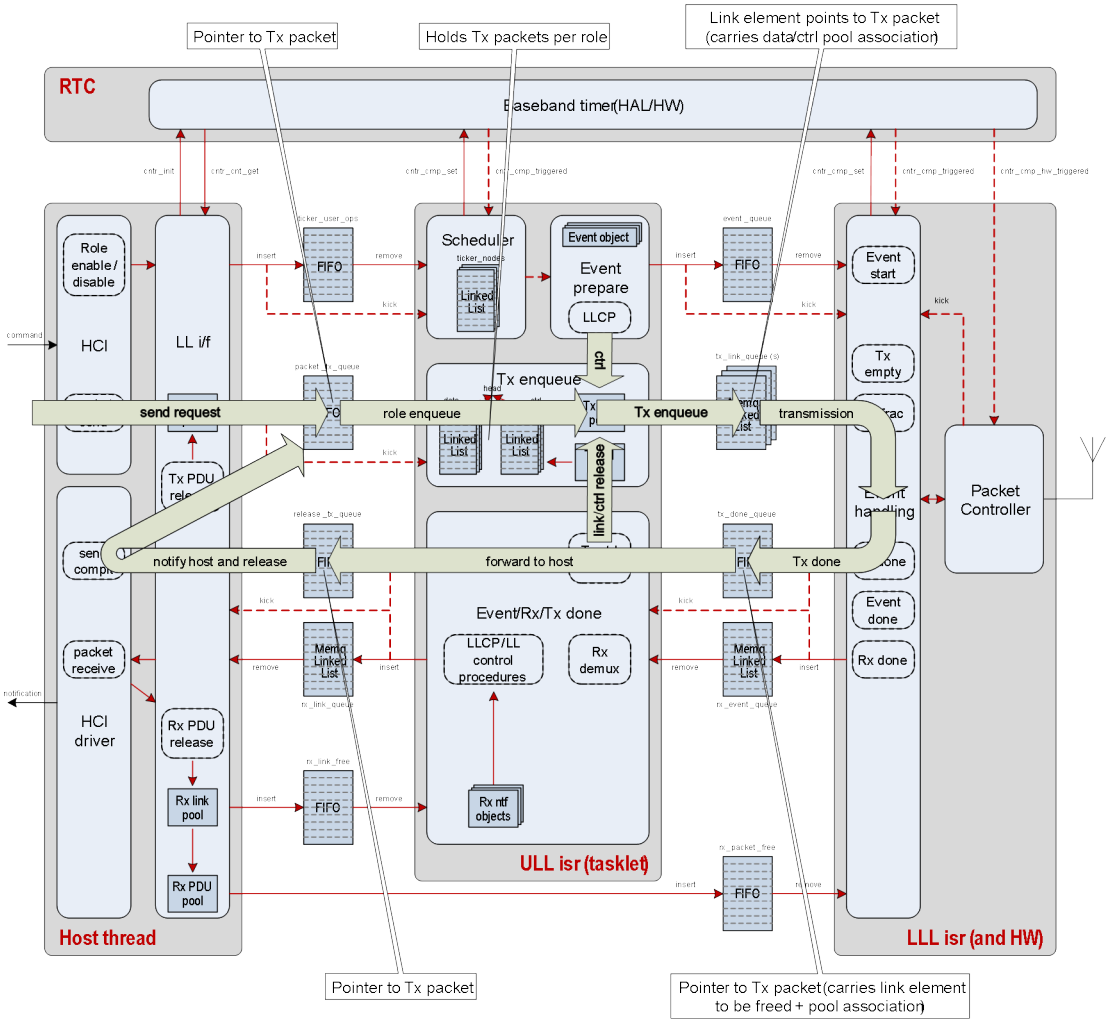


Cancelling Pending Event

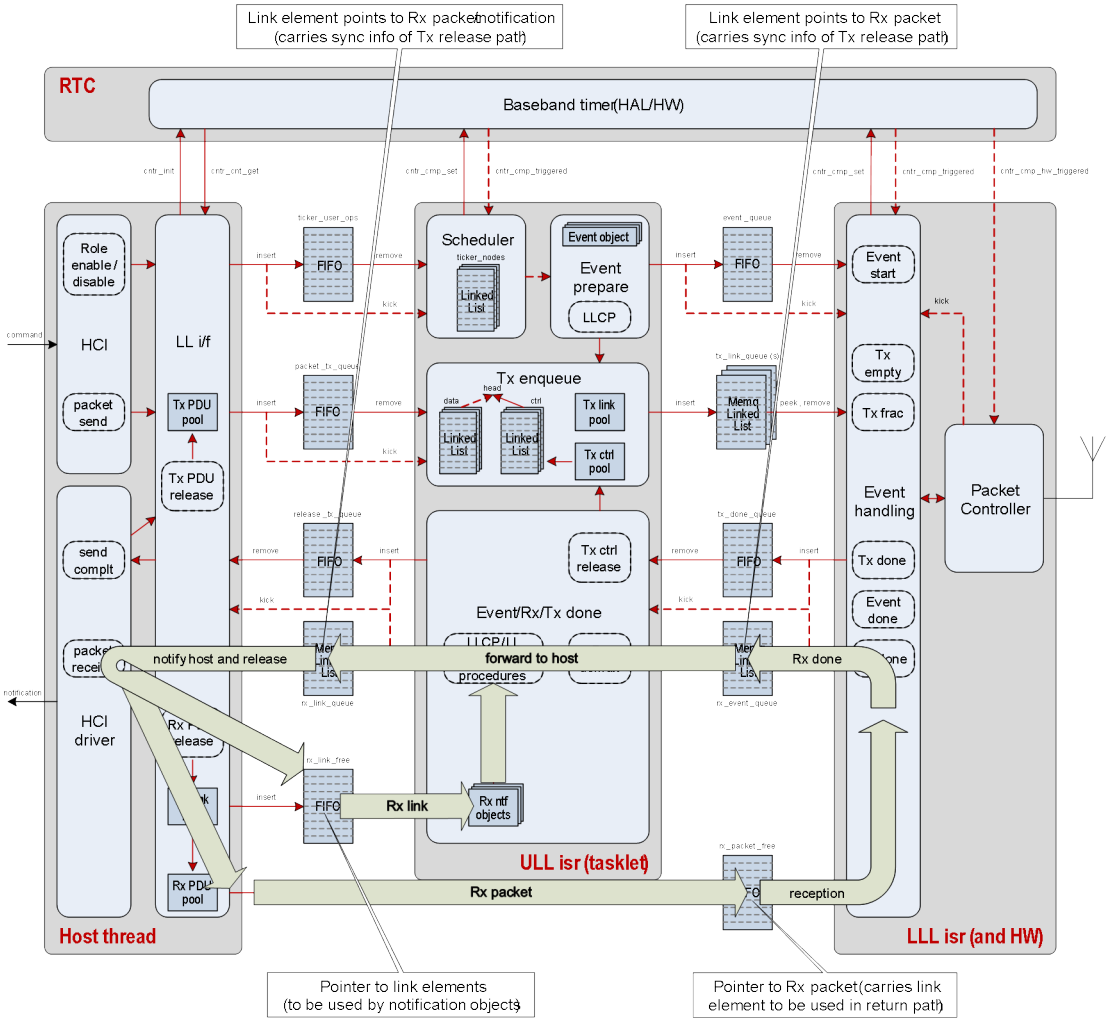


Pre-emption of Active Event

Data Flow

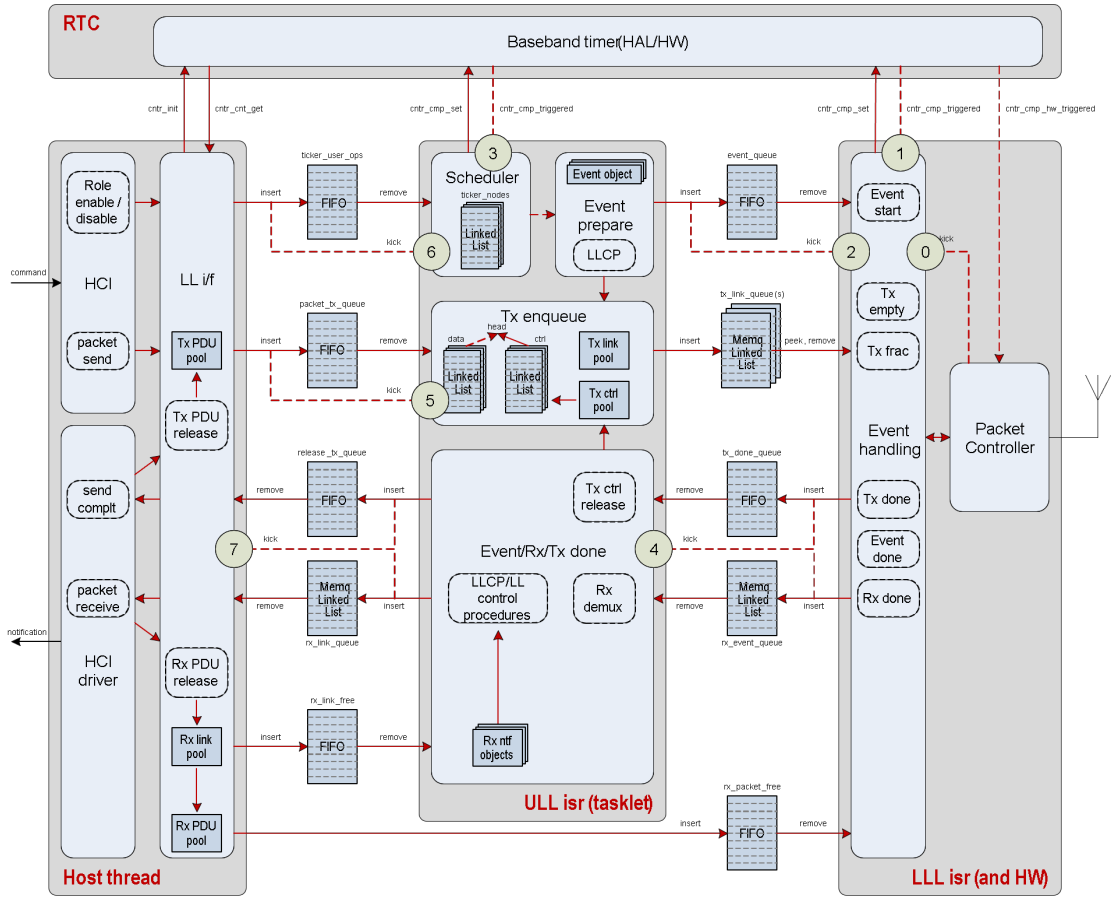


Transmit Data Flow



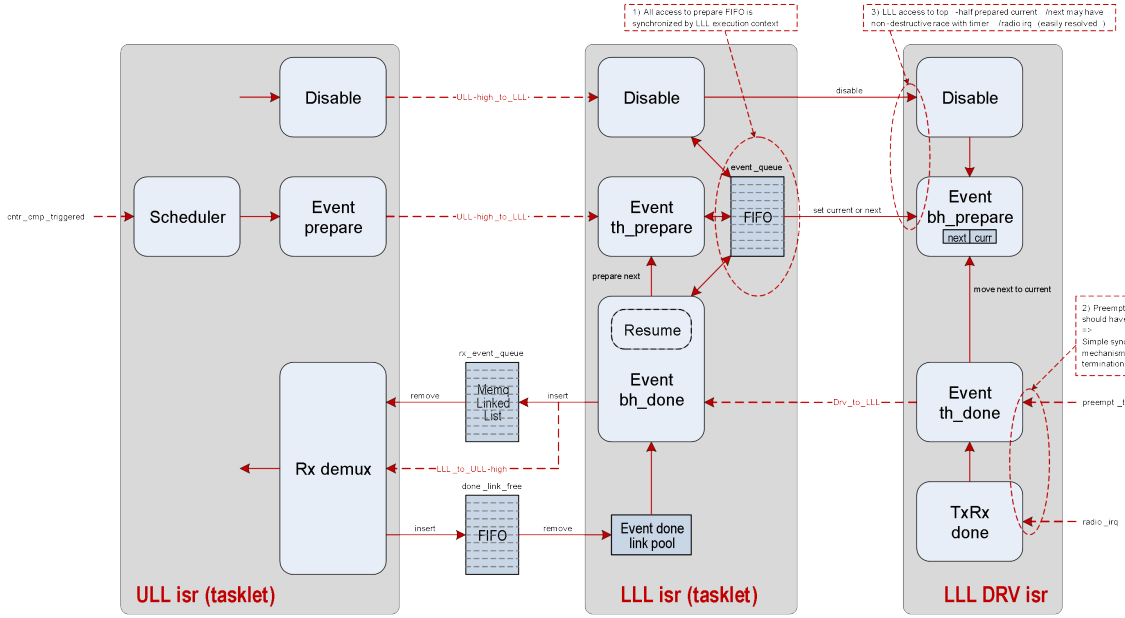
Receive Data Flow

Execution Priorities

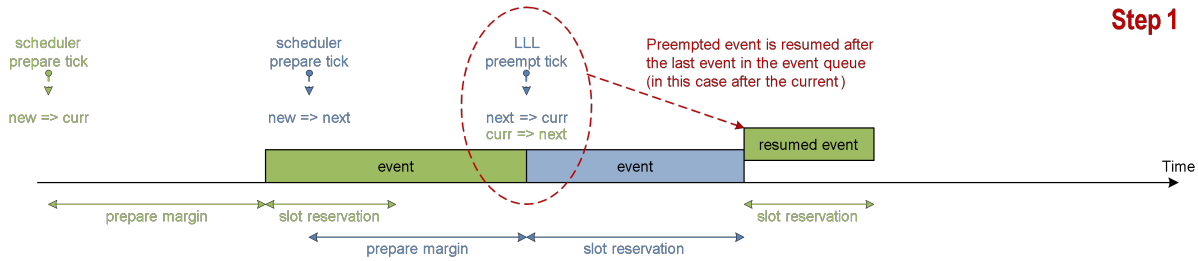


- Event handle (0, 1) < Event preparation (2, 3) < Event/Rx done (4) < Tx request (5) < Role management (6) < Host (7).
- LLL is vendor ISR, ULL is Mayfly ISR concept, Host is kernel thread.

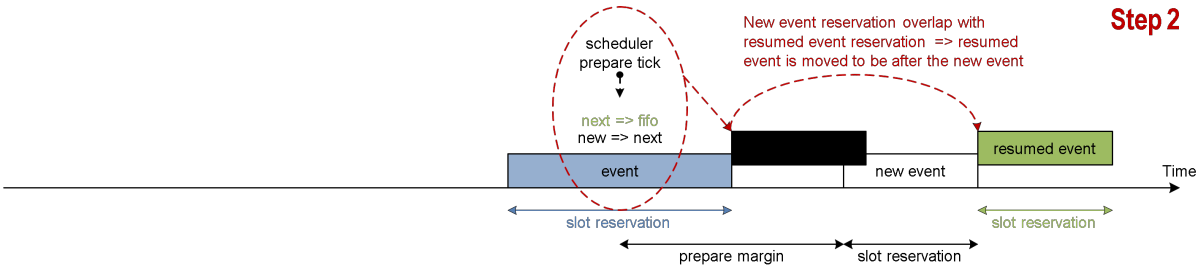
Lower Link Layer



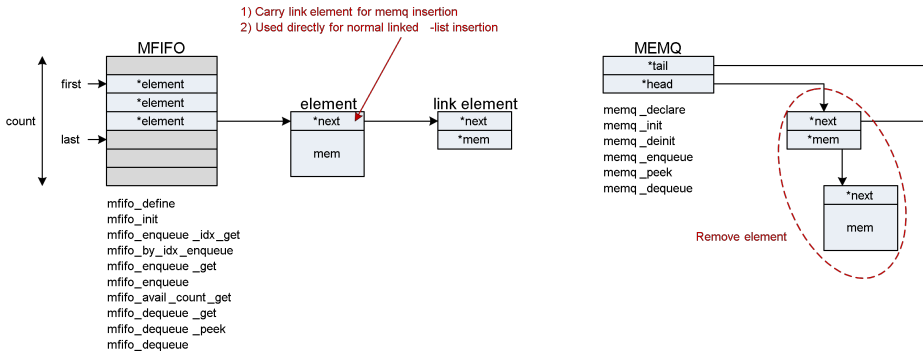
LLL Execution



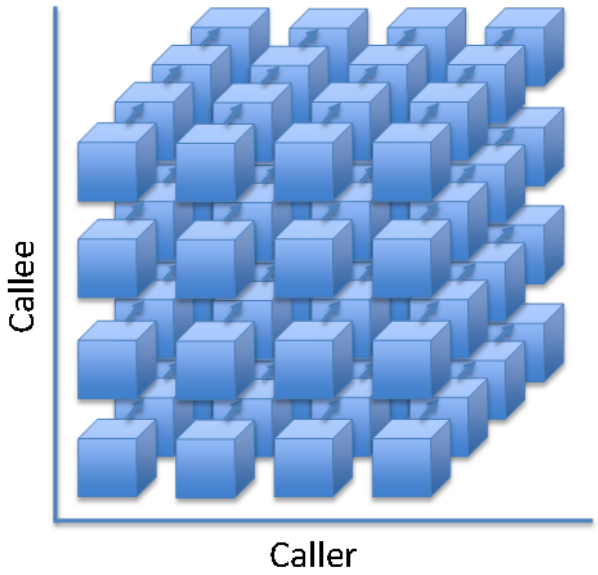
LLL Resume



Bare metal utilities



Memory FIFO and Memory Queue

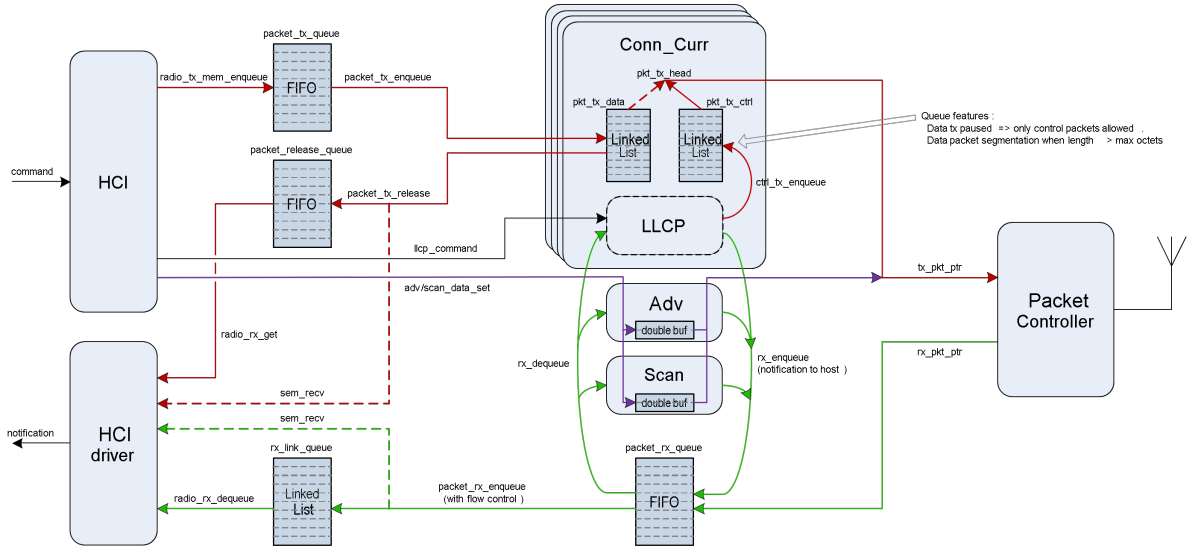


Mayfly

- Mayfly are multi-instance scalable ISR execution contexts

- What a Work is to a Thread, Mayfly is to an ISR
- List of functions executing in ISRs
- Execution priorities map to IRQ priorities
- Facilitate cross execution context scheduling
- Race-to-idle execution
- Lock-less, bare metal

Legacy Controller



Bluetooth Low Energy Controller - Vendor Specific Details

Hardware Requirements

Nordic Semiconductor The Nordic Semiconductor Bluetooth Low Energy Controller implementation requires the following hardware peripherals.

Table 13: SoC Peripheral Use

Resource	nRF Peripheral	# instances	Zephyr Driver Accessible	Description
Clock	NRF_CLOCK	1	Yes	• A Low Frequency Clock (LFCLOCK) or sleep clock, for low power consumption between Bluetooth radio events • A High Frequency Clock (HFCLOCK) or active clock, for high precision packet timing and software based transceiver state switching with inter-frame space (tIFS) timing inside Bluetooth radio events
RTC [a]	NRF_RTC0	1	**No**	• Uses 2 capture/compare registers
Timer	NRF_TIMER0 or NRF_TIMER4 and NRF_TIMER1	2 or 1 Page 1701, 1	**No**	• 2 instances, one each for packet timing and tIFS software switching, respectively • 7 capture/compare registers (3 mandatory, 1 optional for ISR profiling, 4 for single timer tIFS switching) on first instance • 4 capture/compare registers for second instance, if single tIFS timer is not used.
PPI [b]	NRF_PPI	21 channels (20[2]), and 2 channel groups[3]	Yes[4]	• Used for radio mode switching to achieve tIFS timings, for PA/LNA control
DPPI [c]	NRF_DPPI	20 channels, and 2 channel groups Page 17	Yes Page 1	• Used for radio mode switching to achieve tIFS timings, for PA/LNA control
SWI [d]	NRF_SWI4 and NRF_SWI5, or NRF_SWI2 and NRF_SWI3[5]	2	**No**	• 2 instances, for Lower Link Layer and Upper Link Layer Low priority execution context
Radio	NRF_RADIO	1	**No**	• 2.4 GHz radio transceiver with multiple radio standards such as 1 Mbps, 2 Mbps and Coded PHY S2/S8 Long Range Bluetooth Low Energy technology
RNG [e]	NRF_RNG	1	Yes	
ECB [f]	NRF_ECB	1	**No**	
CBC-CCM [g]	NRF_CCM	1	**No**	
AAR [h]	NRF_AAR	1	**No**	
GPIO [i]	NRF_GPIO	2 GPIO pins for PA and LNA, 1 each	Yes	• Additionally, 10 Debug GPIO pins (optional)
GPIOTE [j]	NRF_GPIOTE	1	Yes	

6.1.8 Application Development

Bluetooth applications are developed using the common infrastructure and approach that is described in the *Application Development* section of the documentation.

Additional information that is only relevant to Bluetooth applications can be found on this page.

- *Thread safety*
- *Hardware setup*
  - *Embedded*
  - *Host on Linux with an external Controller*
  - *Simulated nRF5x with BabbleSim*
- *Initialization*
- *Bluetooth Application Example*
- *More Examples*

Thread safety

Calling into the Bluetooth API is intended to be thread safe, unless otherwise noted in the documentation of the API function. The effort to ensure that this is the case for all API calls is an ongoing one, but the overall goal is formally stated in this paragraph. Bug reports and Pull Requests that move the subsystem in the direction of such goal are welcome.

Hardware setup

This section describes the options you have when building and debugging Bluetooth applications with Zephyr. Depending on the hardware that is available to you, the requirements you have and the type of development you prefer you may pick one or another setup to match your needs.

There are 3 possible setups:

1. *Embedded*
2. *External controller*
   - *QEMU host*
   - *native_sim host*
3. *Simulated nRF5x with BabbleSim*

Embedded This setup relies on all software running directly on the embedded platform(s) that the application is targeting. All the *Configurations* and *Build Types* are supported but you might need to build Zephyr more than once if you are using a dual-chip configuration or if you have multiple cores in your SoC each running a different build type (e.g., one running the Host, the other the Controller).

[1] `CONFIG_BT_CTLR_SW_SWITCH_SINGLE_TIMER=y`
[0] `CONFIG_BT_CTLR_TIFS_HW=n`
[2] When not using pre-defined PPI channels
[3] For software-based tIFS switching
[4] Drivers that use nRFx interfaces
[5] For nRF53x Series

To start developing using this setup follow the *Getting Started Guide*, choose one (or more if you are using a dual-chip solution) boards that support Bluetooth and then *run the application*).

There is a way to access the *HCI* traffic between the Host and Controller, even if there is no physical transport. See *Embedded HCI tracing* for instructions.

Note

This is currently only available on GNU/Linux

Host on Linux with an external Controller This setup relies on a "dual-chip" *configuration* which is comprised of the following devices:

1. A *Host-only* application running in the *QEMU* emulator or the native_sim native port of Zephyr
2. A Controller, which can be one of the following types:
   - A commercially available Controller
   - A *Controller-only* build of Zephyr
   - A *Virtual controller*

Warning

Certain external Controllers are either unable to accept the Host to Controller flow control parameters that Zephyr sets by default (Qualcomm), or do not transmit any data from the Controller to the Host (Realtek). If you see a message similar to:

```
<wrn> bt_hci_core: opcode 0x0c33 status 0x12
```

when booting your sample of choice (make sure you have enabled `CONFIG_LOG` in your `prj.conf` before running the sample), or if there is no data flowing from the Controller to the Host, then you need to disable Host to Controller flow control. To do so, set `CONFIG_BT_HCI_ACL_FLOW_CONTROL=n` in your `prj.conf`.

QEMU You can run the Zephyr Host on the *QEMU emulator* and have it interact with a physical external Bluetooth Controller.

Refer to *Running on QEMU or native_sim* for full instructions on how to build and run an application in this setup.

Note

This is currently only available on GNU/Linux

native_sim The native_sim target builds your Zephyr application with the Zephyr kernel, and some minimal HW emulation as a native Linux executable.

This executable is a normal Linux program, which can be debugged and instrumented like any other, and it communicates with a physical or virtual external Controller. Refer to:

- *Running on QEMU or native_sim* for the physical controller
- *Running on a Virtual Controller and native_sim* for the virtual controller

> **Note**
>
> This is currently only available on GNU/Linux

Simulated nRF5x with BabbleSim The nrf52_bsim and nrf5340bsim boards, are simulated target boards which emulate the necessary peripherals of a nRF52/53 SOC to be able to develop and test BLE applications. These boards, use:

- BabbleSim to simulate the nRF5x modem and the radio environment.
- The POSIX arch and native simulator to emulate the processor, and run natively on your host.
- Models of the nrf5x HW

Just like with the native_sim target, the build result is a normal Linux executable. You can find more information on how to run simulations with one or several devices in either of these boards's documentation.

With the nrf52_bsim, typically you do *Combined builds*, but it is also possible to build the controller with one of the HCI UART samples in one simulated device, and the host with the H4 driver instead of the integrated controller in another simulated device.

With the nrf5340bsim, you can build with either, both controller and host on its network core, or, with the network core running only the controller, the application core running the host and your application, and the HCI transport over IPC.

Initialization

The Bluetooth subsystem is initialized using the *bt_enable()* function. The caller should ensure that function succeeds by checking the return code for errors. If a function pointer is passed to *bt_enable()*, the initialization happens asynchronously, and the completion is notified through the given function.

Bluetooth Application Example

A simple Bluetooth beacon application is shown below. The application initializes the Bluetooth Subsystem and enables non-connectable advertising, effectively acting as a Bluetooth Low Energy broadcaster.

```

/*
 * Set Advertisement data. Based on the Eddystone specification:
 * https://github.com/google/eddystone/blob/master/protocol-specification.md
 * https://github.com/google/eddystone/tree/master/eddystone-url
 */
static const struct bt_data ad[] = {
        BT_DATA_BYTES(BT_DATA_FLAGS, BT_LE_AD_NO_BREDR),
        BT_DATA_BYTES(BT_DATA_UUID16_ALL, 0xaa, 0xfe),
        BT_DATA_BYTES(BT_DATA_SVC_DATA16,
                      0xaa, 0xfe, /* Eddystone UUID */
                      0x10, /* Eddystone-URL frame type */
                      0x00, /* Calibrated Tx power at 0m */
                      0x00, /* URL Scheme Prefix http://www. */
                      'z', 'e', 'p', 'h', 'y', 'r',
                      'p', 'r', 'o', 'j', 'e', 'c', 't',
                      0x08) /* .org */
};

```

(continues on next page)

(continued from previous page)

```
/* Set Scan Response data */
static const struct bt_data sd[] = {
        BT_DATA(BT_DATA_NAME_COMPLETE, DEVICE_NAME, DEVICE_NAME_LEN),
};

static void bt_ready(int err)
{
        char addr_s[BT_ADDR_LE_STR_LEN];
        bt_addr_le_t addr = {0};
        size_t count = 1;

        if (err) {
                printk("Bluetooth init failed (err %d)\n", err);
                return;
        }

        printk("Bluetooth initialized\n");

        /* Start advertising */
        err = bt_le_adv_start(BT_LE_ADV_NCONN_IDENTITY, ad, ARRAY_SIZE(ad),
                              sd, ARRAY_SIZE(sd));
        if (err) {
                printk("Advertising failed to start (err %d)\n", err);
                return;
        }


        /* For connectable advertising you would use
         * bt_le_oob_get_local().  For non-connectable non-identity
         * advertising an non-resolvable private address is used;
         * there is no API to retrieve that.
         */

        bt_id_get(&addr, &count);
        bt_addr_le_to_str(&addr, addr_s, sizeof(addr_s));

        printk("Beacon started, advertising as %s\n", addr_s);
}

int main(void)
{
        int err;

        printk("Starting Beacon Demo\n");

        /* Initialize the Bluetooth Subsystem */
        err = bt_enable(bt_ready);
        if (err) {
                printk("Bluetooth init failed (err %d)\n", err);
        }
        return 0;
}
```

The key APIs employed by the beacon sample are *bt_enable()* that's used to initialize Bluetooth and then *bt_le_adv_start()* that's used to start advertising a specific combination of advertising and scan response data.

More Examples

More sample Bluetooth applications are available in `samples/bluetooth/`.

6.1.9 API

Bluetooth Classic Host and profiles

Hands Free Profile (HFP)

API Reference

group `bt_hfp`

Hands Free Profile (HFP)

Defines

`HFP_HF_CMD_OK`

`HFP_HF_CMD_ERROR`

`HFP_HF_CMD_CME_ERROR`

`HFP_HF_CMD_UNKNOWN_ERROR`

Enums

enum `bt_hfp_hf_at_cmd`

Values:

enumerator `BT_HFP_HF_ATA`

enumerator `BT_HFP_HF_AT_CHUP`

Functions

int `bt_hfp_hf_register`(struct *bt_hfp_hf_cb* *cb)

Register HFP HF profile.

Register Handsfree profile callbacks to monitor the state and get the required HFP details to display.

Parameters

- `cb` – callback structure.

Returns

0 in case of success or negative value in case of error.

int `bt_hfp_hf_send_cmd`(struct bt_conn *conn, enum *bt_hfp_hf_at_cmd* cmd)

Handsfree client Send AT.

Send specific AT commands to handsfree client profile.

Parameters

- `conn` – Connection object.
- `cmd` – AT command to be sent.

Returns
0 in case of success or negative value in case of error.

struct `bt_hfp_hf_cmd_complete`

#include <*hfp_hf.h*> HFP HF Command completion field.

struct `bt_hfp_hf_cb`

#include <*hfp_hf.h*> HFP profile application callback.

Public Members

void (*`connected`)(struct bt_conn *conn)

HF connected callback to application.

If this callback is provided it will be called whenever the connection completes.

Param conn
Connection object.

void (*`disconnected`)(struct bt_conn *conn)

HF disconnected callback to application.

If this callback is provided it will be called whenever the connection gets disconnected, including when a connection gets rejected or cancelled or any error in SLC establishment.

Param conn
Connection object.

void (*`sco_connected`)(struct bt_conn *conn, struct bt_conn *sco_conn)

HF SCO/eSCO connected Callback.

If this callback is provided it will be called whenever the SCO/eSCO connection completes.

Param conn
Connection object.
Param sco_conn
SCO/eSCO Connection object.

void (*`sco_disconnected`)(struct bt_conn *sco_conn, uint8_t reason)

HF SCO/eSCO disconnected Callback.

If this callback is provided it will be called whenever the SCO/eSCO connection gets disconnected.

Param conn
Connection object.
Param reason
BT_HCI_ERR_* reason for the disconnection.

void (*`service`)(struct bt_conn *conn, uint32_t value)

HF indicator Callback.

This callback provides service indicator value to the application

Param conn
Connection object.

Param value
service indicator value received from the AG.

void (*`call`)(struct bt_conn *conn, uint32_t value)

HF indicator Callback.

This callback provides call indicator value to the application

Param conn
Connection object.

Param value
call indicator value received from the AG.

void (*`call_setup`)(struct bt_conn *conn, uint32_t value)

HF indicator Callback.

This callback provides call setup indicator value to the application

Param conn
Connection object.

Param value
call setup indicator value received from the AG.

void (*`call_held`)(struct bt_conn *conn, uint32_t value)

HF indicator Callback.

This callback provides call held indicator value to the application

Param conn
Connection object.

Param value
call held indicator value received from the AG.

void (*`signal`)(struct bt_conn *conn, uint32_t value)

HF indicator Callback.

This callback provides signal indicator value to the application

Param conn
Connection object.

Param value
signal indicator value received from the AG.

void (*`roam`)(struct bt_conn *conn, uint32_t value)

HF indicator Callback.

This callback provides roaming indicator value to the application

Param conn
Connection object.

Param value
roaming indicator value received from the AG.

void (*`battery`)(struct bt_conn *conn, uint32_t value)

HF indicator Callback.

This callback battery service indicator value to the application

Param conn
Connection object.

Param value
battery indicator value received from the AG.

void (*`ring_indication`)(struct bt_conn *conn)

HF incoming call Ring indication callback to application.

If this callback is provided it will be called whenever there is an incoming call.

Param conn
Connection object.

void (*`cmd_complete_cb`)(struct bt_conn *conn, struct *bt_hfp_hf_cmd_complete* *cmd)

HF notify command completed callback to application.

The command sent from the application is notified about its status

Param conn
Connection object.

Param cmd
structure contains status of the command including cme.

Serial Port Emulation (RFCOMM)

API Reference

group `bt_rfcomm`

RFCOMM.

Typedefs

typedef enum *bt_rfcomm_role* `bt_rfcomm_role_t`

Role of RFCOMM session and dlc.

Used only by internal APIs

Enums

Values:

enumerator `BT_RFCOMM_CHAN_HFP_HF` = 1

enumerator `BT_RFCOMM_CHAN_HFP_AG`

enumerator `BT_RFCOMM_CHAN_HSP_AG`

enumerator `BT_RFCOMM_CHAN_HSP_HS`

enumerator `BT_RFCOMM_CHAN_SPP`

enum `bt_rfcomm_role`

Role of RFCOMM session and dlc.

Used only by internal APIs

Values:

enumerator BT_RFCOMM_ROLE_ACCEPTOR

enumerator BT_RFCOMM_ROLE_INITIATOR

Functions

int **bt_rfcomm_server_register**(struct *bt_rfcomm_server* *server)

Register RFCOMM server.

Register RFCOMM server for a channel, each new connection is authorized using the *accept()* callback which in case of success shall allocate the dlc structure to be used by the new connection.

Parameters

- **server** – Server structure.

Returns

0 in case of success or negative value in case of error.

int **bt_rfcomm_dlc_connect**(struct bt_conn *conn, struct *bt_rfcomm_dlc* *dlc, uint8_t channel)

Connect RFCOMM channel.

Connect RFCOMM dlc by channel, once the connection is completed dlc connected() callback will be called. If the connection is rejected disconnected() callback is called instead.

Parameters

- **conn** – Connection object.
- **dlc** – Dlc object.
- **channel** – Server channel to connect to.

Returns

0 in case of success or negative value in case of error.

int **bt_rfcomm_dlc_send**(struct *bt_rfcomm_dlc* *dlc, struct *net_buf* *buf)

Send data to RFCOMM.

Send data from buffer to the dlc. Length should be less than or equal to mtu.

Parameters

- **dlc** – Dlc object.
- **buf** – Data buffer.

Returns

Bytes sent in case of success or negative value in case of error.

int **bt_rfcomm_dlc_disconnect**(struct *bt_rfcomm_dlc* *dlc)

Disconnect RFCOMM dlc.

Disconnect RFCOMM dlc, if the connection is pending it will be canceled and as a result the dlc disconnected() callback is called.

Parameters

- **dlc** – Dlc object.

Returns

0 in case of success or negative value in case of error.

struct *net_buf* *bt_rfcomm_create_pdu(struct *net_buf_pool* *pool)

Allocate the buffer from pool after reserving head room for RFCOMM, L2CAP and ACL headers.

Parameters

- pool – Which pool to take the buffer from.

Returns

New buffer.

struct bt_rfcomm_dlc_ops

#include <*rfcomm.h*> RFCOMM DLC operations structure.

Public Members

void (*connected)(struct *bt_rfcomm_dlc* *dlc)

DLC connected callback.

If this callback is provided it will be called whenever the connection completes.

Param dlc

The dlc that has been connected

void (*disconnected)(struct *bt_rfcomm_dlc* *dlc)

DLC disconnected callback.

If this callback is provided it will be called whenever the dlc is disconnected, including when a connection gets rejected or cancelled (both incoming and outgoing)

Param dlc

The dlc that has been Disconnected

void (*recv)(struct *bt_rfcomm_dlc* *dlc, struct *net_buf* *buf)

DLC recv callback.

Param dlc

The dlc receiving data.

Param buf

Buffer containing incoming data.

void (*sent)(struct *bt_rfcomm_dlc* *dlc, int err)

DLC sent callback.

Param dlc

The dlc which has sent data.

Param err

Sent result.

struct bt_rfcomm_dlc

#include <*rfcomm.h*> RFCOMM DLC structure.

struct bt_rfcomm_server

#include <*rfcomm.h*>

Public Members

uint8_t **channel**

Server Channel.

int (***accept**)(struct bt_conn *conn, struct *bt_rfcomm_dlc* **dlc)

Server accept callback.

This callback is called whenever a new incoming connection requires authorization.

Param conn
The connection that is requesting authorization

Param dlc
Pointer to received the allocated dlc

Return
0 in case of success or negative value in case of error.

Service Discovery Protocol (SDP)

API Reference

group bt_sdp

Service class identifiers of standard services and service groups

BT_SDP_SDP_SERVER_SVCLASS

Service Discovery Server.

BT_SDP_BROWSE_GRP_DESC_SVCLASS

Browse Group Descriptor.

BT_SDP_PUBLIC_BROWSE_GROUP

Public Browse Group.

BT_SDP_SERIAL_PORT_SVCLASS

Serial Port.

BT_SDP_LAN_ACCESS_SVCLASS

LAN Access Using PPP.

BT_SDP_DIALUP_NET_SVCLASS

Dialup Networking.

BT_SDP_IRMC_SYNC_SVCLASS

IrMC Sync.

BT_SDP_OBEX_OBJPUSH_SVCLASS

OBEX Object Push.

BT_SDP_OBEX_FILETRANS_SVCLASS

OBEX File Transfer.

BT_SDP_IRMC_SYNC_CMD_SVCLASS
: IrMC Sync Command.

BT_SDP_HEADSET_SVCLASS
: Headset.

BT_SDP_CORDLESS_TELEPHONY_SVCLASS
: Cordless Telephony.

BT_SDP_AUDIO_SOURCE_SVCLASS
: Audio Source.

BT_SDP_AUDIO_SINK_SVCLASS
: Audio Sink.

BT_SDP_AV_REMOTE_TARGET_SVCLASS
: A/V Remote Control Target.

BT_SDP_ADVANCED_AUDIO_SVCLASS
: Advanced Audio Distribution.

BT_SDP_AV_REMOTE_SVCLASS
: A/V Remote Control.

BT_SDP_AV_REMOTE_CONTROLLER_SVCLASS
: A/V Remote Control Controller.

BT_SDP_INTERCOM_SVCLASS
: Intercom.

BT_SDP_FAX_SVCLASS
: Fax.

BT_SDP_HEADSET_AGW_SVCLASS
: Headset AG.

BT_SDP_WAP_SVCLASS
: WAP.

BT_SDP_WAP_CLIENT_SVCLASS
: WAP Client.

BT_SDP_PANU_SVCLASS
: Personal Area Networking User.

BT_SDP_NAP_SVCLASS
: Network Access Point.

`BT_SDP_GN_SVCLASS`
: Group Network.

`BT_SDP_DIRECT_PRINTING_SVCLASS`
: Direct Printing.

`BT_SDP_REFERENCE_PRINTING_SVCLASS`
: Reference Printing.

`BT_SDP_IMAGING_SVCLASS`
: Basic Imaging Profile.

`BT_SDP_IMAGING_RESPONDER_SVCLASS`
: Imaging Responder.

`BT_SDP_IMAGING_ARCHIVE_SVCLASS`
: Imaging Automatic Archive.

`BT_SDP_IMAGING_REFOBJS_SVCLASS`
: Imaging Referenced Objects.

`BT_SDP_HANDSFREE_SVCLASS`
: Handsfree.

`BT_SDP_HANDSFREE_AGW_SVCLASS`
: Handsfree Audio Gateway.

`BT_SDP_DIRECT_PRT_REFOBJS_SVCLASS`
: Direct Printing Reference Objects Service.

`BT_SDP_REFLECTED_UI_SVCLASS`
: Reflected UI.

`BT_SDP_BASIC_PRINTING_SVCLASS`
: Basic Printing.

`BT_SDP_PRINTING_STATUS_SVCLASS`
: Printing Status.

`BT_SDP_HID_SVCLASS`
: Human Interface Device Service.

`BT_SDP_HCR_SVCLASS`
: Hardcopy Cable Replacement.

`BT_SDP_HCR_PRINT_SVCLASS`
: HCR Print.

`BT_SDP_HCR_SCAN_SVCLASS`
: HCR Scan.

`BT_SDP_CIP_SVCLASS`
: Common ISDN Access.

`BT_SDP_VIDEO_CONF_GW_SVCLASS`
: Video Conferencing Gateway.

`BT_SDP_UDI_MT_SVCLASS`
: UDI MT.

`BT_SDP_UDI_TA_SVCLASS`
: UDI TA.

`BT_SDP_AV_SVCLASS`
: Audio/Video.

`BT_SDP_SAP_SVCLASS`
: SIM Access.

`BT_SDP_PBAP_PCE_SVCLASS`
: Phonebook Access Client.

`BT_SDP_PBAP_PSE_SVCLASS`
: Phonebook Access Server.

`BT_SDP_PBAP_SVCLASS`
: Phonebook Access.

`BT_SDP_MAP_MSE_SVCLASS`
: Message Access Server.

`BT_SDP_MAP_MCE_SVCLASS`
: Message Notification Server.

`BT_SDP_MAP_SVCLASS`
: Message Access Profile.

`BT_SDP_GNSS_SVCLASS`
: GNSS.

`BT_SDP_GNSS_SERVER_SVCLASS`
: GNSS Server.

`BT_SDP_MPS_SC_SVCLASS`
: MPS SC.

BT_SDP_MPS_SVCLASS
: MPS.

BT_SDP_PNP_INFO_SVCLASS
: PnP Information.

BT_SDP_GENERIC_NETWORKING_SVCLASS
: Generic Networking.

BT_SDP_GENERIC_FILETRANS_SVCLASS
: Generic File Transfer.

BT_SDP_GENERIC_AUDIO_SVCLASS
: Generic Audio.

BT_SDP_GENERIC_TELEPHONY_SVCLASS
: Generic Telephony.

BT_SDP_UPNP_SVCLASS
: UPnP Service.

BT_SDP_UPNP_IP_SVCLASS
: UPnP IP Service.

BT_SDP_UPNP_PAN_SVCLASS
: UPnP IP PAN.

BT_SDP_UPNP_LAP_SVCLASS
: UPnP IP LAP.

BT_SDP_UPNP_L2CAP_SVCLASS
: UPnP IP L2CAP.

BT_SDP_VIDEO_SOURCE_SVCLASS
: Video Source.

BT_SDP_VIDEO_SINK_SVCLASS
: Video Sink.

BT_SDP_VIDEO_DISTRIBUTION_SVCLASS
: Video Distribution.

BT_SDP_HDP_SVCLASS
: HDP.

BT_SDP_HDP_SOURCE_SVCLASS
: HDP Source.

`BT_SDP_HDP_SINK_SVCLASS`
HDP Sink.

`BT_SDP_GENERIC_ACCESS_SVCLASS`
Generic Access Profile.

`BT_SDP_GENERIC_ATTRIB_SVCLASS`
Generic Attribute Profile.

`BT_SDP_APPLE_AGENT_SVCLASS`
Apple Agent.

Attribute identifier codes

Possible values for attribute-id are listed below.

See SDP Spec, section “Service Attribute Definitions” for more details.

`BT_SDP_ATTR_RECORD_HANDLE`
Service Record Handle.

`BT_SDP_ATTR_SVCLASS_ID_LIST`
Service Class ID List.

`BT_SDP_ATTR_RECORD_STATE`
Service Record State.

`BT_SDP_ATTR_SERVICE_ID`
Service ID.

`BT_SDP_ATTR_PROTO_DESC_LIST`
Protocol Descriptor List.

`BT_SDP_ATTR_BROWSE_GRP_LIST`
Browse Group List.

`BT_SDP_ATTR_LANG_BASE_ATTR_ID_LIST`
Language Base Attribute ID List.

`BT_SDP_ATTR_SVCINFO_TTL`
Service Info Time to Live.

`BT_SDP_ATTR_SERVICE_AVAILABILITY`
Service Availability.

`BT_SDP_ATTR_PROFILE_DESC_LIST`
Bluetooth Profile Descriptor List.

BT_SDP_ATTR_DOC_URL
: Documentation URL.

BT_SDP_ATTR_CLNT_EXEC_URL
: Client Executable URL.

BT_SDP_ATTR_ICON_URL
: Icon URL.

BT_SDP_ATTR_ADD_PROTO_DESC_LIST
: Additional Protocol Descriptor List.

BT_SDP_ATTR_GROUP_ID
: Group ID.

BT_SDP_ATTR_IP_SUBNET
: IP Subnet.

BT_SDP_ATTR_VERSION_NUM_LIST
: Version Number List.

BT_SDP_ATTR_SUPPORTED_FEATURES_LIST
: Supported Features List.

BT_SDP_ATTR_GOEP_L2CAP_PSM
: GOEP L2CAP PSM.

BT_SDP_ATTR_SVCDB_STATE
: Service Database State.

BT_SDP_ATTR_MPSD_SCENARIOS
: MPSD Scenarios.

BT_SDP_ATTR_MPMD_SCENARIOS
: MPMD Scenarios.

BT_SDP_ATTR_MPS_DEPENDENCIES
: Supported Profiles & Protocols.

BT_SDP_ATTR_SERVICE_VERSION
: Service Version.

BT_SDP_ATTR_EXTERNAL_NETWORK
: External Network.

BT_SDP_ATTR_SUPPORTED_DATA_STORES_LIST
: Supported Data Stores List.

BT_SDP_ATTR_DATA_EXCHANGE_SPEC
: Data Exchange Specification.

BT_SDP_ATTR_NETWORK
: Network.

BT_SDP_ATTR_FAX_CLASS1_SUPPORT
: Fax Class 1 Support.

BT_SDP_ATTR_REMOTE_AUDIO_VOLUME_CONTROL
: Remote Audio Volume Control.

BT_SDP_ATTR_MCAP_SUPPORTED_PROCEDURES
: MCAP Supported Procedures.

BT_SDP_ATTR_FAX_CLASS20_SUPPORT
: Fax Class 2.0 Support.

BT_SDP_ATTR_SUPPORTED_FORMATS_LIST
: Supported Formats List.

BT_SDP_ATTR_FAX_CLASS2_SUPPORT
: Fax Class 2 Support (vendor-specific)

BT_SDP_ATTR_AUDIO_FEEDBACK_SUPPORT
: Audio Feedback Support.

BT_SDP_ATTR_NETWORK_ADDRESS
: Network Address.

BT_SDP_ATTR_WAP_GATEWAY
: WAP Gateway.

BT_SDP_ATTR_HOMEPAGE_URL
: Homepage URL.

BT_SDP_ATTR_WAP_STACK_TYPE
: WAP Stack Type.

BT_SDP_ATTR_SECURITY_DESC
: Security Description.

BT_SDP_ATTR_NET_ACCESS_TYPE
: Net Access Type.

BT_SDP_ATTR_MAX_NET_ACCESSRATE
: Max Net Access Rate.

`BT_SDP_ATTR_IP4_SUBNET`
IPv4 Subnet.

`BT_SDP_ATTR_IP6_SUBNET`
IPv6 Subnet.

`BT_SDP_ATTR_SUPPORTED_CAPABILITIES`
BIP Supported Capabilities.

`BT_SDP_ATTR_SUPPORTED_FEATURES`
BIP Supported Features.

`BT_SDP_ATTR_SUPPORTED_FUNCTIONS`
BIP Supported Functions.

`BT_SDP_ATTR_TOTAL_IMAGING_DATA_CAPACITY`
BIP Total Imaging Data Capacity.

`BT_SDP_ATTR_SUPPORTED_REPOSITORIES`
Supported Repositories.

`BT_SDP_ATTR_MAS_INSTANCE_ID`
MAS Instance ID.

`BT_SDP_ATTR_SUPPORTED_MESSAGE_TYPES`
Supported Message Types.

`BT_SDP_ATTR_PBAP_SUPPORTED_FEATURES`
PBAP Supported Features.

`BT_SDP_ATTR_MAP_SUPPORTED_FEATURES`
MAP Supported Features.

`BT_SDP_ATTR_SPECIFICATION_ID`
Specification ID.

`BT_SDP_ATTR_VENDOR_ID`
Vendor ID.

`BT_SDP_ATTR_PRODUCT_ID`
Product ID.

`BT_SDP_ATTR_VERSION`
Version.

`BT_SDP_ATTR_PRIMARY_RECORD`
Primary Record.

BT_SDP_ATTR_VENDOR_ID_SOURCE
: Vendor ID Source.

BT_SDP_ATTR_HID_DEVICE_RELEASE_NUMBER
: HID Device Release Number.

BT_SDP_ATTR_HID_PARSER_VERSION
: HID Parser Version.

BT_SDP_ATTR_HID_DEVICE_SUBCLASS
: HID Device Subclass.

BT_SDP_ATTR_HID_COUNTRY_CODE
: HID Country Code.

BT_SDP_ATTR_HID_VIRTUAL_CABLE
: HID Virtual Cable.

BT_SDP_ATTR_HID_RECONNECT_INITIATE
: HID Reconnect Initiate.

BT_SDP_ATTR_HID_DESCRIPTOR_LIST
: HID Descriptor List.

BT_SDP_ATTR_HID_LANG_ID_BASE_LIST
: HID Language ID Base List.

BT_SDP_ATTR_HID_SDP_DISABLE
: HID SDP Disable.

BT_SDP_ATTR_HID_BATTERY_POWER
: HID Battery Power.

BT_SDP_ATTR_HID_REMOTE_WAKEUP
: HID Remote Wakeup.

BT_SDP_ATTR_HID_PROFILE_VERSION
: HID Profile Version.

BT_SDP_ATTR_HID_SUPERVISION_TIMEOUT
: HID Supervision Timeout.

BT_SDP_ATTR_HID_NORMALLY_CONNECTABLE
: HID Normally Connectable.

BT_SDP_ATTR_HID_BOOT_DEVICE
: HID Boot Device.

The Data representation in SDP PDUs (pps 339, 340 of BT SDP Spec)

These are the exact data type+size descriptor values that go into the PDU buffer.

The datatype (leading 5bits) + size descriptor (last 3 bits) is 8 bits. The size descriptor is critical to extract the right number of bytes for the data value from the PDU.

For most basic types, the datatype+size descriptor is straightforward. However for constructed types and strings, the size of the data is in the next "n" bytes following the 8 bits (datatype+size) descriptor. Exactly what the "n" is specified in the 3 bits of the data size descriptor.

TextString and URLString can be of size 2^{8, 16, 32} bytes DataSequence and DataSequenceAlternates can be of size 2^{8, 16, 32} The size are computed post-facto in the API and are not known apriori.

`BT_SDP_DATA_NIL`
: Nil, the null type.

`BT_SDP_UINT8`
: Unsigned 8-bit integer.

`BT_SDP_UINT16`
: Unsigned 16-bit integer.

`BT_SDP_UINT32`
: Unsigned 32-bit integer.

`BT_SDP_UINT64`
: Unsigned 64-bit integer.

`BT_SDP_UINT128`
: Unsigned 128-bit integer.

`BT_SDP_INT8`
: Signed 8-bit integer.

`BT_SDP_INT16`
: Signed 16-bit integer.

`BT_SDP_INT32`
: Signed 32-bit integer.

`BT_SDP_INT64`
: Signed 64-bit integer.

`BT_SDP_INT128`
: Signed 128-bit integer.

`BT_SDP_UUID_UNSPEC`
: UUID, unspecified size.

BT_SDP_UUID16
: UUID, 16-bit.

BT_SDP_UUID32
: UUID, 32-bit.

BT_SDP_UUID128
: UUID, 128-bit.

BT_SDP_TEXT_STR_UNSPEC
: Text string, unspecified size.

BT_SDP_TEXT_STR8
: Text string, 8-bit length.

BT_SDP_TEXT_STR16
: Text string, 16-bit length.

BT_SDP_TEXT_STR32
: Text string, 32-bit length.

BT_SDP_BOOL
: Boolean.

BT_SDP_SEQ_UNSPEC
: Data element sequence, unspecified size.

BT_SDP_SEQ8
: Data element sequence, 8-bit length.

BT_SDP_SEQ16
: Data element sequence, 16-bit length.

BT_SDP_SEQ32
: Data element sequence, 32-bit length.

BT_SDP_ALT_UNSPEC
: Data element alternative, unspecified size.

BT_SDP_ALT8
: Data element alternative, 8-bit length.

BT_SDP_ALT16
: Data element alternative, 16-bit length.

BT_SDP_ALT32
: Data element alternative, 32-bit length.

BT_SDP_URL_STR_UNSPEC
: URL string, unspecified size.

BT_SDP_URL_STR8
: URL string, 8-bit length.

BT_SDP_URL_STR16
: URL string, 16-bit length.

BT_SDP_URL_STR32
: URL string, 32-bit length.

Defines

BT_SDP_SERVER_RECORD_HANDLE

BT_SDP_PRIMARY_LANG_BASE

BT_SDP_ATTR_SVCNAME_PRIMARY

BT_SDP_ATTR_SVCDESC_PRIMARY

BT_SDP_ATTR_PROVNAME_PRIMARY

BT_SDP_TYPE_DESC_MASK

BT_SDP_SIZE_DESC_MASK

BT_SDP_SIZE_INDEX_OFFSET

BT_SDP_ARRAY_8(...)
: Declare an array of 8-bit elements in an attribute.

BT_SDP_ARRAY_16(...)
: Declare an array of 16-bit elements in an attribute.

BT_SDP_ARRAY_32(...)
: Declare an array of 32-bit elements in an attribute.

BT_SDP_TYPE_SIZE(_type)
: Declare a fixed-size data element header.

  Parameters

  - **_type** – Data element header containing type and size descriptors.

BT_SDP_TYPE_SIZE_VAR(_type, _size)
: Declare a variable-size data element header.

  Parameters

  - **_type** – Data element header containing type and size descriptors.
  - **_size** – The actual size of the data.

BT_SDP_DATA_ELEM_LIST(...)
Declare a list of data elements.

BT_SDP_NEW_SERVICE
SDP New Service Record Declaration Macro.

Helper macro to declare a new service record. Default attributes: Record Handle, Record State, Language Base, Root Browse Group

BT_SDP_LIST(_att_id, _type_size, _data_elem_seq)
Generic SDP List Attribute Declaration Macro.

Helper macro to declare a list attribute.

Parameters

- **_att_id** – List Attribute ID.
- **_data_elem_seq** – Data element sequence for the list.
- **_type_size** – SDP type and size descriptor.

BT_SDP_SERVICE_ID(_uuid)
SDP Service ID Attribute Declaration Macro.

Helper macro to declare a service ID attribute.

Parameters

- **_uuid** – Service ID 16bit UUID.

BT_SDP_SERVICE_NAME(_name)
SDP Name Attribute Declaration Macro.

Helper macro to declare a service name attribute.

Parameters

- **_name** – Service name as a string (up to 256 chars).

BT_SDP_SUPPORTED_FEATURES(_features)
SDP Supported Features Attribute Declaration Macro.

Helper macro to declare supported features of a profile/protocol.

Parameters

- **_features** – Feature mask as 16bit unsigned integer.

BT_SDP_RECORD(_attrs)
SDP Service Declaration Macro.

Helper macro to declare a service.

Parameters

- **_attrs** – List of attributes for the service record.

Typedefs

typedef uint8_t (*bt_sdp_discover_func_t)(struct bt_conn *conn, struct *bt_sdp_client_result* *result)
Callback type reporting to user that there is a resolved result on remote for given UUID and the result record buffer can be used by user for further inspection.

A function of this type is given by the user to the *bt_sdp_discover_params* object. It'll be called on each valid record discovery completion for given UUID. When UUID resolution gives back no records then NULL is passed to the user. Otherwise user can get valid record(s) and then the internal hint 'next record' is set to false saying the UUID resolution is complete or the hint can be set by caller to true meaning that next record is available for given UUID. The returned function value allows the user to control retrieving follow-up resolved records if any. If the user doesn't want to read more resolved records for given UUID since current record data fulfills its requirements then should return BT_SDP_DISCOVER_UUID_STOP. Otherwise returned value means more subcall iterations are allowable.

Param conn
Connection object identifying connection to queried remote.

Param result
Object pointing to logical unparsed SDP record collected on base of response driven by given UUID.

Return
BT_SDP_DISCOVER_UUID_STOP in case of no more need to read next record data and continue discovery for given UUID. By returning BT_SDP_DISCOVER_UUID_CONTINUE user allows this discovery continuation.

Enums

Helper enum to be used as return value of bt_sdp_discover_func_t.

The value informs the caller to perform further pending actions or stop them.

Values:

enumerator **BT_SDP_DISCOVER_UUID_STOP** = 0

enumerator **BT_SDP_DISCOVER_UUID_CONTINUE**

enum **bt_sdp_proto**

Protocols to be asked about specific parameters.

Values:

enumerator **BT_SDP_PROTO_RFCOMM** = 0x0003

enumerator **BT_SDP_PROTO_L2CAP** = 0x0100

Functions

int **bt_sdp_register_service**(struct *bt_sdp_record* *service)

Register a Service Record.

Register a Service Record. Applications can make use of macros such as BT_SDP_DECLARE_SERVICE, BT_SDP_LIST, BT_SDP_SERVICE_ID, BT_SDP_SERVICE_NAME, etc. A service declaration must start with BT_SDP_NEW_SERVICE.

Parameters

- `service` – Service record declared using BT_SDP_DECLARE_SERVICE.

Returns
0 in case of success or negative value in case of error.

int `bt_sdp_discover`(struct bt_conn *conn, const struct *bt_sdp_discover_params* *params)

Allows user to start SDP discovery session.

The function performs SDP service discovery on remote server driven by user delivered discovery parameters. Discovery session is made as soon as no SDP transaction is ongoing between peers and if any then this one is queued to be processed at discovery completion of previous one. On the service discovery completion the callback function will be called to get feedback to user about findings.

Parameters

- `conn` – Object identifying connection to remote.
- `params` – SDP discovery parameters.

Returns
0 in case of success or negative value in case of error.

int `bt_sdp_discover_cancel`(struct bt_conn *conn, const struct *bt_sdp_discover_params* *params)

Release waiting SDP discovery request.

It can cancel valid waiting SDP client request identified by SDP discovery parameters object.

Parameters

- `conn` – Object identifying connection to remote.
- `params` – SDP discovery parameters.

Returns
0 in case of success or negative value in case of error.

int `bt_sdp_get_proto_param`(const struct *net_buf* *buf, enum *bt_sdp_proto* proto, uint16_t *param)

Give to user parameter value related to given stacked protocol UUID.

API extracts specific parameter associated with given protocol UUID available in Protocol Descriptor List attribute.

Parameters

- `buf` – Original buffered raw record data.
- `proto` – Known protocol to be checked like RFCOMM or L2CAP.
- `param` – On success populated by found parameter value.

Returns
0 on success when specific parameter associated with given protocol value is found, or negative if error occurred during processing.

int `bt_sdp_get_addl_proto_param`(const struct *net_buf* *buf, enum *bt_sdp_proto* proto, uint8_t param_index, uint16_t *param)

Get additional parameter value related to given stacked protocol UUID.

API extracts specific parameter associated with given protocol UUID available in Additional Protocol Descriptor List attribute.

Parameters

- `buf` – Original buffered raw record data.

- `proto` – Known protocol to be checked like RFCOMM or L2CAP.
- `param_index` – There may be more than one parameter related to the given protocol UUID. This function returns the result that is indexed by this parameter. It's value is from 0, 0 means the first matched result, 1 means the second matched result.
- `param` – **[out]** On success populated by found parameter value.

Returns
0 on success when a specific parameter associated with a given protocol value is found, or negative if error occurred during processing.

int `bt_sdp_get_profile_version`(const struct *net_buf* *buf, uint16_t profile, uint16_t *version)

Get profile version.

Helper API extracting remote profile version number. To get it proper generic profile parameter needs to be selected usually listed in SDP Interoperability Requirements section for given profile specification.

Parameters

- `buf` – Original buffered raw record data.
- `profile` – Profile family identifier the profile belongs.
- `version` – On success populated by found version number.

Returns
0 on success, negative value if error occurred during processing.

int `bt_sdp_get_features`(const struct *net_buf* *buf, uint16_t *features)

Get SupportedFeatures attribute value.

Allows if exposed by remote retrieve SupportedFeature attribute.

Parameters

- `buf` – Buffer holding original raw record data from remote.
- `features` – On success object to be populated with SupportedFeature mask.

Returns
0 on success if feature found and valid, negative in case any error

struct `bt_sdp_data_elem`

#include <*sdp.h*> SDP Generic Data Element Value.

Public Members

uint8_t `type`

Type of the data element.

uint32_t `data_size`

Size of the data element.

uint32_t `total_size`

Total size of the data element.

struct **bt_sdp_attribute**

#include <*sdp.h*> SDP Attribute Value.

Public Members

uint16_t **id**

Attribute ID.

struct *bt_sdp_data_elem* **val**

Attribute data.

struct **bt_sdp_record**

#include <*sdp.h*> SDP Service Record Value.

Public Members

uint32_t **handle**

Redundant, for quick ref.

struct *bt_sdp_attribute* ***attrs**

Base addr of attr array.

size_t **attr_count**

Number of attributes.

uint8_t **index**

Index of the record in LL.

struct *bt_sdp_record* ***next**

Next service record.

struct **bt_sdp_client_result**

#include <*sdp.h*> Generic SDP Client Query Result data holder.

Public Members

struct *net_buf* ***resp_buf**

buffer containing unparsed SDP record result for given UUID

bool **next_record_hint**

flag pointing that there are more result chunks for given UUID

const struct *bt_uuid* ***uuid**

Reference to UUID object on behalf one discovery was started.

struct **bt_sdp_discover_params**

#include <*sdp.h*> Main user structure used in SDP discovery of remote.

Public Members

const struct *bt_uuid* *uuid
: UUID (service) to be discovered on remote SDP entity.

bt_sdp_discover_func_t func
: Discover callback to be called on resolved SDP record.

struct *net_buf_pool* *pool
: Memory buffer enabled by user for SDP query results

Bluetooth LE Audio

Bluetooth Audio

API Reference

group bt_audio
: Bluetooth Audio.

Unicast Announcement Type

BT_AUDIO_UNICAST_ANNOUNCEMENT_GENERAL
: Unicast Server is connectable and is requesting a connection.

BT_AUDIO_UNICAST_ANNOUNCEMENT_TARGETED
: Unicast Server is connectable but is not requesting a connection.

Defines

BT_AUDIO_BROADCAST_ID_SIZE
: Size of the broadcast ID in octets.

BT_AUDIO_BROADCAST_ID_MAX
: Maximum broadcast ID value.

BT_AUDIO_PD_PREF_NONE
: Indicates that the server have no preference for the presentation delay.

BT_AUDIO_PD_MAX
: Maximum presentation delay in microseconds.

BT_AUDIO_BROADCAST_CODE_SIZE
: Maximum size of the broadcast code in octets.

`BT_AUDIO_BROADCAST_NAME_LEN_MIN`
: The minimum size of a Broadcast Name as defined by Bluetooth Assigned Numbers.

`BT_AUDIO_BROADCAST_NAME_LEN_MAX`
: The maximum size of a Broadcast Name as defined by Bluetooth Assigned Numbers.

`BT_AUDIO_LANG_SIZE`
: Size of the stream language value, e.g.

  “eng”

`BT_AUDIO_CODEC_CAP_CHAN_COUNT_MIN`
: Minimum supported channel counts.

`BT_AUDIO_CODEC_CAP_CHAN_COUNT_MAX`
: Maximum supported channel counts.

`BT_AUDIO_CODEC_CAP_CHAN_COUNT_SUPPORT(...)`
: Channel count support capability.

  Macro accepts variable number of channel counts. The allowed channel counts are defined by specification and have to be in range from *BT_AUDIO_CODEC_CAP_CHAN_COUNT_MIN* to *BT_AUDIO_CODEC_CAP_CHAN_COUNT_MAX* inclusive.

  Example to support 1 and 3 channels: *BT_AUDIO_CODEC_CAP_CHAN_COUNT_SUPPORT(1, 3)*

`BT_AUDIO_CONTEXT_TYPE_ANY`
: Any known context.

`BT_AUDIO_METADATA_TYPE_IS_KNOWN(_type)`
: Helper to check whether metadata type is known by the stack.

  Note

  `_type` is evaluated thrice.

`BT_AUDIO_CODEC_DATA(_type, _bytes...)`
: Helper to declare elements of *bt_audio_codec_cap* arrays.

  This macro is mainly for creating an array of struct *bt_audio_codec_cap* data arrays.

  Parameters

  - **`_type`** – Type of advertising data field
  - **`_bytes`** – Variable number of single-byte parameters

`BT_AUDIO_CODEC_CFG(_id, _cid, _vid, _data, _meta)`
: Helper to declare *Codec config parsing APIs*.

  Parameters

  - **`_id`** – Codec ID
  - **`_cid`** – Company ID
  - **`_vid`** – Vendor ID

- **_data** – Codec Specific Data in LVT format
- **_meta** – Codec Specific Metadata in LVT format

BT_AUDIO_CODEC_CAP(_id, _cid, _vid, _data, _meta)

Helper to declare Codec capability parsing APIs structure.

Parameters

- **_id** – Codec ID
- **_cid** – Company ID
- **_vid** – Vendor ID
- **_data** – Codec Specific Data in LVT format
- **_meta** – Codec Specific Metadata in LVT format

BT_AUDIO_LOCATION_ANY

Any known location.

BT_AUDIO_CODEC_QOS(_interval, _framing, _phy, _sdu, _rtn, _latency, _pd)

Helper to declare elements of *bt_audio_codec_qos*.

Parameters

- **_interval** – SDU interval (usec)
- **_framing** – Framing
- **_phy** – Target PHY
- **_sdu** – Maximum SDU Size
- **_rtn** – Retransmission number
- **_latency** – Maximum Transport Latency (msec)
- **_pd** – Presentation Delay (usec)

BT_AUDIO_CODEC_QOS_UNFRAMED(_interval, _sdu, _rtn, _latency, _pd)

Helper to declare Input Unframed *bt_audio_codec_qos*.

Parameters

- **_interval** – SDU interval (usec)
- **_sdu** – Maximum SDU Size
- **_rtn** – Retransmission number
- **_latency** – Maximum Transport Latency (msec)
- **_pd** – Presentation Delay (usec)

BT_AUDIO_CODEC_QOS_FRAMED(_interval, _sdu, _rtn, _latency, _pd)

Helper to declare Input Framed *bt_audio_codec_qos*.

Parameters

- **_interval** – SDU interval (usec)
- **_sdu** – Maximum SDU Size
- **_rtn** – Retransmission number
- **_latency** – Maximum Transport Latency (msec)
- **_pd** – Presentation Delay (usec)

BT_AUDIO_CODEC_QOS_PREF(_unframed_supported, _phy, _rtn, _latency, _pd_min, _pd_max, _pref_pd_min, _pref_pd_max)

Helper to declare elements of *bt_audio_codec_qos_pref*.

Parameters

- **_unframed_supported** – Unframed PDUs supported
- **_phy** – Preferred Target PHY
- **_rtn** – Preferred Retransmission number
- **_latency** – Preferred Maximum Transport Latency (msec)
- **_pd_min** – Minimum Presentation Delay (usec)
- **_pd_max** – Maximum Presentation Delay (usec)
- **_pref_pd_min** – Preferred Minimum Presentation Delay (usec)
- **_pref_pd_max** – Preferred Maximum Presentation Delay (usec)

Enums

enum **bt_audio_codec_cap_type**

Codec capability types.

Used to build and parse codec capabilities as specified in the PAC specification. Source is assigned numbers for Generic Audio, bluetooth.com.

Values:

enumerator **BT_AUDIO_CODEC_CAP_TYPE_FREQ** = 0x01

Supported sampling frequencies.

enumerator **BT_AUDIO_CODEC_CAP_TYPE_DURATION** = 0x02

Supported frame durations.

enumerator **BT_AUDIO_CODEC_CAP_TYPE_CHAN_COUNT** = 0x03

Supported audio channel counts.

enumerator **BT_AUDIO_CODEC_CAP_TYPE_FRAME_LEN** = 0x04

Supported octets per codec frame.

enumerator **BT_AUDIO_CODEC_CAP_TYPE_FRAME_COUNT** = 0x05

Supported maximum codec frames per SDU

enum **bt_audio_codec_cap_freq**

Supported frequencies bitfield.

Values:

enumerator **BT_AUDIO_CODEC_CAP_FREQ_8KHZ** = *BIT*(0)

8 Khz sampling frequency

enumerator **BT_AUDIO_CODEC_CAP_FREQ_11KHZ** = *BIT*(1)

11.025 Khz sampling frequency

enumerator BT_AUDIO_CODEC_CAP_FREQ_16KHZ = *BIT*(2)
16 Khz sampling frequency

enumerator BT_AUDIO_CODEC_CAP_FREQ_22KHZ = *BIT*(3)
22.05 Khz sampling frequency

enumerator BT_AUDIO_CODEC_CAP_FREQ_24KHZ = *BIT*(4)
24 Khz sampling frequency

enumerator BT_AUDIO_CODEC_CAP_FREQ_32KHZ = *BIT*(5)
32 Khz sampling frequency

enumerator BT_AUDIO_CODEC_CAP_FREQ_44KHZ = *BIT*(6)
44.1 Khz sampling frequency

enumerator BT_AUDIO_CODEC_CAP_FREQ_48KHZ = *BIT*(7)
48 Khz sampling frequency

enumerator BT_AUDIO_CODEC_CAP_FREQ_88KHZ = *BIT*(8)
88.2 Khz sampling frequency

enumerator BT_AUDIO_CODEC_CAP_FREQ_96KHZ = *BIT*(9)
96 Khz sampling frequency

enumerator BT_AUDIO_CODEC_CAP_FREQ_176KHZ = *BIT*(10)
176.4 Khz sampling frequency

enumerator BT_AUDIO_CODEC_CAP_FREQ_192KHZ = *BIT*(11)
192 Khz sampling frequency

enumerator BT_AUDIO_CODEC_CAP_FREQ_384KHZ = *BIT*(12)
384 Khz sampling frequency

enumerator BT_AUDIO_CODEC_CAP_FREQ_ANY = (*BT_AUDIO_CODEC_CAP_FREQ_8KHZ* | *BT_AUDIO_CODEC_CAP_FREQ_11KHZ* | *BT_AUDIO_CODEC_CAP_FREQ_16KHZ* | *BT_AUDIO_CODEC_CAP_FREQ_22KHZ* | *BT_AUDIO_CODEC_CAP_FREQ_24KHZ* | *BT_AUDIO_CODEC_CAP_FREQ_32KHZ* | *BT_AUDIO_CODEC_CAP_FREQ_44KHZ* | *BT_AUDIO_CODEC_CAP_FREQ_48KHZ* | *BT_AUDIO_CODEC_CAP_FREQ_88KHZ* | *BT_AUDIO_CODEC_CAP_FREQ_96KHZ* | *BT_AUDIO_CODEC_CAP_FREQ_176KHZ* | *BT_AUDIO_CODEC_CAP_FREQ_192KHZ* | *BT_AUDIO_CODEC_CAP_FREQ_384KHZ*)
Any frequency capability.

enum bt_audio_codec_cap_frame_dur
Supported frame durations bitfield.

Values:

enumerator BT_AUDIO_CODEC_CAP_DURATION_7_5 = *BIT*(0)
7.5 msec frame duration capability

enumerator BT_AUDIO_CODEC_CAP_DURATION_10 = *BIT*(1)

10 msec frame duration capability

enumerator BT_AUDIO_CODEC_CAP_DURATION_ANY = (*BT_AUDIO_CODEC_CAP_DURATION_7_5* | *BT_AUDIO_CODEC_CAP_DURATION_10*)

Any frame duration capability.

enumerator BT_AUDIO_CODEC_CAP_DURATION_PREFER_7_5 = *BIT*(4)

7.5 msec preferred frame duration capability.

This shall only be set if *BT_AUDIO_CODEC_CAP_DURATION_7_5* is also set, and if *BT_AUDIO_CODEC_CAP_DURATION_PREFER_10* is not set.

enumerator BT_AUDIO_CODEC_CAP_DURATION_PREFER_10 = *BIT*(5)

10 msec preferred frame duration capability

This shall only be set if *BT_AUDIO_CODEC_CAP_DURATION_10* is also set, and if *BT_AUDIO_CODEC_CAP_DURATION_PREFER_7_5* is not set.

enum bt_audio_codec_cap_chan_count

Supported audio capabilities channel count bitfield.

Values:

enumerator BT_AUDIO_CODEC_CAP_CHAN_COUNT_1 = *BIT*(0)

Supporting 1 channel.

enumerator BT_AUDIO_CODEC_CAP_CHAN_COUNT_2 = *BIT*(1)

Supporting 2 channel.

enumerator BT_AUDIO_CODEC_CAP_CHAN_COUNT_3 = *BIT*(2)

Supporting 3 channel.

enumerator BT_AUDIO_CODEC_CAP_CHAN_COUNT_4 = *BIT*(3)

Supporting 4 channel.

enumerator BT_AUDIO_CODEC_CAP_CHAN_COUNT_5 = *BIT*(4)

Supporting 5 channel.

enumerator BT_AUDIO_CODEC_CAP_CHAN_COUNT_6 = *BIT*(5)

Supporting 6 channel.

enumerator BT_AUDIO_CODEC_CAP_CHAN_COUNT_7 = *BIT*(6)

Supporting 7 channel.

enumerator BT_AUDIO_CODEC_CAP_CHAN_COUNT_8 = *BIT*(7)

Supporting 8 channel.

enumerator BT_AUDIO_CODEC_CAP_CHAN_COUNT_ANY = (*BT_AUDIO_CODEC_CAP_CHAN_COUNT_1* | *BT_AUDIO_CODEC_CAP_CHAN_COUNT_2* | *BT_AUDIO_CODEC_CAP_CHAN_COUNT_3* | *BT_AUDIO_CODEC_CAP_CHAN_COUNT_4* | *BT_AUDIO_CODEC_CAP_CHAN_COUNT_5* | *BT_AUDIO_CODEC_CAP_CHAN_COUNT_6* | *BT_AUDIO_CODEC_CAP_CHAN_COUNT_7* | *BT_AUDIO_CODEC_CAP_CHAN_COUNT_8*)

Supporting all channels.

enum **bt_audio_codec_cfg_type**

Codec configuration types.

Used to build and parse codec configurations as specified in the ASCS and BAP specifications. Source is assigned numbers for Generic Audio, bluetooth.com.

Values:

enumerator **BT_AUDIO_CODEC_CFG_FREQ** = 0x01

Sampling frequency.

enumerator **BT_AUDIO_CODEC_CFG_DURATION** = 0x02

Frame duration.

enumerator **BT_AUDIO_CODEC_CFG_CHAN_ALLOC** = 0x03

Audio channel allocation.

enumerator **BT_AUDIO_CODEC_CFG_FRAME_LEN** = 0x04

Octets per codec frame.

enumerator **BT_AUDIO_CODEC_CFG_FRAME_BLKS_PER_SDU** = 0x05

Codec frame blocks per SDU.

enum **bt_audio_codec_cfg_freq**

Codec configuration sampling freqency.

Values:

enumerator **BT_AUDIO_CODEC_CFG_FREQ_8KHZ** = 0x01

8 Khz codec sampling frequency

enumerator **BT_AUDIO_CODEC_CFG_FREQ_11KHZ** = 0x02

11.025 Khz codec sampling frequency

enumerator **BT_AUDIO_CODEC_CFG_FREQ_16KHZ** = 0x03

16 Khz codec sampling frequency

enumerator **BT_AUDIO_CODEC_CFG_FREQ_22KHZ** = 0x04

22.05 Khz codec sampling frequency

enumerator **BT_AUDIO_CODEC_CFG_FREQ_24KHZ** = 0x05

24 Khz codec sampling frequency

enumerator **BT_AUDIO_CODEC_CFG_FREQ_32KHZ** = 0x06

32 Khz codec sampling frequency

enumerator **BT_AUDIO_CODEC_CFG_FREQ_44KHZ** = 0x07

44.1 Khz codec sampling frequency

enumerator BT_AUDIO_CODEC_CFG_FREQ_48KHZ = 0x08

48 Khz codec sampling frequency

enumerator BT_AUDIO_CODEC_CFG_FREQ_88KHZ = 0x09

88.2 Khz codec sampling frequency

enumerator BT_AUDIO_CODEC_CFG_FREQ_96KHZ = 0x0a

96 Khz codec sampling frequency

enumerator BT_AUDIO_CODEC_CFG_FREQ_176KHZ = 0x0b

176.4 Khz codec sampling frequency

enumerator BT_AUDIO_CODEC_CFG_FREQ_192KHZ = 0x0c

192 Khz codec sampling frequency

enumerator BT_AUDIO_CODEC_CFG_FREQ_384KHZ = 0x0d

384 Khz codec sampling frequency

enum bt_audio_codec_cfg_frame_dur

Codec configuration frame duration.

Values:

enumerator BT_AUDIO_CODEC_CFG_DURATION_7_5 = 0x00

7.5 msec Frame Duration configuration

enumerator BT_AUDIO_CODEC_CFG_DURATION_10 = 0x01

10 msec Frame Duration configuration

enum bt_audio_context

Audio Context Type for Generic Audio.

These values are defined by the Generic Audio Assigned Numbers, bluetooth.com

Values:

enumerator BT_AUDIO_CONTEXT_TYPE_PROHIBITED = 0

Prohibited.

enumerator BT_AUDIO_CONTEXT_TYPE_UNSPECIFIED = *BIT*(0)

Identifies audio where the use case context does not match any other defined value, or where the context is unknown or cannot be determined.

enumerator BT_AUDIO_CONTEXT_TYPE_CONVERSATIONAL = *BIT*(1)

Conversation between humans, for example, in telephony or video calls, including traditional cellular as well as VoIP and Push-to-Talk.

enumerator BT_AUDIO_CONTEXT_TYPE_MEDIA = *BIT*(2)

Media, for example, music playback, radio, podcast or movie soundtrack, or tv audio.

enumerator BT_AUDIO_CONTEXT_TYPE_GAME = *BIT*(3)

Audio associated with video gaming, for example gaming media; gaming effects; music and in-game voice chat between participants; or a mix of all the above.

enumerator BT_AUDIO_CONTEXT_TYPE_INSTRUCTIONAL = *BIT*(4)

Instructional audio, for example, in navigation, announcements, or user guidance.

enumerator BT_AUDIO_CONTEXT_TYPE_VOICE_ASSISTANTS = *BIT*(5)

Man-machine communication, for example, with voice recognition or virtual assistants.

enumerator BT_AUDIO_CONTEXT_TYPE_LIVE = *BIT*(6)

Live audio, for example, from a microphone where audio is perceived both through a direct acoustic path and through an LE Audio Stream.

enumerator BT_AUDIO_CONTEXT_TYPE_SOUND_EFFECTS = *BIT*(7)

Sound effects including keyboard and touch feedback; menu and user interface sounds; and other system sounds.

enumerator BT_AUDIO_CONTEXT_TYPE_NOTIFICATIONS = *BIT*(8)

Notification and reminder sounds; attention-seeking audio, for example, in beeps signaling the arrival of a message.

enumerator BT_AUDIO_CONTEXT_TYPE_RINGTONE = *BIT*(9)

Alerts the user to an incoming call, for example, an incoming telephony or video call, including traditional cellular as well as VoIP and Push-to-Talk.

enumerator BT_AUDIO_CONTEXT_TYPE_ALERTS = *BIT*(10)

Alarms and timers; immediate alerts, for example, in a critical battery alarm, timer expiry or alarm clock, toaster, cooker, kettle, microwave, etc.

enumerator BT_AUDIO_CONTEXT_TYPE_EMERGENCY_ALARM = *BIT*(11)

Emergency alarm Emergency sounds, for example, fire alarms or other urgent alerts.

enum bt_audio_parental_rating

Parental rating defined by the Generic Audio assigned numbers (bluetooth.com).

The numbering scheme is aligned with Annex F of EN 300 707 v1.2.1 which defined parental rating for viewing.

Values:

enumerator BT_AUDIO_PARENTAL_RATING_NO_RATING = 0x00

No rating.

enumerator BT_AUDIO_PARENTAL_RATING_AGE_ANY = 0x01

For all ages.

enumerator BT_AUDIO_PARENTAL_RATING_AGE_5_OR_ABOVE = 0x02

Recommended for listeners of age 5 and above.

enumerator BT_AUDIO_PARENTAL_RATING_AGE_6_OR_ABOVE = 0x03
Recommended for listeners of age 6 and above.

enumerator BT_AUDIO_PARENTAL_RATING_AGE_7_OR_ABOVE = 0x04
Recommended for listeners of age 7 and above.

enumerator BT_AUDIO_PARENTAL_RATING_AGE_8_OR_ABOVE = 0x05
Recommended for listeners of age 8 and above.

enumerator BT_AUDIO_PARENTAL_RATING_AGE_9_OR_ABOVE = 0x06
Recommended for listeners of age 9 and above.

enumerator BT_AUDIO_PARENTAL_RATING_AGE_10_OR_ABOVE = 0x07
Recommended for listeners of age 10 and above.

enumerator BT_AUDIO_PARENTAL_RATING_AGE_11_OR_ABOVE = 0x08
Recommended for listeners of age 11 and above.

enumerator BT_AUDIO_PARENTAL_RATING_AGE_12_OR_ABOVE = 0x09
Recommended for listeners of age 12 and above.

enumerator BT_AUDIO_PARENTAL_RATING_AGE_13_OR_ABOVE = 0x0A
Recommended for listeners of age 13 and above.

enumerator BT_AUDIO_PARENTAL_RATING_AGE_14_OR_ABOVE = 0x0B
Recommended for listeners of age 14 and above.

enumerator BT_AUDIO_PARENTAL_RATING_AGE_15_OR_ABOVE = 0x0C
Recommended for listeners of age 15 and above.

enumerator BT_AUDIO_PARENTAL_RATING_AGE_16_OR_ABOVE = 0x0D
Recommended for listeners of age 16 and above.

enumerator BT_AUDIO_PARENTAL_RATING_AGE_17_OR_ABOVE = 0x0E
Recommended for listeners of age 17 and above.

enumerator BT_AUDIO_PARENTAL_RATING_AGE_18_OR_ABOVE = 0x0F
Recommended for listeners of age 18 and above.

enum bt_audio_active_state
Audio Active State defined by the Generic Audio assigned numbers (bluetooth.com).
Values:

enumerator BT_AUDIO_ACTIVE_STATE_DISABLED = 0x00
No audio data is being transmitted.

enumerator BT_AUDIO_ACTIVE_STATE_ENABLED = 0x01
Audio data is being transmitted.

enum **bt_audio_metadata_type**

Codec metadata type IDs.

Metadata types defined by the Generic Audio assigned numbers (bluetooth.com).

Values:

enumerator **BT_AUDIO_METADATA_TYPE_PREF_CONTEXT** = 0x01

Preferred audio context.

Bitfield of preferred audio contexts.

If 0, the context type is not a preferred use case for this codec configuration.

See the BT_AUDIO_CONTEXT_* for valid values.

enumerator **BT_AUDIO_METADATA_TYPE_STREAM_CONTEXT** = 0x02

Streaming audio context.

Bitfield of streaming audio contexts.

If 0, the context type is not a preferred use case for this codec configuration.

See the BT_AUDIO_CONTEXT_* for valid values.

enumerator **BT_AUDIO_METADATA_TYPE_PROGRAM_INFO** = 0x03

UTF-8 encoded title or summary of stream content.

enumerator **BT_AUDIO_METADATA_TYPE_LANG** = 0x04

Language.

3 octet lower case language code defined by ISO 639-3 Possible values can be found at https://iso639-3.sil.org/code_tables/639/data

enumerator **BT_AUDIO_METADATA_TYPE_CCID_LIST** = 0x05

Array of 8-bit CCID values.

enumerator **BT_AUDIO_METADATA_TYPE_PARENTAL_RATING** = 0x06

Parental rating.

See *bt_audio_parental_rating* for valid values.

enumerator **BT_AUDIO_METADATA_TYPE_PROGRAM_INFO_URI** = 0x07

UTF-8 encoded URI for additional Program information.

enumerator **BT_AUDIO_METADATA_TYPE_AUDIO_STATE** = 0x08

Audio active state.

See *bt_audio_active_state* for valid values.

enumerator **BT_AUDIO_METADATA_TYPE_BROADCAST_IMMEDIATE** = 0x09

Broadcast Audio Immediate Rendering flag

enumerator **BT_AUDIO_METADATA_TYPE_EXTENDED** = 0xFE

Extended metadata.

enumerator BT_AUDIO_METADATA_TYPE_VENDOR = 0xFF
Vendor specific metadata.

enum bt_audio_location
Location values for BT Audio.
These values are defined by the Generic Audio Assigned Numbers, bluetooth.com
Values:

enumerator BT_AUDIO_LOCATION_MONO_AUDIO = 0
Mono Audio (no specified Audio Location)

enumerator BT_AUDIO_LOCATION_FRONT_LEFT = *BIT*(0)
Front Left.

enumerator BT_AUDIO_LOCATION_FRONT_RIGHT = *BIT*(1)
Front Right.

enumerator BT_AUDIO_LOCATION_FRONT_CENTER = *BIT*(2)
Front Center.

enumerator BT_AUDIO_LOCATION_LOW_FREQ_EFFECTS_1 = *BIT*(3)
Low Frequency Effects 1.

enumerator BT_AUDIO_LOCATION_BACK_LEFT = *BIT*(4)
Back Left.

enumerator BT_AUDIO_LOCATION_BACK_RIGHT = *BIT*(5)
Back Right.

enumerator BT_AUDIO_LOCATION_FRONT_LEFT_OF_CENTER = *BIT*(6)
Front Left of Center.

enumerator BT_AUDIO_LOCATION_FRONT_RIGHT_OF_CENTER = *BIT*(7)
Front Right of Center.

enumerator BT_AUDIO_LOCATION_BACK_CENTER = *BIT*(8)
Back Center.

enumerator BT_AUDIO_LOCATION_LOW_FREQ_EFFECTS_2 = *BIT*(9)
Low Frequency Effects 2.

enumerator BT_AUDIO_LOCATION_SIDE_LEFT = *BIT*(10)
Side Left.

enumerator BT_AUDIO_LOCATION_SIDE_RIGHT = *BIT*(11)
Side Right.

enumerator BT_AUDIO_LOCATION_TOP_FRONT_LEFT = *BIT*(12)
Top Front Left.

enumerator BT_AUDIO_LOCATION_TOP_FRONT_RIGHT = *BIT*(13)
Top Front Right.

enumerator BT_AUDIO_LOCATION_TOP_FRONT_CENTER = *BIT*(14)
Top Front Center.

enumerator BT_AUDIO_LOCATION_TOP_CENTER = *BIT*(15)
Top Center.

enumerator BT_AUDIO_LOCATION_TOP_BACK_LEFT = *BIT*(16)
Top Back Left.

enumerator BT_AUDIO_LOCATION_TOP_BACK_RIGHT = *BIT*(17)
Top Back Right.

enumerator BT_AUDIO_LOCATION_TOP_SIDE_LEFT = *BIT*(18)
Top Side Left.

enumerator BT_AUDIO_LOCATION_TOP_SIDE_RIGHT = *BIT*(19)
Top Side Right.

enumerator BT_AUDIO_LOCATION_TOP_BACK_CENTER = *BIT*(20)
Top Back Center.

enumerator BT_AUDIO_LOCATION_BOTTOM_FRONT_CENTER = *BIT*(21)
Bottom Front Center.

enumerator BT_AUDIO_LOCATION_BOTTOM_FRONT_LEFT = *BIT*(22)
Bottom Front Left.

enumerator BT_AUDIO_LOCATION_BOTTOM_FRONT_RIGHT = *BIT*(23)
Bottom Front Right.

enumerator BT_AUDIO_LOCATION_FRONT_LEFT_WIDE = *BIT*(24)
Front Left Wide.

enumerator BT_AUDIO_LOCATION_FRONT_RIGHT_WIDE = *BIT*(25)
Front Right Wide.

enumerator BT_AUDIO_LOCATION_LEFT_SURROUND = *BIT*(26)
Left Surround.

enumerator BT_AUDIO_LOCATION_RIGHT_SURROUND = *BIT*(27)
Right Surround.

enum **bt_audio_dir**

Audio direction from the perspective of the BAP Unicast Server / BAP Broadcast Sink.

Values:

enumerator **BT_AUDIO_DIR_SINK** = 0x01

Audio direction sink.

For a BAP Unicast Client or Broadcast Source this is considered outgoing audio (TX). For a BAP Unicast Server or Broadcast Sink this is considered incoming audio (RX).

enumerator **BT_AUDIO_DIR_SOURCE** = 0x02

Audio direction source.

For a BAP Unicast Client or Broadcast Source this is considered incoming audio (RX). For a BAP Unicast Server or Broadcast Sink this is considered outgoing audio (TX).

enum **bt_audio_codec_qos_framing**

Codec QoS Framing.

Values:

enumerator **BT_AUDIO_CODEC_QOS_FRAMING_UNFRAMED** = 0x00

Packets may be framed or unframed.

enumerator **BT_AUDIO_CODEC_QOS_FRAMING_FRAMED** = 0x01

Packets are always framed.

Codec QoS Preferred PHY.

Values:

enumerator **BT_AUDIO_CODEC_QOS_1M** = *BIT*(0)

LE 1M PHY.

enumerator **BT_AUDIO_CODEC_QOS_2M** = *BIT*(1)

LE 2M PHY.

enumerator **BT_AUDIO_CODEC_QOS_CODED** = *BIT*(2)

LE Coded PHY.

Functions

int **bt_audio_data_parse**(const uint8_t ltv[], size_t size, bool (*func)(struct *bt_data* *data, void *user_data), void *user_data)

Helper for parsing length-type-value data.

Parameters

- **ltv** – Length-type-value (LTV) encoded data.
- **size** – Size of the `ltv` data.

- `func` – Callback function which will be called for each element that's found in the data. The callback should return true to continue parsing, or false to stop parsing.
- `user_data` – User data to be passed to the callback.

Return values

- `0` – if all entries were parsed.
- `-EINVAL` – if the data is incorrectly encoded
- `-ECANCELED` – if parsing was prematurely cancelled by the callback

uint8_t **bt_audio_get_chan_count**(enum *bt_audio_location* chan_allocation)

Function to get the number of channels from the channel allocation.

Parameters

- `chan_allocation` – The channel allocation

Returns

The number of channels

struct **bt_audio_codec_octets_per_codec_frame**

#include <*audio.h*> struct to hold minimum and maximum supported codec frame sizes

Public Members

uint16_t **min**

Minimum number of octets supported per codec frame.

uint16_t **max**

Maximum number of octets supported per codec frame.

struct **bt_audio_codec_cap**

#include <*audio.h*> Codec capability structure.

Public Members

uint8_t **path_id**

Data path ID.

BT_ISO_DATA_PATH_HCI for HCI path, or any other value for vendor specific ID.

bool **ctlr_transcode**

Whether or not the local controller should transcode.

This effectively sets the coding format for the ISO data path to BT_HCI_CODING_FORMAT_TRANSPARENT if false, else uses the *bt_audio_codec_cfg::id*.

uint8_t **id**

Codec ID.

uint16_t **cid**
Codec Company ID.

uint16_t **vid**
Codec Company Vendor ID.

size_t **data_len**
Codec Specific Capabilities Data count.

uint8_t **data**[CONFIG_BT_AUDIO_CODEC_CAP_MAX_DATA_SIZE]
Codec Specific Capabilities Data.

size_t **meta_len**
Codec Specific Capabilities Metadata count.

uint8_t **meta**[CONFIG_BT_AUDIO_CODEC_CAP_MAX_METADATA_SIZE]
Codec Specific Capabilities Metadata.

struct **bt_audio_codec_cfg**
#include <*audio.h*> Codec specific configuration structure.

Public Members

uint8_t **path_id**
Data path ID.

BT_ISO_DATA_PATH_HCI for HCI path, or any other value for vendor specific ID.

bool **ctlr_transcode**
Whether or not the local controller should transcode.

This effectively sets the coding format for the ISO data path to BT_HCI_CODING_FORMAT_TRANSPARENT if false, else uses the *bt_audio_codec_cfg::id*.

uint8_t **id**
Codec ID.

uint16_t **cid**
Codec Company ID.

uint16_t **vid**
Codec Company Vendor ID.

size_t **data_len**
Codec Specific Capabilities Data count.

uint8_t **data**[CONFIG_BT_AUDIO_CODEC_CFG_MAX_DATA_SIZE]
Codec Specific Capabilities Data.

size_t **meta_len**

Codec Specific Capabilities Metadata count.

uint8_t **meta**[CONFIG_BT_AUDIO_CODEC_CFG_MAX_METADATA_SIZE]

Codec Specific Capabilities Metadata.

struct **bt_audio_codec_qos**

#include <*audio.h*> Codec QoS structure.

Public Members

uint32_t **pd**

Presentation Delay in microseconds.

This value can be changed up and until *bt_bap_stream_qos()* has been called. Once a stream has been QoS configured, modifying this field does not modify the value. It is however possible to modify this field and call *bt_bap_stream_qos()* again to update the value, assuming that the stream is in the correct state.

Value range 0 to *BT_AUDIO_PD_MAX*.

enum *bt_audio_codec_qos_framing* **framing**

QoS Framing.

uint8_t **phy**

PHY.

Allowed values are *BT_AUDIO_CODEC_QOS_1M*, *BT_AUDIO_CODEC_QOS_2M* and *BT_AUDIO_CODEC_QOS_CODED*.

uint8_t **rtn**

Retransmission Number.

This a recommendation to the controller, and the actual retransmission number may be different than this.

uint16_t **sdu**

Maximum SDU size.

Value range BT_ISO_MIN_SDU to BT_ISO_MAX_SDU.

uint16_t **latency**

Maximum Transport Latency.

Not used for the `CONFIG_BT_BAP_BROADCAST_SINK` role.

uint32_t **interval**

SDU Interval.

Value range BT_ISO_SDU_INTERVAL_MIN to BT_ISO_SDU_INTERVAL_MAX

uint16_t **max_pdu**

Maximum PDU size.

Maximum size, in octets, of the payload from link layer to link layer.

Value range BT_ISO_CONNECTED_PDU_MIN to BT_ISO_PDU_MAX for connected ISO.

Value range BT_ISO_BROADCAST_PDU_MIN to BT_ISO_PDU_MAX for broadcast ISO.

uint8_t **burst_number**

Burst number.

Value range BT_ISO_BN_MIN to BT_ISO_BN_MAX.

uint8_t **num_subevents**

Number of subevents.

Maximum number of subevents in each CIS or BIS event.

Value range BT_ISO_NSE_MIN to BT_ISO_NSE_MAX.

struct bt_audio_codec_qos

Connected Isochronous Group (CIG) parameters.

The fields in this struct affect the value sent to the controller via HCI when creating the CIG. Once the group has been created with *bt_bap_unicast_group_create()*, modifying these fields will not affect the group.

struct **bt_audio_codec_qos_pref**

#include <audio.h> Audio Stream Quality of Service Preference structure.

Public Members

bool **unframed_supported**

Unframed PDUs supported.

Unlike the other fields, this is not a preference but whether the codec supports unframed ISOAL PDUs.

uint8_t **phy**

Preferred PHY.

uint8_t **rtn**

Preferred Retransmission Number.

uint16_t **latency**

Preferred Transport Latency.

uint32_t **pd_min**

Minimum Presentation Delay in microseconds.

Unlike the other fields, this is not a preference but a minimum requirement.

Value range 0 to *BT_AUDIO_PD_MAX*, or *BT_AUDIO_PD_PREF_NONE* to indicate no preference.

uint32_t **pd_max**

Maximum Presentation Delay.

Unlike the other fields, this is not a preference but a maximum requirement.

Value range 0 to *BT_AUDIO_PD_MAX*, or *BT_AUDIO_PD_PREF_NONE* to indicate no preference.

uint32_t **pref_pd_min**

Preferred minimum Presentation Delay.

Value range 0 to *BT_AUDIO_PD_MAX*.

uint32_t **pref_pd_max**

Preferred maximum Presentation Delay.

Value range 0 to *BT_AUDIO_PD_MAX*.

group **bt_audio_codec_cfg**

Audio codec Config APIs.

Functions to parse codec config data when formatted as LTV wrapped into *Codec config parsing APIs*.

Functions

int **bt_audio_codec_cfg_freq_to_freq_hz**(enum *bt_audio_codec_cfg_freq* freq)

Convert assigned numbers frequency to frequency value.

Parameters

- **freq** – The assigned numbers frequency to convert.

Return values

- **-EINVAL** – if arguments are invalid.
- **The** – converted frequency value in Hz.

int **bt_audio_codec_cfg_freq_hz_to_freq**(uint32_t freq_hz)

Convert frequency value to assigned numbers frequency.

Parameters

- **freq_hz** – The frequency value to convert.

Return values

- **-EINVAL** – if arguments are invalid.
- **The** – assigned numbers frequency (*bt_audio_codec_cfg_freq*).

int **bt_audio_codec_cfg_get_freq**(const struct *bt_audio_codec_cfg* *codec_cfg)

Extract the frequency from a codec configuration.

Parameters

- **codec_cfg** – The codec configuration to extract data from.

Return values

- **A** – *bt_audio_codec_cfg_freq* value

- **-EINVAL** – if arguments are invalid
- **-ENODATA** – if not found
- **-EBADMSG** – if found value has invalid size or value

int **bt_audio_codec_cfg_set_freq**(struct *bt_audio_codec_cfg* *codec_cfg, enum *bt_audio_codec_cfg_freq* freq)

Set the frequency of a codec configuration.

Parameters

- **codec_cfg** – The codec configuration to set data for.
- **freq** – The assigned numbers frequency to set.

Return values

- **The** – data_len of codec_cfg on success
- **-EINVAL** – if arguments are invalid
- **-ENOMEM** – if the new value could not set or added due to memory

int **bt_audio_codec_cfg_frame_dur_to_frame_dur_us**(enum *bt_audio_codec_cfg_frame_dur* frame_dur)

Convert assigned numbers frame duration to duration in microseconds.

Parameters

- **frame_dur** – The assigned numbers frame duration to convert.

Return values

- **-EINVAL** – if arguments are invalid.
- **The** – converted frame duration value in microseconds.

int **bt_audio_codec_cfg_frame_dur_us_to_frame_dur**(uint32_t frame_dur_us)

Convert frame duration in microseconds to assigned numbers frame duration.

Parameters

- **frame_dur_us** – The frame duration in microseconds to convert.

Return values

- **-EINVAL** – if arguments are invalid.
- **The** – assigned numbers frame duration (*bt_audio_codec_cfg_frame_dur*).

int **bt_audio_codec_cfg_get_frame_dur**(const struct *bt_audio_codec_cfg* *codec_cfg)

Extract frame duration from BT codec config.

Parameters

- **codec_cfg** – The codec configuration to extract data from.

Return values

- **A** – *bt_audio_codec_cfg_frame_dur* value
- **-EINVAL** – if arguments are invalid
- **-ENODATA** – if not found
- **-EBADMSG** – if found value has invalid size or value

int **bt_audio_codec_cfg_set_frame_dur**(struct *bt_audio_codec_cfg* *codec_cfg, enum *bt_audio_codec_cfg_frame_dur* frame_dur)

Set the frame duration of a codec configuration.

Parameters

- **codec_cfg** – The codec configuration to set data for.
- **frame_dur** – The assigned numbers frame duration to set.

Return values

- **The** – data_len of codec_cfg on success
- **-EINVAL** – if arguments are invalid
- **-ENOMEM** – if the new value could not set or added due to memory

int **bt_audio_codec_cfg_get_chan_allocation**(const struct *bt_audio_codec_cfg* *codec_cfg, enum *bt_audio_location* *chan_allocation, bool fallback_to_default)

Extract channel allocation from BT codec config.

The value returned is a bit field representing one or more audio locations as specified by *bt_audio_location* Shall match one or more of the bits set in BT_PAC_SNK_LOC/BT_PAC_SRC_LOC.

Up to the configured *BT_AUDIO_CODEC_CAP_TYPE_CHAN_COUNT* number of channels can be present.

Parameters

- **codec_cfg** – The codec configuration to extract data from.
- **chan_allocation** – Pointer to the variable to store the extracted value in.
- **fallback_to_default** – If true this function will provide the default value of *BT_AUDIO_LOCATION_MONO_AUDIO* if the type is not found when codec_cfg.id is BT_HCI_CODING_FORMAT_LC3.

Return values

- **0** – if value is found and stored in the pointer provided
- **-EINVAL** – if arguments are invalid
- **-ENODATA** – if not found
- **-EBADMSG** – if found value has invalid size or value

int **bt_audio_codec_cfg_set_chan_allocation**(struct *bt_audio_codec_cfg* *codec_cfg, enum *bt_audio_location* chan_allocation)

Set the channel allocation of a codec configuration.

Parameters

- **codec_cfg** – The codec configuration to set data for.
- **chan_allocation** – The channel allocation to set.

Return values

- **The** – data_len of codec_cfg on success
- **-EINVAL** – if arguments are invalid
- **-ENOMEM** – if the new value could not set or added due to memory

int **bt_audio_codec_cfg_get_octets_per_frame**(const struct *bt_audio_codec_cfg* *codec_cfg)

Extract frame size in octets from BT codec config.

The overall SDU size will be octets_per_frame * blocks_per_sdu.

The Bluetooth specifications are not clear about this value - it does not state that the codec shall use this SDU size only. A codec like LC3 supports variable bit-rate (per SDU) hence it might be allowed for an encoder to reduce the frame size below this value. Hence it is recommended to use the received SDU size and divide by blocks_per_sdu rather than relying on this octets_per_sdu value to be fixed.

Parameters

- **codec_cfg** – The codec configuration to extract data from.

Return values

- **Frame** – length in octets
- **-EINVAL** – if arguments are invalid
- **-ENODATA** – if not found
- **-EBADMSG** – if found value has invalid size or value

int **bt_audio_codec_cfg_set_octets_per_frame**(struct *bt_audio_codec_cfg* *codec_cfg, uint16_t octets_per_frame)

Set the octets per codec frame of a codec configuration.

Parameters

- **codec_cfg** – The codec configuration to set data for.
- **octets_per_frame** – The octets per codec frame to set.

Return values

- **The** – data_len of `codec_cfg` on success
- **-EINVAL** – if arguments are invalid
- **-ENOMEM** – if the new value could not set or added due to memory

int **bt_audio_codec_cfg_get_frame_blocks_per_sdu**(const struct *bt_audio_codec_cfg* *codec_cfg, bool fallback_to_default)

Extract number of audio frame blocks in each SDU from BT codec config.

The overall SDU size will be octets_per_frame * frame_blocks_per_sdu * number-of-channels.

If this value is not present a default value of 1 shall be used.

A frame block is one or more frames that represents data for the same period of time but for different channels. If the stream have two audio channels and this value is two there will be four frames in the SDU.

Parameters

- **codec_cfg** – The codec configuration to extract data from.
- **fallback_to_default** – If true this function will return the default value of 1 if the type is not found when `codec_cfg.id` is BT_HCI_CODING_FORMAT_LC3.

Return values

- **The** – count of codec frame blocks in each SDU.
- **-EINVAL** – if arguments are invalid

- -ENODATA – if not found
- -EBADMSG – if found value has invalid size or value

int bt_audio_codec_cfg_set_frame_blocks_per_sdu(struct *bt_audio_codec_cfg* *codec_cfg, uint8_t frame_blocks)

Set the frame blocks per SDU of a codec configuration.

Parameters

- codec_cfg – The codec configuration to set data for.
- frame_blocks – The frame blocks per SDU to set.

Return values

- The – data_len of codec_cfg on success
- -EINVAL – if arguments are invalid
- -ENOMEM – if the new value could not set or added due to memory

int bt_audio_codec_cfg_get_val(const struct *bt_audio_codec_cfg* *codec_cfg, enum *bt_audio_codec_cfg_type* type, const uint8_t **data)

Lookup a specific codec configuration value.

Parameters

- codec_cfg – **[in]** The codec data to search in.
- type – **[in]** The type id to look for
- data – **[out]** Pointer to the data-pointer to update when item is found

Return values

- Length – of found data (may be 0)
- -EINVAL – if arguments are invalid
- -ENODATA – if not found

int bt_audio_codec_cfg_set_val(struct *bt_audio_codec_cfg* *codec_cfg, enum *bt_audio_codec_cfg_type* type, const uint8_t *data, size_t data_len)

Set or add a specific codec configuration value.

Parameters

- codec_cfg – The codec data to set the value in.
- type – The type id to set
- data – Pointer to the data-pointer to set
- data_len – Length of data

Return values

- The – data_len of codec_cfg on success
- -EINVAL – if arguments are invalid
- -ENOMEM – if the new value could not set or added due to memory

int bt_audio_codec_cfg_unset_val(struct *bt_audio_codec_cfg* *codec_cfg, enum *bt_audio_codec_cfg_type* type)

Unset a specific codec configuration value.

The type and the value will be removed from the codec configuration.

Parameters

- **codec_cfg** – The codec data to set the value in.
- **type** – The type id to unset.

Return values

- **The** – data_len of codec_cfg on success
- **-EINVAL** – if arguments are invalid

int **bt_audio_codec_cfg_meta_get_val**(const struct *bt_audio_codec_cfg* *codec_cfg, uint8_t type, const uint8_t **data)

Lookup a specific metadata value based on type.

Parameters

- **codec_cfg** – **[in]** The codec data to search in.
- **type** – **[in]** The type id to look for
- **data** – **[out]** Pointer to the data-pointer to update when item is found

Return values

- **Length** – of found data (may be 0)
- **-EINVAL** – if arguments are invalid
- **-ENODATA** – if not found

int **bt_audio_codec_cfg_meta_set_val**(struct *bt_audio_codec_cfg* *codec_cfg, enum *bt_audio_metadata_type* type, const uint8_t *data, size_t data_len)

Set or add a specific codec configuration metadata value.

Parameters

- **codec_cfg** – The codec configuration to set the value in.
- **type** – The type id to set.
- **data** – Pointer to the data-pointer to set.
- **data_len** – Length of data.

Return values

- **The** – meta_len of codec_cfg on success
- **-EINVAL** – if arguments are invalid
- **-ENOMEM** – if the new value could not set or added due to memory

int **bt_audio_codec_cfg_meta_unset_val**(struct *bt_audio_codec_cfg* *codec_cfg, enum *bt_audio_metadata_type* type)

Unset a specific codec configuration metadata value.

The type and the value will be removed from the codec configuration metadata.

Parameters

- **codec_cfg** – The codec data to set the value in.
- **type** – The type id to unset.

Return values

- **The** – meta_len of codec_cfg on success
- **-EINVAL** – if arguments are invalid

int **bt_audio_codec_cfg_meta_get_pref_context**(const struct *bt_audio_codec_cfg* *codec_cfg, bool fallback_to_default)

Extract preferred contexts.

See *BT_AUDIO_METADATA_TYPE_PREF_CONTEXT* for more information about this value.

Parameters

- **codec_cfg** – The codec data to search in.
- **fallback_to_default** – If true this function will provide the default value of *BT_AUDIO_CONTEXT_TYPE_UNSPECIFIED* if the type is not found when `codec_cfg.id` is BT_HCI_CODING_FORMAT_LC3.

Return values

- **The** – preferred context type if positive or 0
- **-EINVAL** – if arguments are invalid
- **-ENODATA** – if not found
- **-EBADMSG** – if found value has invalid size

int **bt_audio_codec_cfg_meta_set_pref_context**(struct *bt_audio_codec_cfg* *codec_cfg, enum *bt_audio_context* ctx)

Set the preferred context of a codec configuration metadata.

Parameters

- **codec_cfg** – The codec configuration to set data for.
- **ctx** – The preferred context to set.

Return values

- **The** – data_len of `codec_cfg` on success
- **-EINVAL** – if arguments are invalid
- **-ENOMEM** – if the new value could not set or added due to memory

int **bt_audio_codec_cfg_meta_get_stream_context**(const struct *bt_audio_codec_cfg* *codec_cfg)

Extract stream contexts.

See *BT_AUDIO_METADATA_TYPE_STREAM_CONTEXT* for more information about this value.

Parameters

- **codec_cfg** – The codec data to search in.

Return values

- **The** – stream context type if positive or 0
- **-EINVAL** – if arguments are invalid
- **-ENODATA** – if not found
- **-EBADMSG** – if found value has invalid size

int **bt_audio_codec_cfg_meta_set_stream_context**(struct *bt_audio_codec_cfg* *codec_cfg, enum *bt_audio_context* ctx)

Set the stream context of a codec configuration metadata.

Parameters

- **codec_cfg** – The codec configuration to set data for.

- **ctx** – The stream context to set.

Return values

- **The** – data_len of codec_cfg on success
- **-EINVAL** – if arguments are invalid
- **-ENOMEM** – if the new value could not set or added due to memory

int **bt_audio_codec_cfg_meta_get_program_info**(const struct *bt_audio_codec_cfg* *codec_cfg, const uint8_t **program_info)

Extract program info.

See *BT_AUDIO_METADATA_TYPE_PROGRAM_INFO* for more information about this value.

Parameters

- **codec_cfg** – **[in]** The codec data to search in.
- **program_info** – **[out]** Pointer to the UTF-8 formatted program info.

Return values

- **The** – length of the program_info (may be 0)
- **-EINVAL** – if arguments are invalid
- **-ENODATA** – if not found

int **bt_audio_codec_cfg_meta_set_program_info**(struct *bt_audio_codec_cfg* *codec_cfg, const uint8_t *program_info, size_t program_info_len)

Set the program info of a codec configuration metadata.

Parameters

- **codec_cfg** – The codec configuration to set data for.
- **program_info** – The program info to set.
- **program_info_len** – The length of program_info.

Return values

- **The** – data_len of codec_cfg on success
- **-EINVAL** – if arguments are invalid
- **-ENOMEM** – if the new value could not set or added due to memory

int **bt_audio_codec_cfg_meta_get_lang**(const struct *bt_audio_codec_cfg* *codec_cfg, const uint8_t **lang)

Extract language.

See *BT_AUDIO_METADATA_TYPE_LANG* for more information about this value.

Parameters

- **codec_cfg** – **[in]** The codec data to search in.
- **lang** – **[out]** Pointer to the language bytes (of length BT_AUDIO_LANG_SIZE)

Return values

- **The** – language if positive or 0
- **-EINVAL** – if arguments are invalid
- **-ENODATA** – if not found

- **-EBADMSG** – if found value has invalid size

int **bt_audio_codec_cfg_meta_set_lang**(struct *bt_audio_codec_cfg* *codec_cfg, const uint8_t lang[3])

Set the language of a codec configuration metadata.

Parameters

- **codec_cfg** – The codec configuration to set data for.
- **lang** – The 24-bit language to set.

Return values

- **The** – data_len of `codec_cfg` on success
- **-EINVAL** – if arguments are invalid
- **-ENOMEM** – if the new value could not set or added due to memory

int **bt_audio_codec_cfg_meta_get_ccid_list**(const struct *bt_audio_codec_cfg* *codec_cfg, const uint8_t **ccid_list)

Extract CCID list.

See *BT_AUDIO_METADATA_TYPE_CCID_LIST* for more information about this value.

Parameters

- **codec_cfg** – **[in]** The codec data to search in.
- **ccid_list** – **[out]** Pointer to the array containing 8-bit CCIDs.

Return values

- **The** – length of the `ccid_list` (may be 0)
- **-EINVAL** – if arguments are invalid
- **-ENODATA** – if not found

int **bt_audio_codec_cfg_meta_set_ccid_list**(struct *bt_audio_codec_cfg* *codec_cfg, const uint8_t *ccid_list, size_t ccid_list_len)

Set the CCID list of a codec configuration metadata.

Parameters

- **codec_cfg** – The codec configuration to set data for.
- **ccid_list** – The program info to set.
- **ccid_list_len** – The length of `ccid_list`.

Return values

- **The** – data_len of `codec_cfg` on success
- **-EINVAL** – if arguments are invalid
- **-ENOMEM** – if the new value could not set or added due to memory

int **bt_audio_codec_cfg_meta_get_parental_rating**(const struct *bt_audio_codec_cfg* *codec_cfg)

Extract parental rating.

See *BT_AUDIO_METADATA_TYPE_PARENTAL_RATING* for more information about this value.

Parameters

- **codec_cfg** – The codec data to search in.

Return values

- **The** – parental rating if positive or 0
- **-EINVAL** – if arguments are invalid
- **-ENODATA** – if not found
- **-EBADMSG** – if found value has invalid size

int **bt_audio_codec_cfg_meta_set_parental_rating**(struct *bt_audio_codec_cfg* *codec_cfg, enum *bt_audio_parental_rating* parental_rating)

Set the parental rating of a codec configuration metadata.

Parameters

- **codec_cfg** – The codec configuration to set data for.
- **parental_rating** – The parental rating to set.

Return values

- **The** – data_len of `codec_cfg` on success
- **-EINVAL** – if arguments are invalid
- **-ENOMEM** – if the new value could not set or added due to memory

int **bt_audio_codec_cfg_meta_get_program_info_uri**(const struct *bt_audio_codec_cfg* *codec_cfg, const uint8_t **program_info_uri)

Extract program info URI.

See *BT_AUDIO_METADATA_TYPE_PROGRAM_INFO_URI* for more information about this value.

Parameters

- **codec_cfg** – **[in]** The codec data to search in.
- **program_info_uri** – **[out]** Pointer to the UTF-8 formatted program info URI.

Return values

- **The** – length of the `ccid_list` (may be 0)
- **-EINVAL** – if arguments are invalid
- **-ENODATA** – if not found

int **bt_audio_codec_cfg_meta_set_program_info_uri**(struct *bt_audio_codec_cfg* *codec_cfg, const uint8_t *program_info_uri, size_t program_info_uri_len)

Set the program info URI of a codec configuration metadata.

Parameters

- **codec_cfg** – The codec configuration to set data for.
- **program_info_uri** – The program info URI to set.
- **program_info_uri_len** – The length of `program_info_uri`.

Return values

- **The** – data_len of `codec_cfg` on success
- **-EINVAL** – if arguments are invalid
- **-ENOMEM** – if the new value could not set or added due to memory

int **bt_audio_codec_cfg_meta_get_audio_active_state**(const struct *bt_audio_codec_cfg* *codec_cfg)

Extract audio active state.

See *BT_AUDIO_METADATA_TYPE_AUDIO_STATE* for more information about this value.

Parameters

- **codec_cfg** – The codec data to search in.

Return values

- **The** – preferred context type if positive or 0
- **-EINVAL** – if arguments are invalid
- **-ENODATA** – if not found
- **-EBADMSG** – if found value has invalid size

int **bt_audio_codec_cfg_meta_set_audio_active_state**(struct *bt_audio_codec_cfg* *codec_cfg, enum *bt_audio_active_state* state)

Set the audio active state of a codec configuration metadata.

Parameters

- **codec_cfg** – The codec configuration to set data for.
- **state** – The audio active state to set.

Return values

- **The** – data_len of `codec_cfg` on success
- **-EINVAL** – if arguments are invalid
- **-ENOMEM** – if the new value could not set or added due to memory

int **bt_audio_codec_cfg_meta_get_bcast_audio_immediate_rend_flag**(const struct *bt_audio_codec_cfg* *codec_cfg)

Extract broadcast audio immediate rendering flag.

See *BT_AUDIO_METADATA_TYPE_BROADCAST_IMMEDIATE* for more information about this value.

Parameters

- **codec_cfg** – The codec data to search in.

Return values

- **0** – if the flag was found
- **-EINVAL** – if arguments are invalid
- **-ENODATA** – if not the flag was not found

int **bt_audio_codec_cfg_meta_set_bcast_audio_immediate_rend_flag**(struct *bt_audio_codec_cfg* *codec_cfg)

Set the broadcast audio immediate rendering flag of a codec configuration metadata.

Parameters

- **codec_cfg** – The codec configuration to set data for.

Return values

- **The** – data_len of codec_cfg on success
- **-EINVAL** – if arguments are invalid
- **-ENOMEM** – if the new value could not set or added due to memory

int **bt_audio_codec_cfg_meta_get_extended**(const struct *bt_audio_codec_cfg* *codec_cfg, const uint8_t **extended_meta)

Extract extended metadata.

See *BT_AUDIO_METADATA_TYPE_EXTENDED* for more information about this value.

Parameters

- **codec_cfg** – **[in]** The codec data to search in.
- **extended_meta** – **[out]** Pointer to the extended metadata.

Return values

- **The** – length of the ccid_list (may be 0)
- **-EINVAL** – if arguments are invalid
- **-ENODATA** – if not found

int **bt_audio_codec_cfg_meta_set_extended**(struct *bt_audio_codec_cfg* *codec_cfg, const uint8_t *extended_meta, size_t extended_meta_len)

Set the extended metadata of a codec configuration metadata.

Parameters

- **codec_cfg** – The codec configuration to set data for.
- **extended_meta** – The extended metadata to set.
- **extended_meta_len** – The length of extended_meta.

Return values

- **The** – data_len of codec_cfg on success
- **-EINVAL** – if arguments are invalid
- **-ENOMEM** – if the new value could not set or added due to memory

int **bt_audio_codec_cfg_meta_get_vendor**(const struct *bt_audio_codec_cfg* *codec_cfg, const uint8_t **vendor_meta)

Extract vendor specific metadata.

See *BT_AUDIO_METADATA_TYPE_VENDOR* for more information about this value.

Parameters

- **codec_cfg** – **[in]** The codec data to search in.
- **vendor_meta** – **[out]** Pointer to the vendor specific metadata.

Return values

- **The** – length of the ccid_list (may be 0)
- **-EINVAL** – if arguments are invalid
- **-ENODATA** – if not found

int **bt_audio_codec_cfg_meta_set_vendor**(struct *bt_audio_codec_cfg* *codec_cfg, const uint8_t *vendor_meta, size_t vendor_meta_len)

Set the vendor specific metadata of a codec configuration metadata.

Parameters

- `codec_cfg` – The codec configuration to set data for.
- `vendor_meta` – The vendor specific metadata to set.
- `vendor_meta_len` – The length of `vendor_meta`.

Return values

- `The` – data_len of `codec_cfg` on success
- `-EINVAL` – if arguments are invalid
- `-ENOMEM` – if the new value could not set or added due to memory

Basic Audio Profile

Related code samples

Bluetooth: Broadcast Audio Assistant
Use LE Audio Broadcast Assistant functionality

Bluetooth: Common Audio Profile Acceptor
CAP Acceptor sample that advertises audio availability to CAP Initiators.

Bluetooth: Common Audio Profile Initiator
CAP Initiator sample that connects to CAP Acceptors and setup audio streaming.

API Reference

group `bt_bap`

Bluetooth Basic Audio Profile (BAP)

The Basic Audio Profile (BAP) allows for both unicast and broadcast Audio Stream control.

Since
3.0

Version
0.8.0

Defines

`BT_BAP_PA_INTERVAL_UNKNOWN`

Value indicating that the periodic advertising interval is unknown.

`BT_BAP_BIS_SYNC_NO_PREF`

Broadcast Assistant no BIS sync preference.

Value indicating that the Broadcast Assistant has no preference to which BIS the Scan Delegator syncs to

BT_BAP_ASCS_RSP(c, r)

Macro used to initialise the object storing values of ASE Control Point notification.

Parameters

- c – Response Code field
- r – Reason field - *bt_bap_ascs_reason* or *bt_audio_metadata_type* (see notes in *bt_bap_ascs_rsp*).

Typedefs

typedef bool (*bt_bap_scan_delegator_state_func_t)(const struct *bt_bap_scan_delegator_recv_state* *recv_state, void *user_data)

Callback function for Scan Delegator receive state search functions.

Param recv_state
The receive state.

Param user_data
User data.

Retval true
to stop iterating. If this is used in the context of *bt_bap_scan_delegator_find_state()*, the recv_state will be returned by *bt_bap_scan_delegator_find_state()*

Retval false
to continue iterating

typedef void (*bt_bap_broadcast_assistant_write_cb)(struct bt_conn *conn, int err)

Callback function for writes.

Param conn
The connection to the peer device.

Param err
Error value. 0 on success, GATT error on fail.

Enums

enum bt_bap_pa_state

Periodic advertising state reported by the Scan Delegator.

Values:

enumerator BT_BAP_PA_STATE_NOT_SYNCED = 0x00

The periodic advertising has not been synchronized.

enumerator BT_BAP_PA_STATE_INFO_REQ = 0x01

Waiting for SyncInfo from Broadcast Assistant.

enumerator BT_BAP_PA_STATE_SYNCED = 0x02

Synchronized to periodic advertising.

enumerator BT_BAP_PA_STATE_FAILED = 0x03
Failed to synchronized to periodic advertising.

enumerator BT_BAP_PA_STATE_NO_PAST = 0x04
No periodic advertising sync transfer receiver from Broadcast Assistant.

enum bt_bap_big_enc_state
Broadcast Isochronous Group encryption state reported by the Scan Delegator.
Values:

enumerator BT_BAP_BIG_ENC_STATE_NO_ENC = 0x00
The Broadcast Isochronous Group not encrypted.

enumerator BT_BAP_BIG_ENC_STATE_BCODE_REQ = 0x01
The Broadcast Isochronous Group broadcast code requested.

enumerator BT_BAP_BIG_ENC_STATE_DEC = 0x02
The Broadcast Isochronous Group decrypted.

enumerator BT_BAP_BIG_ENC_STATE_BAD_CODE = 0x03
The Broadcast Isochronous Group bad broadcast code.

enum bt_bap_bass_att_err
Broadcast Audio Scan Service (BASS) specific ATT error codes.
Values:

enumerator BT_BAP_BASS_ERR_OPCODE_NOT_SUPPORTED = 0x80
Opcode not supported.

enumerator BT_BAP_BASS_ERR_INVALID_SRC_ID = 0x81
Invalid source ID supplied.

enum bt_bap_ep_state
Endpoint states.
Values:

enumerator BT_BAP_EP_STATE_IDLE = 0x00
Audio Stream Endpoint Idle state.

enumerator BT_BAP_EP_STATE_CODEC_CONFIGURED = 0x01
Audio Stream Endpoint Codec Configured state.

enumerator BT_BAP_EP_STATE_QOS_CONFIGURED = 0x02
Audio Stream Endpoint QoS Configured state.

enumerator BT_BAP_EP_STATE_ENABLING = 0x03
Audio Stream Endpoint Enabling state.

enumerator BT_BAP_EP_STATE_STREAMING = 0x04
Audio Stream Endpoint Streaming state.

enumerator BT_BAP_EP_STATE_DISABLING = 0x05
Audio Stream Endpoint Disabling state.

enumerator BT_BAP_EP_STATE_RELEASING = 0x06
Audio Stream Endpoint Streaming state.

enum bt_bap_ascs_rsp_code
Response Status Code.
These are sent by the server to the client when a stream operation is requested.
Values:

enumerator BT_BAP_ASCS_RSP_CODE_SUCCESS = 0x00
Server completed operation successfully.

enumerator BT_BAP_ASCS_RSP_CODE_NOT_SUPPORTED = 0x01
Server did not support operation by client.

enumerator BT_BAP_ASCS_RSP_CODE_INVALID_LENGTH = 0x02
Server rejected due to invalid operation length.

enumerator BT_BAP_ASCS_RSP_CODE_INVALID_ASE = 0x03
Invalid ASE ID.

enumerator BT_BAP_ASCS_RSP_CODE_INVALID_ASE_STATE = 0x04
Invalid ASE state.

enumerator BT_BAP_ASCS_RSP_CODE_INVALID_DIR = 0x05
Invalid operation for direction.

enumerator BT_BAP_ASCS_RSP_CODE_CAP_UNSUPPORTED = 0x06
Capabilities not supported by server.

enumerator BT_BAP_ASCS_RSP_CODE_CONF_UNSUPPORTED = 0x07
Configuration parameters not supported by server.

enumerator BT_BAP_ASCS_RSP_CODE_CONF_REJECTED = 0x08
Configuration parameters rejected by server.

enumerator BT_BAP_ASCS_RSP_CODE_CONF_INVALID = 0x09
Invalid Configuration parameters.

enumerator BT_BAP_ASCS_RSP_CODE_METADATA_UNSUPPORTED = 0x0a
Unsupported metadata.

enumerator BT_BAP_ASCS_RSP_CODE_METADATA_REJECTED = 0x0b
Metadata rejected by server.

enumerator BT_BAP_ASCS_RSP_CODE_METADATA_INVALID = 0x0c
Invalid metadata.

enumerator BT_BAP_ASCS_RSP_CODE_NO_MEM = 0x0d
Server has insufficient resources.

enumerator BT_BAP_ASCS_RSP_CODE_UNSPECIFIED = 0x0e
Unspecified error.

enum bt_bap_ascs_reason
Response Reasons.

These are used if the *bt_bap_ascs_rsp_code* value is *BT_BAP_ASCS_RSP_CODE_CONF_UNSUPPORTED*, *BT_BAP_ASCS_RSP_CODE_CONF_REJECTED* or *BT_BAP_ASCS_RSP_CODE_CONF_INVALID*.

Values:

enumerator BT_BAP_ASCS_REASON_NONE = 0x00
No reason.

enumerator BT_BAP_ASCS_REASON_CODEC = 0x01
Codec ID.

enumerator BT_BAP_ASCS_REASON_CODEC_DATA = 0x02
Codec configuration.

enumerator BT_BAP_ASCS_REASON_INTERVAL = 0x03
SDU interval.

enumerator BT_BAP_ASCS_REASON_FRAMING = 0x04
Framing.

enumerator BT_BAP_ASCS_REASON_PHY = 0x05
PHY.

enumerator BT_BAP_ASCS_REASON_SDU = 0x06
Maximum SDU size.

enumerator BT_BAP_ASCS_REASON_RTN = 0x07
RTN.

enumerator BT_BAP_ASCS_REASON_LATENCY = 0x08
Max transport latency.

enumerator BT_BAP_ASCS_REASON_PD = 0x09
Presendation delay.

enumerator **BT_BAP_ASCS_REASON_CIS** = 0x0a

Invalid CIS mapping.

Functions

int **bt_bap_ep_get_info**(const struct bt_bap_ep *ep, struct *bt_bap_ep_info* *info)

Return structure holding information of audio stream endpoint.

Parameters

- **ep** – The audio stream endpoint object.
- **info** – The structure object to be filled with the info.

Return values

- **0** – in case of success
- **-EINVAL** – if ep or info are NULL

void **bt_bap_stream_cb_register**(struct *bt_bap_stream* *stream, struct *bt_bap_stream_ops* *ops)

Register Audio callbacks for a stream.

Register Audio callbacks for a stream.

Parameters

- **stream** – Stream object.
- **ops** – Stream operations structure.

int **bt_bap_stream_config**(struct bt_conn *conn, struct *bt_bap_stream* *stream, struct bt_bap_ep *ep, struct *bt_audio_codec_cfg* *codec_cfg)

Configure Audio Stream.

This procedure is used by a client to configure a new stream using the remote endpoint, local capability and codec configuration.

Parameters

- **conn** – Connection object
- **stream** – Stream object being configured
- **ep** – Remote Audio Endpoint being configured
- **codec_cfg** – Codec configuration

Returns

Allocated Audio Stream object or NULL in case of error.

int **bt_bap_stream_reconfig**(struct *bt_bap_stream* *stream, struct *bt_audio_codec_cfg* *codec_cfg)

Reconfigure Audio Stream.

This procedure is used by a unicast client or unicast server to reconfigure a stream to use a different local codec configuration.

This can only be done for unicast streams.

Parameters

- **stream** – Stream object being reconfigured
- **codec_cfg** – Codec configuration

Returns
0 in case of success or negative value in case of error.

int **bt_bap_stream_qos**(struct bt_conn *conn, struct bt_bap_unicast_group *group)

Configure Audio Stream QoS.

This procedure is used by a client to configure the Quality of Service of streams in a unicast group. All streams in the group for the specified conn will have the Quality of Service configured. This shall only be used to configure unicast streams.

Parameters

- **conn** – Connection object
- **group** – Unicast group object

Returns
0 in case of success or negative value in case of error.

int **bt_bap_stream_enable**(struct *bt_bap_stream* *stream, const uint8_t meta[], size_t meta_len)

Enable Audio Stream.

This procedure is used by a client to enable a stream.

This shall only be called for unicast streams, as broadcast streams will always be enabled once created.

Parameters

- **stream** – Stream object
- **meta** – Metadata
- **meta_len** – Metadata length

Returns
0 in case of success or negative value in case of error.

int **bt_bap_stream_metadata**(struct *bt_bap_stream* *stream, const uint8_t meta[], size_t meta_len)

Change Audio Stream Metadata.

This procedure is used by a unicast client or unicast server to change the metadata of a stream.

Parameters

- **stream** – Stream object
- **meta** – Metadata
- **meta_len** – Metadata length

Returns
0 in case of success or negative value in case of error.

int **bt_bap_stream_disable**(struct *bt_bap_stream* *stream)

Disable Audio Stream.

This procedure is used by a unicast client or unicast server to disable a stream.

This shall only be called for unicast streams, as broadcast streams will always be enabled once created.

Parameters

- **stream** – Stream object

Returns
0 in case of success or negative value in case of error.

int **bt_bap_stream_connect**(struct *bt_bap_stream* *stream)

Connect unicast audio stream.

This procedure is used by a unicast client to connect the connected isochronous stream (CIS) associated with the audio stream. If two audio streams share a CIS, then this only needs to be done once for those streams. This can only be done for streams in the QoS configured or enabled states.

The *bt_bap_stream_ops.connected()* callback will be called on the streams once this has finished.

This shall only be called for unicast streams, and only as the unicast client (`CONFIG_BT_BAP_UNICAST_CLIENT`).

Parameters

- **stream** – Stream object

Return values

- **0** – in case of success
- **-EINVAL** – if the stream, endpoint, ISO channel or connection is NULL
- **-EBADMSG** – if the stream or ISO channel is in an invalid state for connection
- **-EOPNOTSUPP** – if the role of the stream is not BT_HCI_ROLE_CENTRAL
- **-EALREADY** – if the ISO channel is already connecting or connected
- **-EBUSY** – if another ISO channel is connecting
- **-ENOEXEC** – if otherwise rejected by the ISO layer

int **bt_bap_stream_start**(struct *bt_bap_stream* *stream)

Start Audio Stream.

This procedure is used by a unicast client or unicast server to make a stream start streaming.

For the unicast client, this will send the receiver start ready command to the unicast server for *BT_AUDIO_DIR_SOURCE* ASEs. The CIS is required to be connected first by *bt_bap_stream_connect()* before the command can be sent.

For the unicast server, this will execute the receiver start ready command on the unicast server for *BT_AUDIO_DIR_SINK* ASEs. If the CIS is not connected yet, the stream will go into the streaming state as soon as the CIS is connected.

This shall only be called for unicast streams.

Broadcast sinks will always be started once synchronized, and broadcast source streams shall be started with *bt_bap_broadcast_source_start()*.

Parameters

- **stream** – Stream object

Returns

0 in case of success or negative value in case of error.

int **bt_bap_stream_stop**(struct *bt_bap_stream* *stream)

Stop Audio Stream.

This procedure is used by a client to make a stream stop streaming.

This shall only be called for unicast streams. Broadcast sinks cannot be stopped. Broadcast sources shall be stopped with *bt_bap_broadcast_source_stop()*.

Parameters

- **stream** – Stream object

Returns
0 in case of success or negative value in case of error.

int **bt_bap_stream_release**(struct *bt_bap_stream* *stream)

Release Audio Stream.

This procedure is used by a unicast client or unicast server to release a unicast stream.

Broadcast sink streams cannot be released, but can be deleted by *bt_bap_broadcast_sink_delete()*. Broadcast source streams cannot be released, but can be deleted by *bt_bap_broadcast_source_delete()*.

Parameters

- **stream** – Stream object

Returns
0 in case of success or negative value in case of error.

int **bt_bap_stream_send**(struct *bt_bap_stream* *stream, struct *net_buf* *buf, uint16_t seq_num)

Send data to Audio stream without timestamp.

Send data from buffer to the stream.

Note

Support for sending must be supported, determined by CONFIG_BT_AUDIO_TX .

Parameters

- **stream** – Stream object.
- **buf** – Buffer containing data to be sent.
- **seq_num** – Packet Sequence number. This value shall be incremented for each call to this function and at least once per SDU interval for a specific channel.

Returns
Bytes sent in case of success or negative value in case of error.

int **bt_bap_stream_send_ts**(struct *bt_bap_stream* *stream, struct *net_buf* *buf, uint16_t seq_num, uint32_t ts)

Send data to Audio stream with timestamp.

Send data from buffer to the stream.

Note

Support for sending must be supported, determined by CONFIG_BT_AUDIO_TX .

Parameters

- **stream** – Stream object.
- **buf** – Buffer containing data to be sent.
- **seq_num** – Packet Sequence number. This value shall be incremented for each call to this function and at least once per SDU interval for a specific channel.

- **ts** – Timestamp of the SDU in microseconds (us). This value can be used to transmit multiple SDUs in the same SDU interval in a CIG or BIG.

Returns
Bytes sent in case of success or negative value in case of error.

int **bt_bap_stream_get_tx_sync**(struct *bt_bap_stream* *stream, struct bt_iso_tx_info *info)

Get ISO transmission timing info for a Basic Audio Profile stream.

Reads timing information for transmitted ISO packet on an ISO channel. The HCI_LE_Read_ISO_TX_Sync HCI command is used to retrieve this information from the controller.

Note

An SDU must have already been successfully transmitted on the ISO channel for this function to return successfully. Support for sending must be supported, determined by `CONFIG_BT_AUDIO_TX`.

Parameters

- **stream** – **[in]** Stream object.
- **info** – **[out]** Transmit info object.

Return values

- **0** – on success
- **-EINVAL** – if the stream is invalid, if the stream is not configured for sending or if it is not connected with a isochronous stream
- **Any** – return value from bt_iso_chan_get_tx_sync()

void **bt_bap_scan_delegator_register_cb**(struct *bt_bap_scan_delegator_cb* *cb)

Register the callbacks for the Basic Audio Profile Scan Delegator.

Only one set of callbacks can be registered at any one time, and calling this function multiple times will override any previously registered callbacks.

Parameters

- **cb** – Pointer to the callback struct

int **bt_bap_scan_delegator_set_pa_state**(uint8_t src_id, enum *bt_bap_pa_state* pa_state)

Set the periodic advertising sync state to syncing.

Set the periodic advertising sync state for a receive state to syncing, notifying Broadcast Assistants.

Parameters

- **src_id** – The source id used to identify the receive state.
- **pa_state** – The Periodic Advertising sync state to set. BT_BAP_PA_STATE_NOT_SYNCED and BT_BAP_PA_STATE_SYNCED is not necessary to provide, as they are handled internally.

Returns
int Error value. 0 on success, errno on fail.

int **bt_bap_scan_delegator_set_bis_sync_state**(uint8_t src_id, uint32_t bis_synced[CONFIG_BT_BAP_BASS_MAX_SUBGROUPS])

Set the sync state of a receive state in the server.

Parameters

- `src_id` – The source id used to identify the receive state.
- `bis_synced` – Array of bitfields to set the BIS sync state for each subgroup.

Returns
int Error value. 0 on success, ERRNO on fail.

int `bt_bap_scan_delegator_add_src`(const struct *bt_bap_scan_delegator_add_src_param* *param)

Add a receive state source locally.

This will notify any connected clients about the new source. This allows them to modify and even remove it.

If `CONFIG_BT_BAP_BROADCAST_SINK` is enabled, any Broadcast Sink sources are autonomously added.

Parameters

- `param` – The parameters for adding the new source

Returns
int errno on failure, or source ID on success.

int `bt_bap_scan_delegator_mod_src`(const struct *bt_bap_scan_delegator_mod_src_param* *param)

Add a receive state source locally.

This will notify any connected clients about the new source. This allows them to modify and even remove it.

If `CONFIG_BT_BAP_BROADCAST_SINK` is enabled, any Broadcast Sink sources are autonomously modified.

Parameters

- `param` – The parameters for adding the new source

Returns
int errno on failure, or source ID on success.

int `bt_bap_scan_delegator_rem_src`(uint8_t src_id)

Remove a receive state source.

This will remove the receive state. If the receive state periodic advertising is synced, *bt_bap_scan_delegator_cb.pa_sync_term_req()* will be called.

If `CONFIG_BT_BAP_BROADCAST_SINK` is enabled, any Broadcast Sink sources are autonomously removed.

Parameters

- `src_id` – The source ID to remove

Returns
int Error value. 0 on success, errno on fail.

void `bt_bap_scan_delegator_foreach_state`(*bt_bap_scan_delegator_state_func_t* func, void *user_data)

Iterate through all existing receive states.

Parameters

- `func` – The callback function
- `user_data` – User specified data that sent to the callback function

const struct *bt_bap_scan_delegator_recv_state* ***bt_bap_scan_delegator_find_state**(*bt_bap_scan_delegato*
func,
void
*user_data)

Find and return a receive state based on a compare function.

Parameters

- **func** – The compare callback function
- **user_data** – User specified data that sent to the callback function

Returns
The first receive state where the func returned true, or NULL

int **bt_bap_broadcast_assistant_discover**(struct bt_conn *conn)

Discover Broadcast Audio Scan Service on the server.

Warning: Only one connection can be active at any time; discovering for a new connection, will delete all previous data.

Parameters

- **conn** – The connection

Returns
int Error value. 0 on success, GATT error or ERRNO on fail.

int **bt_bap_broadcast_assistant_scan_start**(struct bt_conn *conn, bool start_scan)

Scan start for BISes for a remote server.

This will let the Broadcast Audio Scan Service server know that this device is actively scanning for broadcast sources. The function can optionally also start scanning, if the caller does not want to start scanning itself.

Scan results, if start_scan is true, is sent to the bt_bap_broadcast_assistant_scan_cb callback.

Parameters

- **conn** – Connection to the Broadcast Audio Scan Service server. Used to let the server know that we are scanning.
- **start_scan** – Start scanning if true. If false, the application should enable scan itself.

Returns
int Error value. 0 on success, GATT error or ERRNO on fail.

int **bt_bap_broadcast_assistant_scan_stop**(struct bt_conn *conn)

Stop remote scanning for BISes for a server.

Parameters

- **conn** – Connection to the server.

Returns
int Error value. 0 on success, GATT error or ERRNO on fail.

int **bt_bap_broadcast_assistant_register_cb**(struct *bt_bap_broadcast_assistant_cb* *cb)

Registers the callbacks used by Broadcast Audio Scan Service client.

Parameters

- **cb** – The callback structure.

Return values

- **0** – on success

- **-EINVAL** – if cb is NULL
- **-EALREADY** – if cb was already registered

int **bt_bap_broadcast_assistant_unregister_cb**(struct *bt_bap_broadcast_assistant_cb* *cb)

Unregisters the callbacks used by the Broadcast Audio Scan Service client.

Parameters

- **cb** – The callback structure.

Return values

- **0** – on success
- **-EINVAL** – if cb is NULL
- **-EALREADY** – if cb was not registered

int **bt_bap_broadcast_assistant_add_src**(struct bt_conn *conn, const struct *bt_bap_broadcast_assistant_add_src_param* *param)

Add a source on the server.

Parameters

- **conn** – Connection to the server.
- **param** – Parameter struct.

Returns
Error value. 0 on success, GATT error or ERRNO on fail.

int **bt_bap_broadcast_assistant_mod_src**(struct bt_conn *conn, const struct *bt_bap_broadcast_assistant_mod_src_param* *param)

Modify a source on the server.

Parameters

- **conn** – Connection to the server.
- **param** – Parameter struct.

Returns
Error value. 0 on success, GATT error or ERRNO on fail.

int **bt_bap_broadcast_assistant_set_broadcast_code**(struct bt_conn *conn, uint8_t src_id, const uint8_t broadcast_code[*BT_AUDIO_BROADCAST_CODE_SIZE*])

Set a broadcast code to the specified receive state.

Parameters

- **conn** – Connection to the server.
- **src_id** – Source ID of the receive state.
- **broadcast_code** – The broadcast code.

Returns
Error value. 0 on success, GATT error or ERRNO on fail.

int **bt_bap_broadcast_assistant_rem_src**(struct bt_conn *conn, uint8_t src_id)

Remove a source from the server.

Parameters

- **conn** – Connection to the server.

- **src_id** – Source ID of the receive state.

Returns
Error value. 0 on success, GATT error or ERRNO on fail.

int **bt_bap_broadcast_assistant_read_recv_state**(struct bt_conn *conn, uint8_t idx)

Read the specified receive state from the server.

Parameters

- **conn** – Connection to the server.
- **idx** – The index of the receive start (0 up to the value from bt_bap_broadcast_assistant_discover_cb)

Returns
Error value. 0 on success, GATT error or ERRNO on fail.

struct **bt_bap_ascs_rsp**

#include <bap.h> Structure storing values of fields of ASE Control Point notification.

Public Members

enum *bt_bap_ascs_rsp_code* **code**

Value of the Response Code field.

The following response codes are accepted:
- *BT_BAP_ASCS_RSP_CODE_SUCCESS*
- *BT_BAP_ASCS_RSP_CODE_CAP_UNSUPPORTED*
- *BT_BAP_ASCS_RSP_CODE_CONF_UNSUPPORTED*
- *BT_BAP_ASCS_RSP_CODE_CONF_REJECTED*
- *BT_BAP_ASCS_RSP_CODE_METADATA_UNSUPPORTED*
- *BT_BAP_ASCS_RSP_CODE_METADATA_REJECTED*
- *BT_BAP_ASCS_RSP_CODE_NO_MEM*
- *BT_BAP_ASCS_RSP_CODE_UNSPECIFIED*

enum *bt_bap_ascs_reason* **reason**

Response reason.

If the Response Code is one of the following:
- *BT_BAP_ASCS_RSP_CODE_CONF_UNSUPPORTED*
- *BT_BAP_ASCS_RSP_CODE_CONF_REJECTED* all values from *bt_bap_ascs_reason* can be used.

If the Response Code is one of the following:
- *BT_BAP_ASCS_RSP_CODE_SUCCESS*
- *BT_BAP_ASCS_RSP_CODE_CAP_UNSUPPORTED*
- *BT_BAP_ASCS_RSP_CODE_NO_MEM*
- *BT_BAP_ASCS_RSP_CODE_UNSPECIFIED* only value *BT_BAP_ASCS_REASON_NONE* shall be used.

enum *bt_audio_metadata_type* **metadata_type**

Response metadata type.

If the Response Code is one of the following:
- *BT_BAP_ASCS_RSP_CODE_METADATA_UNSUPPORTED*
- *BT_BAP_ASCS_RSP_CODE_METADATA_REJECTED* the value of the Metadata Type shall be used.

union **bt_bap_ascs_rsp**

Value of the Reason field.

The meaning of this value depend on the Response Code field.

struct **bt_bap_bass_subgroup**

#include <*bap.h*> Struct to hold subgroup specific information for the receive state.

Public Members

uint32_t **bis_sync**

BIS synced bitfield.

uint8_t **metadata_len**

Length of the metadata.

uint8_t **metadata**[CONFIG_BT_AUDIO_CODEC_CFG_MAX_METADATA_SIZE]

The metadata.

struct **bt_bap_scan_delegator_recv_state**

#include <*bap.h*> Represents the Broadcast Audio Scan Service receive state.

Public Members

uint8_t **src_id**

The source ID

bt_addr_le_t **addr**

The Bluetooth address.

uint8_t **adv_sid**

The advertising set ID.

enum *bt_bap_pa_state* **pa_sync_state**

The periodic adverting sync state.

enum *bt_bap_big_enc_state* **encrypt_state**

The broadcast isochronous group encryption state.

uint32_t **broadcast_id**

The 24-bit broadcast ID.

uint8_t **bad_code**[*BT_AUDIO_BROADCAST_CODE_SIZE*]

The bad broadcast code.

Only valid if encrypt_state is *BT_BAP_BIG_ENC_STATE_BCODE_REQ*

uint8_t **num_subgroups**

Number of subgroups.

struct *bt_bap_bass_subgroup* **subgroups**[CONFIG_BT_BAP_BASS_MAX_SUBGROUPS]

Subgroup specific information.

struct **bt_bap_scan_delegator_cb**

#include <*bap.h*> Struct to hold the Basic Audio Profile Scan Delegator callbacks.

These can be registered for usage with *bt_bap_scan_delegator_register_cb()*.

Public Members

void (***recv_state_updated**)(struct bt_conn *conn, const struct *bt_bap_scan_delegator_recv_state* *recv_state)

Receive state updated.

Param conn
Pointer to the connection to a remote device if the change was caused by it, otherwise NULL.

Param recv_state
Pointer to the receive state that was updated.

Return
0 in case of success or negative value in case of error.

int (***pa_sync_req**)(struct bt_conn *conn, const struct *bt_bap_scan_delegator_recv_state* *recv_state, bool past_avail, uint16_t pa_interval)

Periodic advertising sync request.

Request from peer device to synchronize with the periodic advertiser denoted by the `recv_state`. To notify the Broadcast Assistant about any pending sync

Param conn
Pointer to the connection requesting the periodic advertising sync.

Param recv_state
Pointer to the receive state that is being requested for periodic advertising sync.

Param past_avail
True if periodic advertising sync transfer is available.

Param pa_interval
The periodic advertising interval.

Return
0 in case of accept, or other value to reject.

int (***pa_sync_term_req**)(struct bt_conn *conn, const struct *bt_bap_scan_delegator_recv_state* *recv_state)

Periodic advertising sync termination request.

Request from peer device to terminate the periodic advertiser sync denoted by the `recv_state`.

Param conn
Pointer to the connection requesting the periodic advertising sync termination.

Param recv_state
Pointer to the receive state that is being requested for periodic advertising sync.

Return
0 in case of success or negative value in case of error.

void (*`broadcast_code`)(struct bt_conn *conn, const struct *bt_bap_scan_delegator_recv_state* *recv_state, const uint8_t broadcast_code[*BT_AUDIO_BROADCAST_CODE_SIZE*])

Broadcast code received.

Broadcast code received from a broadcast assistant

Param conn
Pointer to the connection providing the broadcast code.
Param recv_state
Pointer to the receive state the broadcast code is being provided for.
Param broadcast_code
The 16-octet broadcast code

int (*`bis_sync_req`)(struct bt_conn *conn, const struct *bt_bap_scan_delegator_recv_state* *recv_state, const uint32_t bis_sync_req[CONFIG_BT_BAP_BASS_MAX_SUBGROUPS])

Broadcast Isochronous Stream synchronize request.

Request from Broadcast Assistant device to modify the Broadcast Isochronous Stream states. The request shall be fulfilled with accordance to the `bis_sync_req` within reasonable time. The Broadcast Assistant may also request fewer, or none, indexes to be synchronized.

Param conn
[in] Pointer to the connection of the Broadcast Assistant requesting the sync.
Param recv_state
[in] Pointer to the receive state that is being requested for the sync.
Param bis_sync_req
[in] Array of bitfields of which BIS indexes that is requested to sync for each subgroup by the Broadcast Assistant. A value of 0 indicates a request to terminate the BIG sync.
Return
0 in case of accept, or other value to reject.

struct `bt_bap_ep_info`

#include <*bap.h*> Structure holding information of audio stream endpoint.

Public Members

uint8_t `id`

The ID of the endpoint.

enum *bt_bap_ep_state* `state`

The state of the endpoint.

enum *bt_audio_dir* `dir`

Capabilities type.

struct bt_iso_chan *`iso_chan`

The isochronous channel associated with the endpoint.

bool **can_send**

True if the stream associated with the endpoint is able to send data.

bool **can_recv**

True if the stream associated with the endpoint is able to receive data.

struct bt_bap_ep ***paired_ep**

Pointer to paired endpoint if the endpoint is part of a bidirectional CIS, otherwise NULL.

const struct *bt_audio_codec_qos_pref* ***qos_pref**

Pointer to the preferred QoS settings associated with the endpoint.

struct **bt_bap_stream**

#include <bap.h> Basic Audio Profile stream structure.

Streams represents a stream configuration of a Remote Endpoint and a Local Capability.

Note

Streams are unidirectional but can be paired with other streams to use a bidirectional connected isochronous stream.

Public Members

struct bt_conn ***conn**

Connection reference.

struct bt_bap_ep ***ep**

Endpoint reference.

struct *bt_audio_codec_cfg* ***codec_cfg**

Codec Configuration.

struct *bt_audio_codec_qos* ***qos**

QoS Configuration.

struct *bt_bap_stream_ops* ***ops**

Audio stream operations.

void ***user_data**

Stream user data.

struct **bt_bap_stream_ops**

#include <bap.h> Stream operation.

Public Members

void (*`configured`)(struct *bt_bap_stream* *stream, const struct *bt_audio_codec_qos_pref* *pref)

Stream configured callback.

Configured callback is called whenever an Audio Stream has been configured.

Param stream
Stream object that has been configured.

Param pref
Remote QoS preferences.

void (*`qos_set`)(struct *bt_bap_stream* *stream)

Stream QoS set callback.

QoS set callback is called whenever an Audio Stream Quality of Service has been set or updated.

Param stream
Stream object that had its QoS updated.

void (*`enabled`)(struct *bt_bap_stream* *stream)

Stream enabled callback.

Enabled callback is called whenever an Audio Stream has been enabled.

Param stream
Stream object that has been enabled.

void (*`metadata_updated`)(struct *bt_bap_stream* *stream)

Stream metadata updated callback.

Metadata Updated callback is called whenever an Audio Stream's metadata has been updated.

Param stream
Stream object that had its metadata updated.

void (*`disabled`)(struct *bt_bap_stream* *stream)

Stream disabled callback.

Disabled callback is called whenever an Audio Stream has been disabled.

Param stream
Stream object that has been disabled.

void (*`released`)(struct *bt_bap_stream* *stream)

Stream released callback.

Released callback is called whenever a Audio Stream has been released and can be deallocated.

Param stream
Stream object that has been released.

void (*`started`)(struct *bt_bap_stream* *stream)

Stream started callback.

Started callback is called whenever an Audio Stream has been started and will be usable for streaming.

Param stream
Stream object that has been started.

void (*stopped)(struct *bt_bap_stream* *stream, uint8_t reason)

Stream stopped callback.

Stopped callback is called whenever an Audio Stream has been stopped.

Param stream
Stream object that has been stopped.

Param reason
BT_HCI_ERR_* reason for the disconnection.

void (*recv)(struct *bt_bap_stream* *stream, const struct bt_iso_recv_info *info, struct *net_buf* *buf)

Stream audio HCI receive callback.

This callback is only used if the ISO data path is HCI.

Param stream
Stream object.

Param info
Pointer to the metadata for the buffer. The lifetime of the pointer is linked to the lifetime of the *net_buf*. Metadata such as sequence number and timestamp can be provided by the bluetooth controller.

Param buf
Buffer containing incoming audio data.

void (*sent)(struct *bt_bap_stream* *stream)

Stream audio HCI sent callback.

This callback will be called once the controller marks the SDU as completed. When the controller does so is implementation dependent. It could be after the SDU is enqueued for transmission, or after it is sent on air or flushed.

This callback is only used if the ISO data path is HCI.

Param stream
Stream object.

void (*connected)(struct *bt_bap_stream* *stream)

Isochronous channel connected callback.

If this callback is provided it will be called whenever the isochronous channel for the stream has been connected. This does not mean that the stream is ready to be used, which is indicated by the *bt_bap_stream_ops::started* callback.

If the stream shares an isochronous channel with another stream, then this callback may still be called, without the stream going into the started state.

Param stream
Stream object.

void (*disconnected)(struct *bt_bap_stream* *stream, uint8_t reason)

Isochronous channel disconnected callback.

If this callback is provided it will be called whenever the isochronous channel is disconnected, including when a connection gets rejected.

If the stream shares an isochronous channel with another stream, then this callback may not be called, even if the stream is leaving the streaming state.

Param stream
Stream object.

Param reason
BT_HCI_ERR_* reason for the disconnection.

struct **bt_bap_scan_delegator_add_src_param**

#include <bap.h> Parameters for *bt_bap_scan_delegator_add_src()*

Public Members

struct bt_le_per_adv_sync ***pa_sync**

The periodic adverting sync.

enum *bt_bap_big_enc_state* **encrypt_state**

The broadcast isochronous group encryption state.

uint32_t **broadcast_id**

The 24-bit broadcast ID.

uint8_t **num_subgroups**

Number of subgroups.

struct *bt_bap_bass_subgroup* **subgroups**[CONFIG_BT_BAP_BASS_MAX_SUBGROUPS]

Subgroup specific information.

struct **bt_bap_scan_delegator_mod_src_param**

#include <bap.h> Parameters for *bt_bap_scan_delegator_mod_src()*

Public Members

uint8_t **src_id**

The periodic adverting sync.

enum *bt_bap_big_enc_state* **encrypt_state**

The broadcast isochronous group encryption state.

uint32_t **broadcast_id**

The 24-bit broadcast ID.

uint8_t **num_subgroups**

Number of subgroups.

struct *bt_bap_bass_subgroup* **subgroups**[CONFIG_BT_BAP_BASS_MAX_SUBGROUPS]

Subgroup specific information.

If a subgroup's metadata_len is set to 0, the existing metadata for the subgroup will remain unchanged

struct **bt_bap_broadcast_assistant_cb**

#include <bap.h> Struct to hold the Basic Audio Profile Broadcast Assistant callbacks.

These can be registered for usage with *bt_bap_broadcast_assistant_register_cb()*.

Public Members

void (*discover)(struct bt_conn *conn, int err, uint8_t recv_state_count)

Callback function for bt_bap_broadcast_assistant_discover.

Param conn
The connection that was used to discover Broadcast Audio Scan Service.

Param err
Error value. 0 on success, GATT error or ERRNO on fail.

Param recv_state_count
Number of receive states on the server.

void (*scan)(const struct *bt_le_scan_recv_info* *info, uint32_t broadcast_id)

Callback function for Broadcast Audio Scan Service client scan results.

Called when the scanner finds an advertiser that advertises the BT_UUID_BROADCAST_AUDIO UUID.

Param info
Advertiser information.

Param broadcast_id
24-bit broadcast ID.

void (*recv_state)(struct bt_conn *conn, int err, const struct *bt_bap_scan_delegator_recv_state* *state)

Callback function for when a receive state is read or updated.

Called whenever a receive state is read or updated.

Param conn
The connection to the Broadcast Audio Scan Service server.

Param err
Error value. 0 on success, GATT error on fail.

Param state
The receive state or NULL if the receive state is empty.

void (*recv_state_removed)(struct bt_conn *conn, uint8_t src_id)

Callback function for when a receive state is removed.

Param conn
The connection to the Broadcast Audio Scan Service server.

Param src_id
The receive state.

void (*scan_start)(struct bt_conn *conn, int err)

Callback function for *bt_bap_broadcast_assistant_scan_start()*.

Param conn
The connection to the peer device.

Param err
Error value. 0 on success, GATT error on fail.

void (*scan_stop)(struct bt_conn *conn, int err)

Callback function for *bt_bap_broadcast_assistant_scan_stop()*.

Param conn
The connection to the peer device.

Param err
Error value. 0 on success, GATT error on fail.

void (*add_src)(struct bt_conn *conn, int err)

Callback function for *bt_bap_broadcast_assistant_add_src()*.

Param conn
The connection to the peer device.

Param err
Error value. 0 on success, GATT error on fail.

void (*`mod_src`)(struct bt_conn *conn, int err)

Callback function for *bt_bap_broadcast_assistant_mod_src()*.

Param conn
The connection to the peer device.

Param err
Error value. 0 on success, GATT error on fail.

void (*`broadcast_code`)(struct bt_conn *conn, int err)

Callback function for bt_bap_broadcast_assistant_broadcast_code().

Param conn
The connection to the peer device.

Param err
Error value. 0 on success, GATT error on fail.

void (*`rem_src`)(struct bt_conn *conn, int err)

Callback function for *bt_bap_broadcast_assistant_rem_src()*.

Param conn
The connection to the peer device.

Param err
Error value. 0 on success, GATT error on fail.

struct `bt_bap_broadcast_assistant_add_src_param`

#include <bap.h> Parameters for adding a source to a Broadcast Audio Scan Service server.

Public Members

bt_addr_le_t `addr`

Address of the advertiser.

uint8_t `adv_sid`

SID of the advertising set.

bool `pa_sync`

Whether to sync to periodic advertisements.

uint32_t `broadcast_id`

24-bit broadcast ID

uint16_t `pa_interval`

Periodic advertising interval in milliseconds.

BT_BAP_PA_INTERVAL_UNKNOWN if unknown.

uint8_t **num_subgroups**
Number of subgroups.

struct *bt_bap_bass_subgroup* ***subgroups**
Pointer to array of subgroups.

struct **bt_bap_broadcast_assistant_mod_src_param**
#include <*bap.h*> Parameters for modifying a source.

Public Members

uint8_t **src_id**
Source ID of the receive state.

bool **pa_sync**
Whether to sync to periodic advertisements.

uint16_t **pa_interval**
Periodic advertising interval.

BT_BAP_PA_INTERVAL_UNKNOWN if unknown.

uint8_t **num_subgroups**
Number of subgroups.

struct *bt_bap_bass_subgroup* ***subgroups**
Pointer to array of subgroups.

group **bt_bap_unicast_client**

Functions

int **bt_bap_unicast_group_create**(struct *bt_bap_unicast_group_param* *param, struct bt_bap_unicast_group **unicast_group)
Create audio unicast group.

Create a new audio unicast group with one or more audio streams as a unicast client. Streams in a unicast group shall share the same interval, framing and latency (see *bt_audio_codec_qos*).

Parameters

- **param** – **[in]** The unicast group create parameters.
- **unicast_group** – **[out]** Pointer to the unicast group created.

Returns
Zero on success or (negative) error code otherwise.

int **bt_bap_unicast_group_add_streams**(struct bt_bap_unicast_group *unicast_group, struct *bt_bap_unicast_group_stream_pair_param* params[], size_t num_param)

Add streams to a unicast group as a unicast client.

This function can be used to add additional streams to a bt_bap_unicast_group.

This can be called at any time before any of the streams in the group has been started (see *bt_bap_stream_ops.started()*). This can also be called after the streams have been stopped (see *bt_bap_stream_ops.stopped()*).

Once a stream has been added to a unicast group, it cannot be removed. To remove a stream from a group, the group must be deleted with *bt_bap_unicast_group_delete()*, but this will require all streams in the group to be released first.

Parameters

- **unicast_group** – Pointer to the unicast group
- **params** – Array of stream parameters with streams being added to the group.
- **num_param** – Number of parameters in `params`.

Returns
0 in case of success or negative value in case of error.

int **bt_bap_unicast_group_delete**(struct bt_bap_unicast_group *unicast_group)

Delete audio unicast group.

Delete a audio unicast group as a client. All streams in the group shall be in the idle or configured state.

Parameters

- **unicast_group** – Pointer to the unicast group to delete

Returns
Zero on success or (negative) error code otherwise.

int **bt_bap_unicast_client_register_cb**(const struct *bt_bap_unicast_client_cb* *cb)

Register unicast client callbacks.

Only one callback structure can be registered, and attempting to registering more than one will result in an error.

Parameters

- **cb** – Unicast client callback structure.

Returns
0 in case of success or negative value in case of error.

int **bt_bap_unicast_client_discover**(struct bt_conn *conn, enum *bt_audio_dir* dir)

Discover remote capabilities and endpoints.

This procedure is used by a client to discover remote capabilities and endpoints and notifies via params callback.

Parameters

- **conn** – Connection object
- **dir** – The type of remote endpoints and capabilities to discover.

struct **bt_bap_unicast_group_stream_param**

#include <bap.h> Parameter struct for each stream in the unicast group.

Public Members

struct *bt_bap_stream* ***stream**

Pointer to a stream object.

struct *bt_audio_codec_qos* ***qos**

The QoS settings for the stream object.

struct **bt_bap_unicast_group_stream_pair_param**

#include <*bap.h*> Parameter struct for the unicast group functions.

Parameter struct for the *bt_bap_unicast_group_create()* and *bt_bap_unicast_group_add_streams()* functions.

Public Members

struct *bt_bap_unicast_group_stream_param* ***rx_param**

Pointer to a receiving stream parameters.

struct *bt_bap_unicast_group_stream_param* ***tx_param**

Pointer to a transmitting stream parameters.

struct **bt_bap_unicast_group_param**

#include <*bap.h*> Parameters for the creating unicast groups with *bt_bap_unicast_group_create()*

Public Members

size_t **params_count**

The number of parameters in `params`.

struct *bt_bap_unicast_group_stream_pair_param* ***params**

Array of stream parameters.

uint8_t **packing**

Unicast Group packing mode.

BT_ISO_PACKING_SEQUENTIAL or BT_ISO_PACKING_INTERLEAVED.

> **Note**
>
> This is a recommendation to the controller, which the controller may ignore.

uint8_t **c_to_p_ft**

Central to Peripheral flush timeout.

The flush timeout in multiples of ISO_Interval for each payload sent from the Central to Peripheral.

Value range from BT_ISO_FT_MIN to BT_ISO_FT_MAX

uint8_t **p_to_c_ft**

Peripheral to Central flush timeout.

The flush timeout in multiples of ISO_Interval for each payload sent from the Peripheral to Central.

Value range from BT_ISO_FT_MIN to BT_ISO_FT_MAX.

uint16_t **iso_interval**

ISO interval.

Time between consecutive CIS anchor points.

Value range from BT_ISO_ISO_INTERVAL_MIN to BT_ISO_ISO_INTERVAL_MAX.

struct **bt_bap_unicast_client_cb**

#include <*bap.h*> Unicast Client callback structure.

Public Members

void (***location**)(struct bt_conn *conn, enum *bt_audio_dir* dir, enum *bt_audio_location* loc)

Remote Unicast Server Audio Locations.

This callback is called whenever the audio locations is read from the server or otherwise notified to the client.

Param conn
Connection to the remote unicast server.

Param dir
Direction of the location.

Param loc
The location bitfield value.

Return
0 in case of success or negative value in case of error.

void (***available_contexts**)(struct bt_conn *conn, enum *bt_audio_context* snk_ctx, enum *bt_audio_context* src_ctx)

Remote Unicast Server Available Contexts.

This callback is called whenever the available contexts are read from the server or otherwise notified to the client.

Param conn
Connection to the remote unicast server.

Param snk_ctx
The sink context bitfield value.

Param src_ctx
The source context bitfield value.

Return
0 in case of success or negative value in case of error.

void (***config**)(struct *bt_bap_stream* *stream, enum *bt_bap_ascs_rsp_code* rsp_code, enum *bt_bap_ascs_reason* reason)

Callback function for *bt_bap_stream_config()* and *bt_bap_stream_reconfig()*.

Called when the codec configure operation is completed on the server.

Param stream
Stream the operation was performed on.

Param rsp_code
Response code.
Param reason
Reason code.

void (*qos)(struct *bt_bap_stream* *stream, enum *bt_bap_ascs_rsp_code* rsp_code, enum *bt_bap_ascs_reason* reason)

Callback function for *bt_bap_stream_qos()*.

Called when the QoS configure operation is completed on the server. This will be called for each stream in the group that was being QoS configured.
Param stream
Stream the operation was performed on. May be NULL if there is no stream associated with the ASE ID sent by the server.
Param rsp_code
Response code.
Param reason
Reason code.

void (*enable)(struct *bt_bap_stream* *stream, enum *bt_bap_ascs_rsp_code* rsp_code, enum *bt_bap_ascs_reason* reason)

Callback function for *bt_bap_stream_enable()*.

Called when the enable operation is completed on the server.
Param stream
Stream the operation was performed on. May be NULL if there is no stream associated with the ASE ID sent by the server.
Param rsp_code
Response code.
Param reason
Reason code.

void (*start)(struct *bt_bap_stream* *stream, enum *bt_bap_ascs_rsp_code* rsp_code, enum *bt_bap_ascs_reason* reason)

Callback function for *bt_bap_stream_start()*.

Called when the start operation is completed on the server. This will only be called if the stream supplied to *bt_bap_stream_start()* is for a *BT_AUDIO_DIR_SOURCE* endpoint.
Param stream
Stream the operation was performed on. May be NULL if there is no stream associated with the ASE ID sent by the server.
Param rsp_code
Response code.
Param reason
Reason code.

void (*stop)(struct *bt_bap_stream* *stream, enum *bt_bap_ascs_rsp_code* rsp_code, enum *bt_bap_ascs_reason* reason)

Callback function for *bt_bap_stream_stop()*.

Called when the stop operation is completed on the server. This will only be called if the stream supplied to *bt_bap_stream_stop()* is for a *BT_AUDIO_DIR_SOURCE* endpoint.
Param stream
Stream the operation was performed on. May be NULL if there is no stream associated with the ASE ID sent by the server.

Param rsp_code
Response code.
Param reason
Reason code.

void (*`disable`)(struct *bt_bap_stream* *stream, enum *bt_bap_ascs_rsp_code* rsp_code, enum *bt_bap_ascs_reason* reason)

Callback function for *bt_bap_stream_disable()*.

Called when the disable operation is completed on the server.
Param stream
Stream the operation was performed on. May be NULL if there is no stream associated with the ASE ID sent by the server.
Param rsp_code
Response code.
Param reason
Reason code.

void (*`metadata`)(struct *bt_bap_stream* *stream, enum *bt_bap_ascs_rsp_code* rsp_code, enum *bt_bap_ascs_reason* reason)

Callback function for *bt_bap_stream_metadata()*.

Called when the metadata operation is completed on the server.
Param stream
Stream the operation was performed on. May be NULL if there is no stream associated with the ASE ID sent by the server.
Param rsp_code
Response code.
Param reason
Reason code.

void (*`release`)(struct *bt_bap_stream* *stream, enum *bt_bap_ascs_rsp_code* rsp_code, enum *bt_bap_ascs_reason* reason)

Callback function for *bt_bap_stream_release()*.

Called when the release operation is completed on the server.
Param stream
Stream the operation was performed on. May be NULL if there is no stream associated with the ASE ID sent by the server.
Param rsp_code
Response code.
Param reason
Reason code.

void (*`pac_record`)(struct bt_conn *conn, enum *bt_audio_dir* dir, const struct *bt_audio_codec_cap* *codec_cap)

Remote Published Audio Capability (PAC) record discovered.

Called when a PAC record has been discovered as part of the discovery procedure.

The `codec` is only valid while in the callback, so the values must be stored by the receiver if future use is wanted.

If discovery procedure has complete both `codec` and `ep` are set to NULL.
Param conn
Connection to the remote unicast server.
Param dir
The type of remote endpoints and capabilities discovered.

Param codec_cap
Remote capabilities.

void (*`endpoint`)(struct bt_conn *conn, enum *bt_audio_dir* dir, struct bt_bap_ep *ep)

Remote Audio Stream Endpoint (ASE) discovered.

Called when an ASE has been discovered as part of the discovery procedure.

If discovery procedure has complete both `codec` and `ep` are set to NULL.

Param conn
Connection to the remote unicast server.
Param dir
The type of remote endpoints and capabilities discovered.
Param ep
Remote endpoint.

void (*`discover`)(struct bt_conn *conn, int err, enum *bt_audio_dir* dir)

BAP discovery callback function.

If discovery procedure has completed `ep` is set to NULL and `err` is 0.

If discovery procedure has complete both `codec` and `ep` are set to NULL.

Param conn
Connection to the remote unicast server.
Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.
Param dir
The type of remote endpoints and capabilities discovered.

group `bt_bap_unicast_server`

Typedefs

typedef void (*`bt_bap_ep_func_t`)(struct bt_bap_ep *ep, void *user_data)

The callback function called for each endpoint.

Param ep
The structure object with endpoint info.

Param user_data
Data to pass to the function.

Functions

int `bt_bap_unicast_server_register_cb`(const struct *bt_bap_unicast_server_cb* *cb)

Register unicast server callbacks.

Only one callback structure can be registered, and attempting to registering more than one will result in an error.

Parameters

- `cb` – Unicast server callback structure.

Returns
0 in case of success or negative value in case of error.

int **bt_bap_unicast_server_unregister_cb**(const struct *bt_bap_unicast_server_cb* *cb)

Unregister unicast server callbacks.

May only unregister a callback structure that has previously been registered by *bt_bap_unicast_server_register_cb()*.

Parameters

- **cb** – Unicast server callback structure.

Returns
0 in case of success or negative value in case of error.

void **bt_bap_unicast_server_foreach_ep**(struct bt_conn *conn, *bt_bap_ep_func_t* func, void *user_data)

Iterate through all endpoints of the given connection.

Parameters

- **conn** – Connection object
- **func** – Function to call for each endpoint.
- **user_data** – Data to pass to the callback function.

int **bt_bap_unicast_server_config_ase**(struct bt_conn *conn, struct *bt_bap_stream* *stream, struct *bt_audio_codec_cfg* *codec_cfg, const struct *bt_audio_codec_qos_pref* *qos_pref)

Initialize and configure a new ASE.

Parameters

- **conn** – Connection object
- **stream** – Configured stream object to be attached to the ASE
- **codec_cfg** – Codec configuration
- **qos_pref** – Audio Stream Quality of Service Preference

Returns
0 in case of success or negative value in case of error.

struct **bt_bap_unicast_server_cb**

#include <bap.h> Unicast Server callback structure.

Public Members

int (***config**)(struct bt_conn *conn, const struct bt_bap_ep *ep, enum *bt_audio_dir* dir, const struct *bt_audio_codec_cfg* *codec_cfg, struct *bt_bap_stream* **stream, struct *bt_audio_codec_qos_pref* *const pref, struct *bt_bap_ascs_rsp* *rsp)

Endpoint config request callback.

Config callback is called whenever an endpoint is requested to be configured
Param conn
[in] Connection object.
Param ep
[in] Local Audio Endpoint being configured.
Param dir
[in] Direction of the endpoint.

Param codec_cfg
[in] Codec configuration.
Param stream
[out] Pointer to stream that will be configured for the endpoint.
Param pref
[out] Pointer to a QoS preference object that shall be populated with values. Invalid values will reject the codec configuration request.
Param rsp
[out] Object for the ASE operation response. Only used if the return value is non-zero.
Return
0 in case of success or negative value in case of error.

int (*`reconfig`)(struct *bt_bap_stream* *stream, enum *bt_audio_dir* dir, const struct *bt_audio_codec_cfg* *codec_cfg, struct *bt_audio_codec_qos_pref* *const pref, struct *bt_bap_ascs_rsp* *rsp)

Stream reconfig request callback.

Reconfig callback is called whenever an Audio Stream needs to be reconfigured with different codec configuration.

Param stream
[in] Stream object being reconfigured.
Param dir
[in] Direction of the endpoint.
Param codec_cfg
[in] Codec configuration.
Param pref
[out] Pointer to a QoS preference object that shall be populated with values. Invalid values will reject the codec configuration request.
Param rsp
[out] Object for the ASE operation response. Only used if the return value is non-zero.
Return
0 in case of success or negative value in case of error.

int (*`qos`)(struct *bt_bap_stream* *stream, const struct *bt_audio_codec_qos* *qos, struct *bt_bap_ascs_rsp* *rsp)

Stream QoS request callback.

QoS callback is called whenever an Audio Stream Quality of Service needs to be configured.

Param stream
[in] Stream object being reconfigured.
Param qos
[in] Quality of Service configuration.
Param rsp
[out] Object for the ASE operation response. Only used if the return value is non-zero.
Return
0 in case of success or negative value in case of error.

int (*`enable`)(struct *bt_bap_stream* *stream, const uint8_t meta[], size_t meta_len, struct *bt_bap_ascs_rsp* *rsp)

Stream Enable request callback.

Enable callback is called whenever an Audio Stream is requested to be enabled to stream.

Param stream
[in] Stream object being enabled.
Param meta
[in] Metadata entries.
Param meta_len
[in] Length of metadata.
Param rsp
[out] Object for the ASE operation response. Only used if the return value is non-zero.
Return
0 in case of success or negative value in case of error.

int (*`start`)(struct *bt_bap_stream* *stream, struct *bt_bap_ascs_rsp* *rsp)

Stream Start request callback.

Start callback is called whenever an Audio Stream is requested to start streaming.

Param stream
[in] Stream object.
Param rsp
[out] Object for the ASE operation response. Only used if the return value is non-zero.
Return
0 in case of success or negative value in case of error.

int (*`metadata`)(struct *bt_bap_stream* *stream, const uint8_t meta[], size_t meta_len, struct *bt_bap_ascs_rsp* *rsp)

Stream Metadata update request callback.

Metadata callback is called whenever an Audio Stream is requested to update its metadata.

Param stream
[in] Stream object.
Param meta
[in] Metadata entries.
Param meta_len
[in] Length of metadata.
Param rsp
[out] Object for the ASE operation response. Only used if the return value is non-zero.
Return
0 in case of success or negative value in case of error.

int (*`disable`)(struct *bt_bap_stream* *stream, struct *bt_bap_ascs_rsp* *rsp)

Stream Disable request callback.

Disable callback is called whenever an Audio Stream is requested to disable the stream.

Param stream
[in] Stream object being disabled.
Param rsp
[out] Object for the ASE operation response. Only used if the return value is non-zero.
Return
0 in case of success or negative value in case of error.

int (*`stop`)(struct *bt_bap_stream* *stream, struct *bt_bap_ascs_rsp* *rsp)

Stream Stop callback.

Stop callback is called whenever an Audio Stream is requested to stop streaming.

Param stream
[in] Stream object.

Param rsp
[out] Object for the ASE operation response. Only used if the return value is non-zero.

Return
0 in case of success or negative value in case of error.

int (*`release`)(struct *bt_bap_stream* *stream, struct *bt_bap_ascs_rsp* *rsp)

Stream release callback.

Release callback is called whenever a new Audio Stream needs to be released and thus deallocated.

Param stream
[in] Stream object.

Param rsp
[out] Object for the ASE operation response. Only used if the return value is non-zero.

Return
0 in case of success or negative value in case of error.

group `bt_bap_broadcast`

BAP Broadcast APIs.

Functions

const struct bt_bap_base *`bt_bap_base_get_base_from_ad`(const struct *bt_data* *ad)

Generate a pointer to a BASE from periodic advertising data.

Parameters

- `ad` – The periodic advertising data

Return values

- `NULL` – if the data does not contain a BASE
- `Pointer` – to a bt_bap_base structure

int `bt_bap_base_get_size`(const struct bt_bap_base *base)

Get the size of a BASE.

Parameters

- `base` – The BASE pointer

Return values

- `-EINVAL` – if arguments are invalid
- `The` – size of the BASE

int `bt_bap_base_get_pres_delay`(const struct bt_bap_base *base)

Get the presentation delay value of a BASE.

Parameters

- `base` – The BASE pointer

Return values

- `-EINVAL` – if arguments are invalid

- **The** – 24-bit presentation delay value

int **bt_bap_base_get_subgroup_count**(const struct bt_bap_base *base)

Get the subgroup count of a BASE.

Parameters

- **base** – The BASE pointer

Return values

- **-EINVAL** – if arguments are invalid
- **The** – 8-bit subgroup count value

int **bt_bap_base_get_bis_indexes**(const struct bt_bap_base *base, uint32_t *bis_indexes)

Get all BIS indexes of a BASE.

Parameters

- **base** – **[in]** The BASE pointer
- **bis_indexes** – **[out]** 32-bit BIS index bitfield that will be populated

Return values

- **-EINVAL** – if arguments are invalid
- **0** – on success

int **bt_bap_base_foreach_subgroup**(const struct bt_bap_base *base, bool (*func)(const struct bt_bap_base_subgroup *subgroup, void *user_data), void *user_data)

Iterate on all subgroups in the BASE.

Parameters

- **base** – The BASE pointer
- **func** – Callback function. Return true to continue iterating, or false to stop.
- **user_data** – Userdata supplied to `func`

Return values

- **-EINVAL** – if arguments are invalid
- **-ECANCELED** – if iterating over the subgroups stopped prematurely by `func`
- **0** – if all subgroups were iterated

int **bt_bap_base_get_subgroup_codec_id**(const struct bt_bap_base_subgroup *subgroup, struct *bt_bap_base_codec_id* *codec_id)

Get the codec ID of a subgroup.

Parameters

- **subgroup** – **[in]** The subgroup pointer
- **codec_id** – **[out]** Pointer to the struct where the results are placed

Return values

- **-EINVAL** – if arguments are invalid
- **0** – on success

int **bt_bap_base_get_subgroup_codec_data**(const struct bt_bap_base_subgroup *subgroup, uint8_t **data)

Get the codec configuration data of a subgroup.

Parameters

- **subgroup** – **[in]** The subgroup pointer
- **data** – **[out]** Pointer that will point to the resulting codec configuration data

Return values

- **-EINVAL** – if arguments are invalid
- **0** – on success

int **bt_bap_base_get_subgroup_codec_meta**(const struct bt_bap_base_subgroup *subgroup, uint8_t **meta)

Get the codec metadata of a subgroup.

Parameters

- **subgroup** – **[in]** The subgroup pointer
- **meta** – **[out]** Pointer that will point to the resulting codec metadata

Return values

- **-EINVAL** – if arguments are invalid
- **0** – on success

int **bt_bap_base_subgroup_codec_to_codec_cfg**(const struct bt_bap_base_subgroup *subgroup, struct *bt_audio_codec_cfg* *codec_cfg)

Store subgroup codec data in a *Codec config parsing APIs*.

Parameters

- **subgroup** – **[in]** The subgroup pointer
- **codec_cfg** – **[out]** Pointer to the struct where the results are placed

Return values

- **-EINVAL** – if arguments are invalid
- **-ENOMEM** – if the codec_cfg cannot store the subgroup codec data
- **0** – on success

int **bt_bap_base_get_subgroup_bis_count**(const struct bt_bap_base_subgroup *subgroup)

Get the BIS count of a subgroup.

Parameters

- **subgroup** – The subgroup pointer

Return values

- **-EINVAL** – if arguments are invalid
- **The** – 8-bit BIS count value

int **bt_bap_base_subgroup_get_bis_indexes**(const struct bt_bap_base_subgroup *subgroup, uint32_t *bis_indexes)

Get all BIS indexes of a subgroup.

Parameters

- **subgroup** – **[in]** The subgroup pointer
- **bis_indexes** – **[out]** 32-bit BIS index bitfield that will be populated

Return values

- **-EINVAL** – if arguments are invalid
- **0** – on success

int **bt_bap_base_subgroup_foreach_bis**(const struct bt_bap_base_subgroup *subgroup, bool (*func)(const struct *bt_bap_base_subgroup_bis* *bis, void *user_data), void *user_data)

Iterate on all BIS in the subgroup.

Parameters

- **subgroup** – The subgroup pointer
- **func** – Callback function. Return true to continue iterating, or false to stop.
- **user_data** – Userdata supplied to `func`

Return values

- **-EINVAL** – if arguments are invalid
- **-ECANCELED** – if iterating over the subgroups stopped prematurely by `func`
- **0** – if all BIS were iterated

int **bt_bap_base_subgroup_bis_codec_to_codec_cfg**(const struct *bt_bap_base_subgroup_bis* *bis, struct *bt_audio_codec_cfg* *codec_cfg)

Store BIS codec configuration data in a *Codec config parsing APIs*.

This only sets the *Codec config parsing APIs* data and *Codec config parsing APIs* data_len, but is useful to use the BIS codec configuration data with the bt_audio_codec_cfg_* functions.

Parameters

- **bis** – **[in]** The BIS pointer
- **codec_cfg** – **[out]** Pointer to the struct where the results are placed

Return values

- **-EINVAL** – if arguments are invalid
- **-ENOMEM** – if the `codec_cfg` cannot store the `subgroup` codec data
- **0** – on success

struct **bt_bap_base_codec_id**

#include <bap.h> Codec ID structure for a Broadcast Audio Source Endpoint (BASE)

Public Members

uint8_t **id**

Codec ID.

uint16_t **cid**

Codec Company ID.

uint16_t **vid**

Codec Company Vendor ID.

struct **bt_bap_base_subgroup_bis**

#include <*bap.h*> BIS structure for each BIS in a Broadcast Audio Source Endpoint (BASE) subgroup.

Public Members

uint8_t **index**

Unique index of the BIS.

uint8_t **data_len**

Codec Specific Data length.

uint8_t ***data**

Codec Specific Data.

group **bt_bap_broadcast_sink**

BAP Broadcast Sink APIs.

Functions

int **bt_bap_broadcast_sink_register_cb**(struct *bt_bap_broadcast_sink_cb* *cb)

Register Broadcast sink callbacks.

It is possible to register multiple struct of callbacks, but a single struct can only be registered once. Registering the same callback multiple times is undefined behavior and may break the stack.

Parameters

- **cb** – Broadcast sink callback structure.

Return values

- **0** – in case of success
- **-EINVAL** – if cb is NULL

int **bt_bap_broadcast_sink_create**(struct bt_le_per_adv_sync *pa_sync, uint32_t broadcast_id, struct bt_bap_broadcast_sink **sink)

Create a Broadcast Sink from a periodic advertising sync.

This should only be done after verifying that the periodic advertising sync is from a Broadcast Source.

The created Broadcast Sink will need to be supplied to *bt_bap_broadcast_sink_sync()* in order to synchronize to the broadcast audio.

bt_bap_broadcast_sink_cb.pa_synced() will be called with the Broadcast Sink object created if this is successful.

Parameters

- `pa_sync` – Pointer to the periodic advertising sync object.
- `broadcast_id` – 24-bit broadcast ID.
- `sink` – **[out]** Pointer to the Broadcast Sink created.

Returns
0 in case of success or errno value in case of error.

int `bt_bap_broadcast_sink_sync`(struct bt_bap_broadcast_sink *sink, uint32_t indexes_bitfield, struct *bt_bap_stream* *streams[], const uint8_t broadcast_code[16])

Sync to a broadcaster's audio.

Example: The string "Broadcast Code" shall be [42 72 6F 61 64 63 61 73 74 20 43 6F 64 65 00 00]

Parameters

- `sink` – Pointer to the sink object from the base_recv callback.
- `indexes_bitfield` – Bitfield of the BIS index to sync to. To sync to e.g. BIS index 1 and 2, this should have the value of *BIT(1)* | *BIT(2)*.
- `streams` – Stream object pointers to be used for the receiver. If multiple BIS indexes shall be synchronized, multiple streams shall be provided.
- `broadcast_code` – The 16-octet broadcast code. Shall be supplied if the broadcast is encrypted (see *bt_bap_broadcast_sink_cb::syncable*). If the value is a string or a the value is less than 16 octets, the remaining octets shall be 0.

Returns
0 in case of success or negative value in case of error.

int `bt_bap_broadcast_sink_stop`(struct bt_bap_broadcast_sink *sink)

Stop audio broadcast sink.

Stop an audio broadcast sink. The broadcast sink will stop receiving BIGInfo, and audio data can no longer be streamed.

Parameters

- `sink` – Pointer to the broadcast sink

Returns
Zero on success or (negative) error code otherwise.

int `bt_bap_broadcast_sink_delete`(struct bt_bap_broadcast_sink *sink)

Release a broadcast sink.

Once a broadcast sink has been allocated after the pa_synced callback, it can be deleted using this function. If the sink has synchronized to any broadcast audio streams, these must first be stopped using bt_bap_stream_stop.

Parameters

- `sink` – Pointer to the sink object to delete.

Returns
0 in case of success or negative value in case of error.

struct `bt_bap_broadcast_sink_cb`

#include <bap.h> Broadcast Audio Sink callback structure.

Public Members

void (*base_recv)(struct bt_bap_broadcast_sink *sink, const struct bt_bap_base *base, size_t base_size)

Broadcast Audio Source Endpoint (BASE) received.

Callback for when we receive a BASE from a broadcaster after syncing to the broadcaster's periodic advertising.

Param sink
Pointer to the sink structure.

Param base
Broadcast Audio Source Endpoint (BASE).

Param base_size
Size of the base

void (*syncable)(struct bt_bap_broadcast_sink *sink, const struct bt_iso_biginfo *biginfo)

Broadcast sink is syncable.

Called whenever a broadcast sink is not synchronized to audio, but the audio is synchronizable. This is inferred when a BIGInfo report is received.

Once this callback has been called, it is possible to call *bt_bap_broadcast_sink_sync()* to synchronize to the audio stream(s).

Param sink
Pointer to the sink structure.

Param biginfo
The BIGInfo report.

group bt_bap_broadcast_source

BAP Broadcast Source APIs.

Functions

int bt_bap_broadcast_source_create(struct *bt_bap_broadcast_source_param* *param, struct bt_bap_broadcast_source **source)

Create audio broadcast source.

Create a new audio broadcast source with one or more audio streams.

The broadcast source will be visible for scanners once this has been called, and the device will advertise audio announcements.

No audio data can be sent until *bt_bap_broadcast_source_start()* has been called and no audio information (BIGInfo) will be visible to scanners (see *bt_le_per_adv_sync_cb*).

Parameters

- param – **[in]** Pointer to parameters used to create the broadcast source.
- source – **[out]** Pointer to the broadcast source created

Returns
Zero on success or (negative) error code otherwise.

int bt_bap_broadcast_source_reconfig(struct bt_bap_broadcast_source *source, struct *bt_bap_broadcast_source_param* *param)

Reconfigure audio broadcast source.

Reconfigure an audio broadcast source with a new codec and codec quality of service parameters. This can only be done when the source is stopped.

Since this may modify the Broadcast Audio Source Endpoint (BASE), *bt_bap_broadcast_source_get_base()* should be called after this to get the new BASE information.

If the `param.params_count` is smaller than the number of subgroups that have been created in the Broadcast Source, only the first `param.params_count` subgroups are updated. If a stream exist in a subgroup not part of `param`, then that stream is left as is (i.e. it is not removed; the only way to remove a stream from a Broadcast Source is to recreate the Broadcast Source).

Parameters

- **source** – Pointer to the broadcast source
- **param** – Pointer to parameters used to reconfigure the broadcast source.

Returns
Zero on success or (negative) error code otherwise.

int **bt_bap_broadcast_source_update_metadata**(struct bt_bap_broadcast_source *source, const uint8_t meta[], size_t meta_len)

Modify the metadata of an audio broadcast source.

Modify the metadata an audio broadcast source. This can only be done when the source is started. To update the metadata in the stopped state, use *bt_bap_broadcast_source_reconfig()*.

Parameters

- **source** – Pointer to the broadcast source.
- **meta** – Metadata.
- **meta_len** – Length of metadata.

Returns
Zero on success or (negative) error code otherwise.

int **bt_bap_broadcast_source_start**(struct bt_bap_broadcast_source *source, struct bt_le_ext_adv *adv)

Start audio broadcast source.

Start an audio broadcast source with one or more audio streams. The broadcast source will start advertising BIGInfo, and audio data can be streamed.

Parameters

- **source** – Pointer to the broadcast source
- **adv** – Pointer to an extended advertising set with periodic advertising configured.

Returns
Zero on success or (negative) error code otherwise.

int **bt_bap_broadcast_source_stop**(struct bt_bap_broadcast_source *source)

Stop audio broadcast source.

Stop an audio broadcast source. The broadcast source will stop advertising BIGInfo, and audio data can no longer be streamed.

Parameters

- **source** – Pointer to the broadcast source

Returns
Zero on success or (negative) error code otherwise.

int **bt_bap_broadcast_source_delete**(struct bt_bap_broadcast_source *source)

Delete audio broadcast source.

Delete an audio broadcast source. The broadcast source will stop advertising entirely, and the source can no longer be used.

Parameters

- **source** – Pointer to the broadcast source

Returns
Zero on success or (negative) error code otherwise.

int **bt_bap_broadcast_source_get_id**(struct bt_bap_broadcast_source *source, uint32_t *const broadcast_id)

Get the broadcast ID of a broadcast source.

This will return the 3-octet broadcast ID that should be advertised in the extended advertising data with *BT_UUID_BROADCAST_AUDIO_VAL* as *BT_DATA_SVC_DATA16*.

See table 3.14 in the Basic Audio Profile v1.0.1 for the structure.

Parameters

- **source** – **[in]** Pointer to the broadcast source.
- **broadcast_id** – **[out]** Pointer to the 3-octet broadcast ID.

Returns
Zero on success or (negative) error code otherwise.

int **bt_bap_broadcast_source_get_base**(struct bt_bap_broadcast_source *source, struct *net_buf_simple* *base_buf)

Get the Broadcast Audio Stream Endpoint of a broadcast source.

This will encode the BASE of a broadcast source into a buffer, that can be used for advertisement. The encoded BASE will thus be encoded as little-endian. The BASE shall be put into the periodic advertising data (see *bt_le_per_adv_set_data()*).

See table 3.15 in the Basic Audio Profile v1.0.1 for the structure.

Parameters

- **source** – Pointer to the broadcast source.
- **base_buf** – Pointer to a buffer where the BASE will be inserted.

Returns
Zero on success or (negative) error code otherwise.

struct **bt_bap_broadcast_source_stream_param**

#include <bap.h> Broadcast Source stream parameters.

Public Members

struct *bt_bap_stream* ***stream**

Audio stream.

size_t **data_len**

The number of elements in the `data` array.

The BIS specific data may be omitted and this set to 0.

uint8_t ***data**

BIS Codec Specific Configuration.

struct **bt_bap_broadcast_source_subgroup_param**

#include <*bap.h*> Broadcast Source subgroup parameters.

Public Members

size_t **params_count**

The number of parameters in `stream_params`.

struct *bt_bap_broadcast_source_stream_param* ***params**

Array of stream parameters.

struct *bt_audio_codec_cfg* ***codec_cfg**

Subgroup Codec configuration.

struct **bt_bap_broadcast_source_param**

#include <*bap.h*> Broadcast Source create parameters.

Public Members

size_t **params_count**

The number of parameters in `subgroup_params`.

struct *bt_bap_broadcast_source_subgroup_param* ***params**

Array of stream parameters.

struct *bt_audio_codec_qos* ***qos**

Quality of Service configuration.

uint8_t **packing**

Broadcast Source packing mode.

BT_ISO_PACKING_SEQUENTIAL or BT_ISO_PACKING_INTERLEAVED.

> **Note**
>
> This is a recommendation to the controller, which the controller may ignore.

bool **encryption**

Whether or not to encrypt the streams.

uint8_t **broadcast_code**[*BT_AUDIO_BROADCAST_CODE_SIZE*]

Broadcast code.

If the value is a string or a the value is less than 16 octets, the remaining octets shall be 0.

Example: The string “Broadcast Code” shall be [42 72 6F 61 64 63 61 73 74 20 43 6F 64 65 00 00]

uint8_t **irc**

Immediate Repetition Count.

The number of times the scheduled payloads are transmitted in a given event.

Value range from BT_ISO_IRC_MIN to BT_ISO_IRC_MAX.

uint8_t **pto**

Pre-transmission offset.

Offset used for pre-transmissions.

Value range from BT_ISO_PTO_MIN to BT_ISO_PTO_MAX.

uint16_t **iso_interval**

ISO interval.

Time between consecutive BIS anchor points.

Value range from BT_ISO_ISO_INTERVAL_MIN to BT_ISO_ISO_INTERVAL_MAX.

Common Audio Profile

Related code samples

Bluetooth: Common Audio Profile Acceptor
CAP Acceptor sample that advertises audio availability to CAP Initiators.

Bluetooth: Common Audio Profile Initiator
CAP Initiator sample that connects to CAP Acceptors and setup audio streaming.

API Reference

group **bt_cap**

Common Audio Profile (CAP)

Common Audio Profile (CAP) provides procedures to start, update, and stop unicast and broadcast Audio Streams on individual or groups of devices using procedures in the Basic Audio Profile (BAP). This profile also provides procedures to control volume and device input on groups of devices using procedures in the Volume Control Profile (VCP) and the Microphone Control Profile (MICP). This profile specification also refers to the Common Audio Service (CAS).

Since
3.2

Version
0.8.0

Enums

enum **bt_cap_set_type**

Type of CAP set.

Values:

enumerator **BT_CAP_SET_TYPE_AD_HOC**

The set is an ad-hoc set.

enumerator **BT_CAP_SET_TYPE_CSIP**

The set is a CSIP Coordinated Set.

Functions

int **bt_cap_acceptor_register**(const struct *bt_csip_set_member_register_param* *param,
struct bt_csip_set_member_svc_inst **svc_inst)

Register the Common Audio Service.

This will register and enable the service and make it discoverable by clients. This will also register a Coordinated Set Identification Service instance.

This shall only be done as a server, and requires BT_CAP_ACCEPTOR_SET_MEMBER . If BT_CAP_ACCEPTOR_SET_MEMBER is not enabled, the Common Audio Service will by statically registered.

Parameters

- **param** – **[in]** Coordinated Set Identification Service register parameters.
- **svc_inst** – **[out]** Pointer to the registered Coordinated Set Identification Service.

Returns
0 if success, errno on failure.

int **bt_cap_initiator_unicast_discover**(struct bt_conn *conn)

Discovers audio support on a remote device.

This will discover the Common Audio Service (CAS) on the remote device, to verify if the remote device supports the Common Audio Profile.

Parameters

- **conn** – Connection to a remote server.

Return values

- **0** – Success
- **-EINVAL** – conn is NULL
- **-ENOTCONN** – conn is not connected
- **-ENOMEM** – Could not allocated memory for the request

void **bt_cap_stream_ops_register**(struct *bt_cap_stream* *stream, struct *bt_bap_stream_ops* *ops)

Register Audio operations for a Common Audio Profile stream.

Register Audio operations for a stream.

Parameters

- **stream** – Stream object.
- **ops** – Stream operations structure.

int **bt_cap_stream_send**(struct *bt_cap_stream* *stream, struct *net_buf* *buf, uint16_t seq_num)

Send data to Common Audio Profile stream without timestamp.

See *bt_bap_stream_send()* for more information

Note

Support for sending must be supported, determined by CONFIG_BT_AUDIO_TX .

Parameters

- **stream** – Stream object.
- **buf** – Buffer containing data to be sent.
- **seq_num** – Packet Sequence number. This value shall be incremented for each call to this function and at least once per SDU interval for a specific channel.

Return values

- **-EINVAL** – if stream object is NULL
- **Any** – return value from *bt_bap_stream_send()*

int **bt_cap_stream_send_ts**(struct *bt_cap_stream* *stream, struct *net_buf* *buf, uint16_t seq_num, uint32_t ts)

Send data to Common Audio Profile stream with timestamp.

See *bt_bap_stream_send()* for more information

Note

Support for sending must be supported, determined by CONFIG_BT_AUDIO_TX .

Parameters

- **stream** – Stream object.
- **buf** – Buffer containing data to be sent.
- **seq_num** – Packet Sequence number. This value shall be incremented for each call to this function and at least once per SDU interval for a specific channel.
- **ts** – Timestamp of the SDU in microseconds (us). This value can be used to transmit multiple SDUs in the same SDU interval in a CIG or BIG.

Return values

- **-EINVAL** – if stream object is NULL

- `Any` – return value from *bt_bap_stream_send()*

int **bt_cap_stream_get_tx_sync**(struct *bt_cap_stream* *stream, struct bt_iso_tx_info *info)

Get ISO transmission timing info for a Common Audio Profile stream.

See *bt_bap_stream_get_tx_sync()* for more information

Note

Support for sending must be supported, determined by `CONFIG_BT_AUDIO_TX` .

Parameters

- `stream` – **[in]** Stream object.
- `info` – **[out]** Transmit info object.

Return values

- `-EINVAL` – if stream object is NULL
- `Any` – return value from *bt_bap_stream_get_tx_sync()*

int **bt_cap_initiator_register_cb**(const struct *bt_cap_initiator_cb* *cb)

Register Common Audio Profile Initiator callbacks.

Parameters

- `cb` – The callback structure. Shall remain static.

Returns

0 on success or negative error value on failure.

int **bt_cap_initiator_unregister_cb**(const struct *bt_cap_initiator_cb* *cb)

Unregister Common Audio Profile Initiator callbacks.

Parameters

- `cb` – The callback structure that was previously registered.

Return values

- `0` – Success
- `-EINVAL` – cb is NULL or cb was not registered

int **bt_cap_initiator_unicast_audio_start**(const struct *bt_cap_unicast_audio_start_param* *param)

Setup and start unicast audio streams for a set of devices.

The result of this operation is that the streams in `param` will be initialized and will be usable for streaming audio data. The `unicast_group` value can be used to update and stop the streams.

Note

`CONFIG_BT_CAP_INITIATOR` and `CONFIG_BT_BAP_UNICAST_CLIENT` must be enabled for this function to be enabled.

Parameters

- `param` – Parameters to start the audio streams.

Returns
0 on success or negative error value on failure.

int **bt_cap_initiator_unicast_audio_update**(const struct *bt_cap_unicast_audio_update_param* *param)

Update unicast audio streams.

This will update the metadata of one or more streams.

> **Note**
> CONFIG_BT_CAP_INITIATOR and CONFIG_BT_BAP_UNICAST_CLIENT must be enabled for this function to be enabled.

Parameters
- **param** – Update parameters.

Returns
0 on success or negative error value on failure.

int **bt_cap_initiator_unicast_audio_stop**(const struct *bt_cap_unicast_audio_stop_param* *param)

Stop unicast audio streams.

This will stop one or more streams.

> **Note**
> CONFIG_BT_CAP_INITIATOR and CONFIG_BT_BAP_UNICAST_CLIENT must be enabled for this function to be enabled.

Parameters
- **param** – Stop parameters.

Returns
0 on success or negative error value on failure.

int **bt_cap_initiator_unicast_audio_cancel**(void)

Cancel any current Common Audio Profile procedure.

This will stop the current procedure from continuing and making it possible to run a new Common Audio Profile procedure.

It is recommended to do this if any existing procedure takes longer time than expected, which could indicate a missing response from the Common Audio Profile Acceptor.

This does not send any requests to any Common Audio Profile Acceptors involved with the current procedure, and thus notifications from the Common Audio Profile Acceptors may arrive after this has been called. It is thus recommended to either only use this if a procedure has stalled, or wait a short while before starting any new Common Audio Profile procedure after this has been called to avoid getting notifications from the cancelled procedure. The wait time depends on the connection interval, the number of devices in the previous procedure and the behavior of the Common Audio Profile Acceptors.

The respective callbacks of the procedure will be called as part of this with the connection pointer set to 0 and the err value set to -ECANCELED.

Return values

- `0` – on success
- `-EALREADY` – if no procedure is active

int **bt_cap_initiator_broadcast_audio_create**(const struct *bt_cap_initiator_broadcast_create_param* *param, struct bt_cap_broadcast_source **broadcast_source)

Create a Common Audio Profile broadcast source.

Create a new audio broadcast source with one or more audio streams.

> **Note**
>
> CONFIG_BT_CAP_INITIATOR and CONFIG_BT_BAP_BROADCAST_SOURCE must be enabled for this function to be enabled.

Parameters

- `param` – **[in]** Parameters to start the audio streams.
- `broadcast_source` – **[out]** Pointer to the broadcast source created.

Returns

0 on success or negative error value on failure.

int **bt_cap_initiator_broadcast_audio_start**(struct bt_cap_broadcast_source *broadcast_source, struct bt_le_ext_adv *adv)

Start Common Audio Profile broadcast source.

The broadcast source will be visible for scanners once this has been called, and the device will advertise audio announcements.

This will allow the streams in the broadcast source to send audio by calling *bt_bap_stream_send()*.

> **Note**
>
> CONFIG_BT_CAP_INITIATOR and CONFIG_BT_BAP_BROADCAST_SOURCE must be enabled for this function to be enabled.

Parameters

- `broadcast_source` – Pointer to the broadcast source.
- `adv` – Pointer to an extended advertising set with periodic advertising configured.

Returns

0 on success or negative error value on failure.

int **bt_cap_initiator_broadcast_audio_update**(struct bt_cap_broadcast_source *broadcast_source, const uint8_t meta[], size_t meta_len)

Update broadcast audio streams for a Common Audio Profile broadcast source.

> **Note**
>
> CONFIG_BT_CAP_INITIATOR and CONFIG_BT_BAP_BROADCAST_SOURCE must be enabled for this function to be enabled.

Parameters

- **broadcast_source** – The broadcast source to update.
- **meta** – The new metadata. The metadata shall contain a list of CCIDs as well as a non-0 context bitfield.
- **meta_len** – The length of meta.

Returns

0 on success or negative error value on failure.

int **bt_cap_initiator_broadcast_audio_stop**(struct bt_cap_broadcast_source *broadcast_source)

Stop broadcast audio streams for a Common Audio Profile broadcast source.

> **Note**
>
> CONFIG_BT_CAP_INITIATOR and CONFIG_BT_BAP_BROADCAST_SOURCE must be enabled for this function to be enabled.

Parameters

- **broadcast_source** – The broadcast source to stop. The audio streams in this will be stopped and reset.

Returns

0 on success or negative error value on failure.

int **bt_cap_initiator_broadcast_audio_delete**(struct bt_cap_broadcast_source *broadcast_source)

Delete Common Audio Profile broadcast source.

This can only be done after the broadcast source has been stopped by calling *bt_cap_initiator_broadcast_audio_stop()* and after the *bt_bap_stream_ops.stopped()* callback has been called for all streams in the broadcast source.

> **Note**
>
> CONFIG_BT_CAP_INITIATOR and CONFIG_BT_BAP_BROADCAST_SOURCE must be enabled for this function to be enabled.

Parameters

- **broadcast_source** – The broadcast source to delete. The broadcast_source will be invalidated.

Returns

0 on success or negative error value on failure.

int **bt_cap_initiator_broadcast_get_id**(const struct bt_cap_broadcast_source *broadcast_source, uint32_t *const broadcast_id)

Get the broadcast ID of a Common Audio Profile broadcast source.

This will return the 3-octet broadcast ID that should be advertised in the extended advertising data with *BT_UUID_BROADCAST_AUDIO_VAL* as *BT_DATA_SVC_DATA16*.

See table 3.14 in the Basic Audio Profile v1.0.1 for the structure.

Parameters

- `broadcast_source` – **[in]** Pointer to the broadcast source.
- `broadcast_id` – **[out]** Pointer to the 3-octet broadcast ID.

Returns
int 0 if on success, errno on error.

int `bt_cap_initiator_broadcast_get_base`(struct bt_cap_broadcast_source *broadcast_source, struct *net_buf_simple* *base_buf)

Get the Broadcast Audio Stream Endpoint of a Common Audio Profile broadcast source.

This will encode the BASE of a broadcast source into a buffer, that can be used for advertisement. The encoded BASE will thus be encoded as little-endian. The BASE shall be put into the periodic advertising data (see *bt_le_per_adv_set_data()*).

See table 3.15 in the Basic Audio Profile v1.0.1 for the structure.

Parameters

- `broadcast_source` – Pointer to the broadcast source.
- `base_buf` – Pointer to a buffer where the BASE will be inserted.

Returns
int 0 if on success, errno on error.

int `bt_cap_initiator_unicast_to_broadcast`(const struct *bt_cap_unicast_to_broadcast_param* *param, struct bt_cap_broadcast_source **source)

Hands over the data streams in a unicast group to a broadcast source.

The streams in the unicast group will be stopped and the unicast group will be deleted. This can only be done for source streams.

Note

`CONFIG_BT_CAP_INITIATOR`, `CONFIG_BT_BAP_UNICAST_CLIENT` and `CONFIG_BT_BAP_BROADCAST_SOURCE` must be enabled for this function to be enabled.

Parameters

- `param` – The parameters for the handover.
- `source` – The resulting broadcast source.

Returns
0 on success or negative error value on failure.

int `bt_cap_initiator_broadcast_to_unicast`(const struct *bt_cap_broadcast_to_unicast_param* *param, struct bt_bap_unicast_group **unicast_group)

Hands over the data streams in a broadcast source to a unicast group.

The streams in the broadcast source will be stopped and the broadcast source will be deleted.

> **Note**
>
> CONFIG_BT_CAP_INITIATOR , CONFIG_BT_BAP_UNICAST_CLIENT and CONFIG_BT_BAP_BROADCAST_SOURCE must be enabled for this function to be enabled.

Parameters

- **param** – **[in]** The parameters for the handover.
- **unicast_group** – **[out]** The resulting broadcast source.

Returns

0 on success or negative error value on failure.

int **bt_cap_commander_register_cb**(const struct *bt_cap_commander_cb* *cb)

Register Common Audio Profile Commander callbacks.

Parameters

- **cb** – The callback structure. Shall remain static.

Return values

- **0** – Success
- **-EINVAL** – cb is NULL
- **-EALREADY** – Callbacks are already registered

int **bt_cap_commander_unregister_cb**(const struct *bt_cap_commander_cb* *cb)

Unregister Common Audio Profile Commander callbacks.

Parameters

- **cb** – The callback structure that was previously registered.

Return values

- **0** – Success
- **-EINVAL** – cb is NULL or cb was not registered

int **bt_cap_commander_discover**(struct bt_conn *conn)

Discovers audio support on a remote device.

This will discover the Common Audio Service (CAS) on the remote device, to verify if the remote device supports the Common Audio Profile.

> **Note**
>
> CONFIG_BT_CAP_COMMANDER must be enabled for this function. If CONFIG_BT_CAP_INITIATOR is also enabled, it does not matter if *bt_cap_commander_discover()* or *bt_cap_initiator_unicast_discover()* is used.

Parameters

- **conn** – Connection to a remote server.

Return values

- `0` – Success
- `-EINVAL` – conn is NULL
- `-ENOTCONN` – conn is not connected
- `-ENOMEM` – Could not allocated memory for the request
- `-EBUSY` – Already doing discovery for conn

int **bt_cap_commander_cancel**(void)

Cancel any current Common Audio Profile commander procedure.

This will stop the current procedure from continuing and making it possible to run a new Common Audio Profile procedure.

It is recommended to do this if any existing procedure takes longer time than expected, which could indicate a missing response from the Common Audio Profile Acceptor.

This does not send any requests to any Common Audio Profile Acceptors involved with the current procedure, and thus notifications from the Common Audio Profile Acceptors may arrive after this has been called. It is thus recommended to either only use this if a procedure has stalled, or wait a short while before starting any new Common Audio Profile procedure after this has been called to avoid getting notifications from the cancelled procedure. The wait time depends on the connection interval, the number of devices in the previous procedure and the behavior of the Common Audio Profile Acceptors.

The respective callbacks of the procedure will be called as part of this with the connection pointer set to NULL and the err value set to -ECANCELED.

Return values

- `0` – on success
- `-EALREADY` – if no procedure is active

int **bt_cap_commander_broadcast_reception_start**(const struct *bt_cap_commander_broadcast_reception_start_param* *param)

Starts the reception of broadcast audio on one or more remote Common Audio Profile Acceptors.

Parameters

- `param` – The parameters to start the broadcast audio

Returns

0 on success or negative error value on failure.

int **bt_cap_commander_broadcast_reception_stop**(const struct *bt_cap_commander_broadcast_reception_stop_param* *param)

Stops the reception of broadcast audio on one or more remote Common Audio Profile Acceptors.

Parameters

- `param` – The parameters to stop the broadcast audio

Returns

0 on success or negative error value on failure.

int **bt_cap_commander_change_volume**(const struct *bt_cap_commander_change_volume_param* *param)

Change the volume on one or more Common Audio Profile Acceptors.

Parameters

- `param` – The parameters for the volume change

Returns
0 on success or negative error value on failure.

int `bt_cap_commander_change_volume_offset`(const struct *bt_cap_commander_change_volume_offset_param* *param)

Change the volume offset on one or more Common Audio Profile Acceptors.

Parameters

- `param` – The parameters for the volume offset change

Returns
0 on success or negative error value on failure.

int `bt_cap_commander_change_volume_mute_state`(const struct *bt_cap_commander_change_volume_mute_state_param* *param)

Change the volume mute state on one or more Common Audio Profile Acceptors.

Parameters

- `param` – The parameters for the volume mute state change

Returns
0 on success or negative error value on failure.

int `bt_cap_commander_change_microphone_mute_state`(const struct *bt_cap_commander_change_microphone_mute_sta* *param)

Change the microphone mute state on one or more Common Audio Profile Acceptors.

Parameters

- `param` – The parameters for the microphone mute state change

Returns
0 on success or negative error value on failure.

int `bt_cap_commander_change_microphone_gain_setting`(const struct *bt_cap_commander_change_microphone_gain_s* *param)

Change the microphone gain setting on one or more Common Audio Profile Acceptors.

Parameters

- `param` – The parameters for the microphone gain setting change

Returns
0 on success or negative error value on failure.

struct `bt_cap_initiator_cb`

#include <cap.h> Callback structure for CAP procedures.

Public Members

void (*`unicast_discovery_complete`)(struct bt_conn *conn, int err, const struct *bt_csip_set_coordinator_set_member* *member, const struct *bt_csip_set_coordinator_csis_inst* *csis_inst)

Callback for *bt_cap_initiator_unicast_discover()*.

Param conn
The connection pointer supplied to *bt_cap_initiator_unicast_discover()*.

Param err
0 if Common Audio Service was found else -ENODATA.

Param member
Pointer to the set member. NULL if err != 0.

Param csis_inst
The Coordinated Set Identification Service if Common Audio Service was found and includes a Coordinated Set Identification Service. NULL on error or if remote device does not include Coordinated Set Identification Service. NULL if err != 0.

void (*`unicast_start_complete`)(int err, struct bt_conn *conn)

Callback for *bt_cap_initiator_unicast_audio_start()*.

Param err
0 if success, *BT_GATT_ERR()* with a specific ATT (BT_ATT_ERR_*) error code or -ECANCELED if cancelled by *bt_cap_initiator_unicast_audio_cancel()*.

Param conn
Pointer to the connection where the error occurred. NULL if `err` is 0 or if cancelled by *bt_cap_initiator_unicast_audio_cancel()*

void (*`unicast_update_complete`)(int err, struct bt_conn *conn)

Callback for *bt_cap_initiator_unicast_audio_update()*.

Param err
0 if success, *BT_GATT_ERR()* with a specific ATT (BT_ATT_ERR_*) error code or -ECANCELED if cancelled by *bt_cap_initiator_unicast_audio_cancel()*.

Param conn
Pointer to the connection where the error occurred. NULL if `err` is 0 or if cancelled by *bt_cap_initiator_unicast_audio_cancel()*

void (*`unicast_stop_complete`)(int err, struct bt_conn *conn)

Callback for *bt_cap_initiator_unicast_audio_stop()*.

Param err
0 if success, *BT_GATT_ERR()* with a specific ATT (BT_ATT_ERR_*) error code or -ECANCELED if cancelled by *bt_cap_initiator_unicast_audio_cancel()*.

Param conn
Pointer to the connection where the error occurred. NULL if `err` is 0 or if cancelled by *bt_cap_initiator_unicast_audio_cancel()*

union `bt_cap_set_member`

#include <cap.h> Represents a Common Audio Set member that are either in a Coordinated or ad-hoc set.

Public Members

struct bt_conn *`member`

Connection pointer if the type is BT_CAP_SET_TYPE_AD_HOC.

struct *bt_csip_set_coordinator_csis_inst* ***csip**

CSIP Coordinated Set struct used if type is BT_CAP_SET_TYPE_CSIP.

struct **bt_cap_stream**

#include <*cap.h*> Common Audio Profile stream structure.

Streams represents a Basic Audio Profile (BAP) stream and operation callbacks. See *bt_bap_stream* for additional information.

Public Members

struct *bt_bap_stream* **bap_stream**

The underlying BAP audio stream.

struct *bt_bap_stream_ops* ***ops**

Audio stream operations.

struct **bt_cap_unicast_audio_start_stream_param**

#include <*cap.h*> Stream specific parameters for the *bt_cap_initiator_unicast_audio_start()* function.

Public Members

union *bt_cap_set_member* **member**

Coordinated or ad-hoc set member.

struct *bt_cap_stream* ***stream**

Stream for the `member`.

struct bt_bap_ep ***ep**

Endpoint reference for the `stream`.

struct *bt_audio_codec_cfg* ***codec_cfg**

Codec configuration.

The `codec_cfg.meta` shall include a list of CCIDs (*BT_AUDIO_METADATA_TYPE_CCID_LIST*) as well as a non-0 stream context (*BT_AUDIO_METADATA_TYPE_STREAM_CONTEXT*) bitfield.

This value is assigned to the `stream`, and shall remain valid while the stream is non-idle.

struct **bt_cap_unicast_audio_start_param**

#include <*cap.h*> Parameters for the *bt_cap_initiator_unicast_audio_start()* function.

Public Members

enum *bt_cap_set_type* **type**

The type of the set.

size_t **count**

The number of parameters in `stream_params`.

struct *bt_cap_unicast_audio_start_stream_param* ***stream_params**

Array of stream parameters.

struct **bt_cap_unicast_audio_update_stream_param**

#include <*cap.h*> Stream specific parameters for the *bt_cap_initiator_unicast_audio_update()* function.

Public Members

struct *bt_cap_stream* ***stream**

Stream to update.

size_t **meta_len**

The length of `meta`.

uint8_t ***meta**

The new metadata.

The metadata shall contain a list of CCIDs as well as a non-0 context bitfield.

struct **bt_cap_unicast_audio_update_param**

#include <*cap.h*> Parameters for the *bt_cap_initiator_unicast_audio_update()* function.

Public Members

enum *bt_cap_set_type* **type**

The type of the set.

size_t **count**

The number of parameters in `stream_params`.

struct *bt_cap_unicast_audio_update_stream_param* ***stream_params**

Array of stream parameters.

struct **bt_cap_unicast_audio_stop_param**

#include <*cap.h*> Parameters for the *bt_cap_initiator_unicast_audio_stop()* function.

Public Members

enum *bt_cap_set_type* **type**

The type of the set.

size_t **count**

The number of streams in `streams`.

struct *bt_cap_stream* ****streams**

Array of streams to stop.

struct **bt_cap_initiator_broadcast_stream_param**

#include <*cap.h*> Parameters part of *bt_cap_initiator_broadcast_subgroup_param* for *bt_cap_initiator_broadcast_audio_create()*

Public Members

struct *bt_cap_stream* ***stream**

Audio stream.

size_t **data_len**

The length of the p data array.

The BIS specific data may be omitted and this set to 0.

uint8_t ***data**

BIS Codec Specific Configuration.

struct **bt_cap_initiator_broadcast_subgroup_param**

#include <*cap.h*> Parameters part of *bt_cap_initiator_broadcast_create_param* for *bt_cap_initiator_broadcast_audio_create()*

Public Members

size_t **stream_count**

The number of parameters in `stream_params`.

struct *bt_cap_initiator_broadcast_stream_param* ***stream_params**

Array of stream parameters.

struct *bt_audio_codec_cfg* ***codec_cfg**

Subgroup Codec configuration.

struct **bt_cap_initiator_broadcast_create_param**

#include <*cap.h*> Parameters for * *bt_cap_initiator_broadcast_audio_create()*

Public Members

size_t **subgroup_count**

The number of parameters in `subgroup_params`.

struct *bt_cap_initiator_broadcast_subgroup_param* ***subgroup_params**

Array of stream parameters.

struct *bt_audio_codec_qos* ***qos**

Quality of Service configuration.

uint8_t **packing**

Broadcast Source packing mode.

BT_ISO_PACKING_SEQUENTIAL or BT_ISO_PACKING_INTERLEAVED.

> **Note**
>
> This is a recommendation to the controller, which the controller may ignore.

bool **encryption**

Whether or not to encrypt the streams.

uint8_t **broadcast_code**[*BT_AUDIO_BROADCAST_CODE_SIZE*]

16-octet broadcast code.

Only valid if `encrypt` is true.

If the value is a string or a the value is less than 16 octets, the remaining octets shall be 0.

Example: The string “Broadcast Code” shall be [42 72 6F 61 64 63 61 73 74 20 43 6F 64 65 00 00]

uint8_t **irc**

Immediate Repetition Count.

The number of times the scheduled payloads are transmitted in a given event.

Value range from BT_ISO_IRC_MIN to BT_ISO_IRC_MAX.

uint8_t **pto**

Pre-transmission offset.

Offset used for pre-transmissions.

Value range from BT_ISO_PTO_MIN to BT_ISO_PTO_MAX.

uint16_t **iso_interval**

ISO interval.

Time between consecutive BIS anchor points.

Value range from BT_ISO_ISO_INTERVAL_MIN to BT_ISO_ISO_INTERVAL_MAX.

struct **bt_cap_unicast_to_broadcast_param**

#include <*cap.h*> Parameters for *bt_cap_initiator_unicast_to_broadcast()*

Public Members

struct bt_bap_unicast_group *`unicast_group`

The source unicast group with the streams.

bool `encrypt`

Whether or not to encrypt the streams.

If set to true, then the broadcast code in `broadcast_code` will be used to encrypt the streams.

uint8_t `broadcast_code`[BT_ISO_BROADCAST_CODE_SIZE]

16-octet broadcast code.

Only valid if `encrypt` is true.

If the value is a string or a the value is less than 16 octets, the remaining octets shall be 0.

Example: The string "Broadcast Code" shall be [42 72 6F 61 64 63 61 73 74 20 43 6F 64 65 00 00]

struct `bt_cap_broadcast_to_unicast_param`

#include <*cap.h*> Parameters for *bt_cap_initiator_broadcast_to_unicast()*

Public Members

struct bt_cap_broadcast_source *`broadcast_source`

The source broadcast source with the streams.

The broadcast source will be stopped and deleted.

enum *bt_cap_set_type* `type`

The type of the set.

size_t `count`

The number of set members in `members`.

This value shall match the number of streams in the `broadcast_source`.

union *bt_cap_set_member* **`members`

Coordinated or ad-hoc set members.

struct `bt_cap_commander_cb`

#include <*cap.h*> Callback structure for CAP procedures.

Public Members

void (*`discovery_complete`)(struct bt_conn *conn, int err, const struct *bt_csip_set_coordinator_set_member* *member, const struct *bt_csip_set_coordinator_csis_inst* *csis_inst)

Callback for *bt_cap_initiator_unicast_discover()*.

Param conn
The connection pointer supplied to *bt_cap_initiator_unicast_discover()*.
Param err
0 if Common Audio Service was found else -ENODATA.
Param member
Pointer to the set member. NULL if err != 0.
Param csis_inst
The Coordinated Set Identification Service if Common Audio Service was found and includes a Coordinated Set Identification Service. NULL on error or if remote device does not include Coordinated Set Identification Service. NULL if err != 0.

void (*`volume_changed`)(struct bt_conn *conn, int err)

Callback for *bt_cap_commander_change_volume()*.
Param conn
Pointer to the connection where the error occurred. NULL if `err` is 0 or if cancelled by *bt_cap_commander_cancel()*
Param err
0 on success, *BT_GATT_ERR()* with a specific ATT (BT_ATT_ERR_*) error code or -ECANCELED if cancelled by *bt_cap_commander_cancel()*.

void (*`volume_mute_changed`)(struct bt_conn *conn, int err)

Callback for *bt_cap_commander_change_volume_mute_state()*.
Param conn
Pointer to the connection where the error occurred. NULL if `err` is 0 or if cancelled by *bt_cap_commander_cancel()*
Param err
0 on success, *BT_GATT_ERR()* with a specific ATT (BT_ATT_ERR_*) error code or -ECANCELED if cancelled by *bt_cap_commander_cancel()*.

void (*`volume_offset_changed`)(struct bt_conn *conn, int err)

Callback for *bt_cap_commander_change_volume_offset()*.
Param conn
Pointer to the connection where the error occurred. NULL if `err` is 0 or if cancelled by *bt_cap_commander_cancel()*
Param err
0 on success, *BT_GATT_ERR()* with a specific ATT (BT_ATT_ERR_*) error code or -ECANCELED if cancelled by *bt_cap_commander_cancel()*.

void (*`microphone_mute_changed`)(struct bt_conn *conn, int err)

Callback for *bt_cap_commander_change_microphone_mute_state()*.
Param conn
Pointer to the connection where the error occurred. NULL if `err` is 0 or if cancelled by *bt_cap_commander_cancel()*
Param err
0 on success, *BT_GATT_ERR()* with a specific ATT (BT_ATT_ERR_*) error code or -ECANCELED if cancelled by *bt_cap_commander_cancel()*.

void (*`microphone_gain_changed`)(struct bt_conn *conn, int err)

Callback for *bt_cap_commander_change_microphone_gain_setting()*.
Param conn
Pointer to the connection where the error occurred. NULL if `err` is 0 or if cancelled by *bt_cap_commander_cancel()*
Param err
0 on success, *BT_GATT_ERR()* with a specific ATT (BT_ATT_ERR_*) error code or -ECANCELED if cancelled by *bt_cap_commander_cancel()*.

void (*broadcast_reception_start)(struct bt_conn *conn, int err)

Callback for *bt_cap_commander_broadcast_reception_start()*.

Param conn
Pointer to the connection where the error occurred. NULL if `err` is 0 or if cancelled by *bt_cap_commander_cancel()*

Param err
0 on success, *BT_GATT_ERR()* with a specific ATT (BT_ATT_ERR_*) error code or -ECANCELED if cancelled by *bt_cap_commander_cancel()*.

struct bt_cap_commander_broadcast_reception_start_member_param

#include <*cap.h*> Parameters part of *bt_cap_commander_broadcast_reception_start_param* for *bt_cap_commander_broadcast_reception_start()*

Public Members

union *bt_cap_set_member* member

Coordinated or ad-hoc set member.

bt_addr_le_t addr

Address of the advertiser.

uint8_t adv_sid

SID of the advertising set.

uint16_t pa_interval

Periodic advertising interval in milliseconds.

BT_BAP_PA_INTERVAL_UNKNOWN if unknown.

uint32_t broadcast_id

24-bit broadcast ID

struct *bt_bap_bass_subgroup* subgroups[CONFIG_BT_BAP_BASS_MAX_SUBGROUPS]

Pointer to array of subgroups.

At least one bit in one of the subgroups bis_sync parameters shall be set.

size_t num_subgroups

Number of subgroups.

struct bt_cap_commander_broadcast_reception_start_param

#include <*cap.h*> Parameters for starting broadcast reception

Public Members

enum *bt_cap_set_type* type

The type of the set.

struct *bt_cap_commander_broadcast_reception_start_member_param* ***param**

The set of devices for this procedure.

size_t **count**

The number of parameters in `param`.

struct **bt_cap_commander_broadcast_reception_stop_param**

#include <*cap.h*> Parameters for stopping broadcast reception

Public Members

enum *bt_cap_set_type* **type**

The type of the set.

union *bt_cap_set_member* ***members**

Coordinated or ad-hoc set member.

size_t **count**

The number of members in `members`.

struct **bt_cap_commander_change_volume_param**

#include <*cap.h*> Parameters for changing absolute volume

Public Members

enum *bt_cap_set_type* **type**

The type of the set.

union *bt_cap_set_member* ***members**

Coordinated or ad-hoc set member.

size_t **count**

The number of members in `members`.

uint8_t **volume**

The absolute volume to set.

struct **bt_cap_commander_change_volume_offset_member_param**

#include <*cap.h*> Parameters part of *bt_cap_commander_change_volume_offset_param* for *bt_cap_commander_change_volume_offset()*

Public Members

union *bt_cap_set_member* **member**

Coordinated or ad-hoc set member.

int16_t **offset**

The offset to set.

Value shall be between BT_VOCS_MIN_OFFSET and BT_VOCS_MAX_OFFSET

struct **bt_cap_commander_change_volume_offset_param**

#include <*cap.h*> Parameters for changing volume offset.

Public Members

enum *bt_cap_set_type* **type**

The type of the set.

struct *bt_cap_commander_change_volume_offset_member_param* ***param**

The set of devices for this procedure.

size_t **count**

The number of parameters in `param`.

struct **bt_cap_commander_change_volume_mute_state_param**

#include <*cap.h*> Parameters for changing volume mute state.

Public Members

enum *bt_cap_set_type* **type**

The type of the set.

union *bt_cap_set_member* ***members**

Coordinated or ad-hoc set member.

size_t **count**

The number of members in `members`.

bool **mute**

The volume mute state to set.

true to mute, and false to unmute

struct **bt_cap_commander_change_microphone_mute_state_param**

#include <*cap.h*> Parameters for changing microphone mute state.

Public Members

enum *bt_cap_set_type* **type**

The type of the set.

union *bt_cap_set_member* ***members**
Coordinated or ad-hoc set member.

size_t **count**
The number of members in `members`.

bool **mute**
The microphone mute state to set.

true to mute, and false to unmute

struct **bt_cap_commander_change_microphone_gain_setting_member_param**
#include <*cap.h*> Parameters part of *bt_cap_commander_change_microphone_gain_setting_param* for *bt_cap_commander_change_microphone_gain_setting()*

Public Members

union *bt_cap_set_member* **member**
Coordinated or ad-hoc set member.

int8_t **gain**
The microphone gain setting to set.

struct **bt_cap_commander_change_microphone_gain_setting_param**
#include <*cap.h*> Parameters for changing microphone mute state.

Public Members

enum *bt_cap_set_type* **type**
The type of the set.

struct *bt_cap_commander_change_microphone_gain_setting_member_param* ***param**
The set of devices for this procedure.

size_t **count**
The number of parameters in `param`.

Bluetooth Coordinated Sets

API Reference

group **bt_gatt_csip**
Coordinated Set Identification Profile (CSIP)

Published Audio Capabilities Service (PACS)

The Coordinated Set Identification Profile (CSIP) provides procedures to discover and coordinate sets of devices.

Since
3.0

Version
0.8.0

The Published Audio Capabilities Service (PACS) is used to expose capabilities to remote devices.

Since
3.0

Version
0.8.0

Defines

`BT_CSIP_SET_COORDINATOR_DISCOVER_TIMER_VALUE`
Recommended timer for member discovery.

`BT_CSIP_SET_COORDINATOR_MAX_CSIS_INSTANCES`
Defines the maximum number of Coordinated Set Identification service instances for the Coordinated Set Identification Set Coordinator.

`BT_CSIP_READ_SIRK_REQ_RSP_ACCEPT`
Accept the request to read the SIRK as plaintext.

`BT_CSIP_READ_SIRK_REQ_RSP_ACCEPT_ENC`
Accept the request to read the SIRK, but return encrypted SIRK.

`BT_CSIP_READ_SIRK_REQ_RSP_REJECT`
Reject the request to read the SIRK.

`BT_CSIP_READ_SIRK_REQ_RSP_OOB_ONLY`
SIRK is available only via an OOB procedure.

`BT_CSIP_SIRK_SIZE`
Size of the Set Identification Resolving Key (SIRK)

`BT_CSIP_RSI_SIZE`
Size of the Resolvable Set Identifier (RSI)

`BT_CSIP_ERROR_LOCK_DENIED`
Service is already locked.

`BT_CSIP_ERROR_LOCK_RELEASE_DENIED`
Service is not locked.

`BT_CSIP_ERROR_LOCK_INVAL_VALUE`
Invalid lock value.

BT_CSIP_ERROR_SIRK_OOB_ONLY
: SIRK only available out-of-band.

BT_CSIP_ERROR_LOCK_ALREADY_GRANTED
: Client is already owner of the lock.

BT_CSIP_DATA_RSI(_rsi)
: Helper to declare *bt_data* array including RSI.

  This macro is mainly for creating an array of struct *bt_data* elements which is then passed to e.g. *bt_le_ext_adv_start()*.

  Parameters

  - **_rsi** – Pointer to the RSI value

Typedefs

typedef void (***bt_csip_set_coordinator_discover_cb**)(struct bt_conn *conn, const struct *bt_csip_set_coordinator_set_member* *member, int err, size_t set_count)

Callback for discovering Coordinated Set Identification Services.

Param conn
: Pointer to the remote device.

Param member
: Pointer to the set member.

Param err
: 0 on success, or an errno value on error.

Param set_count
: Number of sets on the member.

typedef void (***bt_csip_set_coordinator_lock_set_cb**)(int err)

Callback for locking a set across one or more devices.

Param err
: 0 on success, or an errno value on error.

typedef void (***bt_csip_set_coordinator_lock_changed_cb**)(struct *bt_csip_set_coordinator_csis_inst* *inst, bool locked)

Callback when the lock value on a set of a connected device changes.

Param inst
: The Coordinated Set Identification Service instance that was changed.

Param locked
: Whether the lock is locked or release.

Return
: int Return 0 on success, or an errno value on error.

typedef void (***bt_csip_set_coordinator_sirk_changed_cb**)(struct *bt_csip_set_coordinator_csis_inst* *inst)

Callback when the SIRK value of a set of a connected device changes.

Param inst
: The Coordinated Set Identification Service instance that was changed. The new SIRK can be accessed via the `inst.info`.

typedef void (*bt_csip_set_coordinator_ordered_access_cb_t)(const struct *bt_csip_set_coordinator_set_info* *set_info, int err, bool locked, struct *bt_csip_set_coordinator_set_member* *member)

Callback for *bt_csip_set_coordinator_ordered_access()*

If any of the set members supplied to *bt_csip_set_coordinator_ordered_access()* is in the locked state, this will be called with `locked` true and `member` will be the locked member, and the ordered access procedure is cancelled. Likewise, if any error occurs, the procedure will also be aborted.

Param set_info
Pointer to the a specific set_info struct.

Param err
Error value. 0 on success, GATT error or errno on fail.

Param locked
Whether the lock is locked or release.

Param member
The locked member if `locked` is true, otherwise NULL.

typedef bool (*bt_csip_set_coordinator_ordered_access_t)(const struct *bt_csip_set_coordinator_set_info* *set_info, struct *bt_csip_set_coordinator_set_member* *members[], size_t count)

Callback function definition for *bt_csip_set_coordinator_ordered_access()*

Param set_info
Pointer to the a specific set_info struct.

Param members
Array of members ordered by rank. The procedure shall be done on the members in ascending order.

Param count
Number of members in `members`.

Return
true if the procedures can be successfully done, or false to stop the procedure.

typedef bool (*bt_pacs_cap_foreach_func_t)(const struct *bt_pacs_cap* *cap, void *user_data)

Published Audio Capability iterator callback.

Param cap
Capability found.

Param user_data
Data given.

Return
true to continue to the next capability

Return
false to stop the iteration

Functions

void *bt_csip_set_member_svc_decl_get(const struct bt_csip_set_member_svc_inst *svc_inst)

Get the service declaration attribute.

The first service attribute can be included in any other GATT service.

Parameters

- svc_inst – Pointer to the Coordinated Set Identification Service.

Returns

The first CSIS attribute instance.

int bt_csip_set_member_register(const struct *bt_csip_set_member_register_param* *param, struct bt_csip_set_member_svc_inst **svc_inst)

Register a Coordinated Set Identification Service instance.

This will register and enable the service and make it discoverable by clients.

This shall only be done as a server.

Parameters

- param – Coordinated Set Identification Service register parameters.
- svc_inst – **[out]** Pointer to the registered Coordinated Set Identification Service.

Returns

0 if success, errno on failure.

int bt_csip_set_member_unregister(struct bt_csip_set_member_svc_inst *svc_inst)

Unregister a Coordinated Set Identification Service instance.

This will unregister and disable the service instance.

Parameters

- svc_inst – Pointer to the registered Coordinated Set Identification Service.

Returns

0 if success, errno on failure.

int bt_csip_set_member_sirk(struct bt_csip_set_member_svc_inst *svc_inst, const uint8_t sirk[16])

Set the SIRK of a service instance.

Parameters

- svc_inst – Pointer to the registered Coordinated Set Identification Service.
- sirk – The new SIRK.

int bt_csip_set_member_get_sirk(struct bt_csip_set_member_svc_inst *svc_inst, uint8_t sirk[16])

Get the SIRK of a service instance.

Parameters

- svc_inst – **[in]** Pointer to the registered Coordinated Set Identification Service.
- sirk – **[out]** Array to store the SIRK in.

int **bt_csip_set_member_generate_rsi**(const struct bt_csip_set_member_svc_inst *svc_inst, uint8_t rsi[6])

Generate the Resolvable Set Identifier (RSI) value.

This will generate RSI for given `svc_inst` instance.

Parameters

- **svc_inst** – Pointer to the Coordinated Set Identification Service.
- **rsi** – Pointer to the 6-octet newly generated RSI data in little-endian.

Returns
int 0 if on success, errno on error.

int **bt_csip_set_member_lock**(struct bt_csip_set_member_svc_inst *svc_inst, bool lock, bool force)

Locks a specific Coordinated Set Identification Service instance on the server.

Parameters

- **svc_inst** – Pointer to the Coordinated Set Identification Service.
- **lock** – If true lock the set, if false release the set.
- **force** – This argument only have meaning when `lock` is false (release) and will force release the lock, regardless of who took the lock.

Returns
0 on success, GATT error on error.

int **bt_csip_set_coordinator_discover**(struct bt_conn *conn)

Initialise the csip_set_coordinator instance for a connection.

This will do a discovery on the device and prepare the instance for following commands.

Parameters

- **conn** – Pointer to remote device to perform discovery on.

Returns
int Return 0 on success, or an errno value on error.

struct *bt_csip_set_coordinator_set_member* ***bt_csip_set_coordinator_set_member_by_conn**(const struct bt_conn *conn)

Get the set member from a connection pointer.

Get the Coordinated Set Identification Profile Set Coordinator pointer from a connection pointer. Only Set Coordinators that have been initiated via *bt_csip_set_coordinator_discover()* can be retrieved.

Parameters

- **conn** – Connection pointer.

Return values

- **Pointer** – to a Coordinated Set Identification Profile Set Coordinator instance
- **NULL** – if `conn` is NULL or if the connection has not done discovery yet

bool **bt_csip_set_coordinator_is_set_member**(const uint8_t sirk[16], struct *bt_data* *data)

Check if advertising data indicates a set member.

Parameters

- **sirk** – The SIRK of the set to check against
- **data** – The advertising data

Returns
true if the advertising data indicates a set member, false otherwise

int **bt_csip_set_coordinator_register_cb**(struct *bt_csip_set_coordinator_cb* *cb)

Registers callbacks for csip_set_coordinator.

Parameters

- **cb** – Pointer to the callback structure.

Returns
Return 0 on success, or an errno value on error.

int **bt_csip_set_coordinator_ordered_access**(const struct *bt_csip_set_coordinator_set_member* *members[], uint8_t count, const struct *bt_csip_set_coordinator_set_info* *set_info, *bt_csip_set_coordinator_ordered_access_t* cb)

Access Coordinated Set devices in an ordered manner as a client.

This function will read the lock state of all devices and if all devices are in the unlocked state, then cb will be called with the same members as provided by members, but where the members are ordered by rank (if present). Once this procedure is finished or an error occurs, *bt_csip_set_coordinator_cb::ordered_access* will be called.

This procedure only works if all the members have the lock characteristic, and all either has rank = 0 or unique ranks.

If any of the members are in the locked state, the procedure will be cancelled.

This can only be done on members that are bonded.

Parameters

- **members** – Array of set members to access.
- **count** – Number of set members in members.
- **set_info** – Pointer to the a specific set_info struct, as a member may be part of multiple sets.
- **cb** – The callback function to be called for each member.

int **bt_csip_set_coordinator_lock**(const struct *bt_csip_set_coordinator_set_member* **members, uint8_t count, const struct *bt_csip_set_coordinator_set_info* *set_info)

Lock an array of set members.

The members will be locked starting from lowest rank going up.

TODO: If locking fails, the already locked members will not be unlocked.

Parameters

- **members** – Array of set members to lock.
- **count** – Number of set members in members.
- **set_info** – Pointer to the a specific set_info struct, as a member may be part of multiple sets.

Returns
Return 0 on success, or an errno value on error.

int **bt_csip_set_coordinator_release**(const struct *bt_csip_set_coordinator_set_member* **members, uint8_t count, const struct *bt_csip_set_coordinator_set_info* *set_info)

Release an array of set members.

The members will be released starting from highest rank going down.

Parameters

- **members** – Array of set members to lock.
- **count** – Number of set members in `members`.
- **set_info** – Pointer to the a specific set_info struct, as a member may be part of multiple sets.

Returns

Return 0 on success, or an errno value on error.

void **bt_pacs_cap_foreach**(enum *bt_audio_dir* dir, *bt_pacs_cap_foreach_func_t* func, void *user_data)

Published Audio Capability iterator.

Iterate capabilities with endpoint direction specified.

Parameters

- **dir** – Direction of the endpoint to look capability for.
- **func** – Callback function.
- **user_data** – Data to pass to the callback.

int **bt_pacs_cap_register**(enum *bt_audio_dir* dir, struct *bt_pacs_cap* *cap)

Register Published Audio Capability.

Register Audio Local Capability.

Parameters

- **dir** – Direction of the endpoint to register capability for.
- **cap** – Capability structure.

Returns

0 in case of success or negative value in case of error.

int **bt_pacs_cap_unregister**(enum *bt_audio_dir* dir, struct *bt_pacs_cap* *cap)

Unregister Published Audio Capability.

Unregister Audio Local Capability.

Parameters

- **dir** – Direction of the endpoint to unregister capability for.
- **cap** – Capability structure.

Returns

0 in case of success or negative value in case of error.

int **bt_pacs_set_location**(enum *bt_audio_dir* dir, enum *bt_audio_location* location)

Set the location for an endpoint type.

Parameters

- **dir** – Direction of the endpoints to change location for.
- **location** – The location to be set.

Returns
0 in case of success or negative value in case of error.

int **bt_pacs_set_available_contexts**(enum *bt_audio_dir* dir, enum *bt_audio_context* contexts)

Set the available contexts for an endpoint type.

Parameters

- **dir** – Direction of the endpoints to change available contexts for.
- **contexts** – The contexts to be set.

Returns
0 in case of success or negative value in case of error.

enum *bt_audio_context* **bt_pacs_get_available_contexts**(enum *bt_audio_dir* dir)

Get the available contexts for an endpoint type.

Parameters

- **dir** – Direction of the endpoints to get contexts for.

Returns
Bitmask of available contexts.

int **bt_pacs_conn_set_available_contexts_for_conn**(struct bt_conn *conn, enum *bt_audio_dir* dir, enum *bt_audio_context* *contexts)

Set the available contexts for a given connection.

This function sets the available contexts value for a given `conn` connection object. If the `contexts` parameter is NULL the available contexts value is reset to default. The default value of the available contexts is set using *bt_pacs_set_available_contexts* function. The Available Context Value is reset to default on ACL disconnection.

Parameters

- **conn** – Connection object.
- **dir** – Direction of the endpoints to change available contexts for.
- **contexts** – The contexts to be set or NULL to reset to default.

Returns
0 in case of success or negative value in case of error.

enum *bt_audio_context* **bt_pacs_get_available_contexts_for_conn**(struct bt_conn *conn, enum *bt_audio_dir* dir)

Get the available contexts for a given connection.

This server function returns the available contexts value for a given `conn` connection object. The value returned is the one set with *bt_pacs_conn_set_available_contexts_for_conn* function or the default value set with *bt_pacs_set_available_contexts* function.

Parameters

- **conn** – Connection object.
- **dir** – Direction of the endpoints to get contexts for.

Return values
BT_AUDIO_CONTEXT_TYPE_PROHIBITED – if `conn` or `dir` are invalid

Returns
Bitmask of available contexts.

int **bt_pacs_set_supported_contexts**(enum *bt_audio_dir* dir, enum *bt_audio_context* contexts)

Set the supported contexts for an endpoint type.

Parameters

- **dir** – Direction of the endpoints to change available contexts for.
- **contexts** – The contexts to be set.

Returns

0 in case of success or negative value in case of error.

struct **bt_csip_set_member_cb**

#include <csip.h> Callback structure for the Coordinated Set Identification Service.

Public Members

void (***lock_changed**)(struct bt_conn *conn, struct bt_csip_set_member_svc_inst *svc_inst, bool locked)

Callback whenever the lock changes on the server.

Param conn
The connection to the client that changed the lock. NULL if server changed it, either by calling *bt_csip_set_member_lock()* or by timeout.

Param svc_inst
Pointer to the Coordinated Set Identification Service.

Param locked
Whether the lock was locked or released.

uint8_t (***sirk_read_req**)(struct bt_conn *conn, struct bt_csip_set_member_svc_inst *svc_inst)

Request from a peer device to read the sirk.

If this callback is not set, all clients will be allowed to read the SIRK unencrypted.

Param conn
The connection to the client that requested to read the SIRK.

Param svc_inst
Pointer to the Coordinated Set Identification Service.

Return
A BT_CSIP_READ_SIRK_REQ_RSP_* response code.

struct **bt_csip_set_member_register_param**

#include <csip.h> Register structure for Coordinated Set Identification Service.

Public Members

uint8_t **set_size**

Size of the set.

If set to 0, the set size characteristic won't be initialized.

uint8_t **sirk**[16]

The unique Set Identity Resolving Key (SIRK)

This shall be unique between different sets, and shall be the same for each set member for each set.

bool **lockable**

Boolean to set whether the set is lockable by clients.

Setting this to false will disable the lock characteristic.

uint8_t **rank**

Rank of this device in this set.

If the lockable parameter is set to true, this shall be > 0 and <= to the set_size. If the lockable parameter is set to false, this may be set to 0 to disable the rank characteristic.

struct *bt_csip_set_member_cb* ***cb**

Pointer to the callback structure.

const struct *bt_gatt_service* ***parent**

Parent service pointer.

Mandatory parent service pointer if this CSIS instance is included by another service. All CSIS instances when `CONFIG_BT_CSIP_SET_MEMBER_MAX_INSTANCE_COUNT` is above 1 shall be included by another service, as per the Coordinated Set Identification Profile (CSIP).

struct **bt_csip_set_coordinator_set_info**

#include <*csip.h*> Information about a specific set.

Public Members

uint8_t **sirk**[16]

The 16 octet set Set Identity Resolving Key (SIRK)

The SIRK may not be exposed by the server over Bluetooth, and may require an out-of-band solution.

uint8_t **set_size**

The size of the set.

Will be 0 if not exposed by the server.

uint8_t **rank**

The rank of the set on the remote device.

Will be 0 if not exposed by the server.

bool **lockable**

Whether or not the set can be locked on this device.

struct **bt_csip_set_coordinator_csis_inst**

#include <*csip.h*> Struct representing a coordinated set instance on a remote device.

The values in this struct will be populated during discovery of sets (*bt_csip_set_coordinator_discover()*).

Public Members

struct *bt_csip_set_coordinator_set_info* **info**
Information about the coordinated set.

void ***svc_inst**
Internally used pointer value.

struct **bt_csip_set_coordinator_set_member**
#include <*csip.h*> Struct representing a remote device as a set member.

Public Members

struct *bt_csip_set_coordinator_csis_inst* **insts**[0]
Array of Coordinated Set Identification Service instances for the remote device.

struct **bt_csip_set_coordinator_cb**
#include <*csip.h*> Struct to hold the Coordinated Set Identification Profile Set Coordinator callbacks.

These can be registered for usage with *bt_csip_set_coordinator_register_cb()*.

Public Members

bt_csip_set_coordinator_discover_cb **discover**
Callback when discovery has finished.

bt_csip_set_coordinator_lock_set_cb **lock_set**
Callback when locking a set has finished.

bt_csip_set_coordinator_lock_set_cb **release_set**
Callback when unlocking a set has finished.

bt_csip_set_coordinator_lock_changed_cb **lock_changed**
Callback when a set's lock state has changed.

bt_csip_set_coordinator_sirk_changed_cb **sirk_changed**
Callback when a set's SIRK has changed.

bt_csip_set_coordinator_ordered_access_cb_t **ordered_access**
Callback for the ordered access procedure.

struct **bt_pacs_cap**
#include <*pacs.h*> Published Audio Capability structure.

Public Members

const struct *bt_audio_codec_cap* *codec_cap

Codec capability reference.

Bluetooth Media

API Reference

Media Control Service

group bt_mcs

Media Control Service (MCS)

Definitions and types related to the Media Control Service and Media Control Profile specifications.

Since
3.0

Version
0.8.0

Playback speeds

The playback speed (s) is calculated by the value of 2 to the power of p divided by 64.

All values from -128 to 127 allowed, only some examples defined.

BT_MCS_PLAYBACK_SPEED_MIN

Minimum playback speed, resulting in 25 % speed.

BT_MCS_PLAYBACK_SPEED_QUARTER

Quarter playback speed, resulting in 25 % speed.

BT_MCS_PLAYBACK_SPEED_HALF

Half playback speed, resulting in 50 % speed.

BT_MCS_PLAYBACK_SPEED_UNITY

Unity playback speed, resulting in 100 % speed.

BT_MCS_PLAYBACK_SPEED_DOUBLE

Double playback speed, resulting in 200 % speed.

BT_MCS_PLAYBACK_SPEED_MAX

Max playback speed, resulting in 395.7 % speed (nearly 400 %)

Seeking speed

The allowed values for seeking speed are the range -64 to -4 (endpoints included), the value 0, and the range 4 to 64 (endpoints included).

`BT_MCS_SEEKING_SPEED_FACTOR_MAX`
: Maximum seeking speed - Can be negated.

`BT_MCS_SEEKING_SPEED_FACTOR_MIN`
: Minimum seeking speed - Can be negated.

`BT_MCS_SEEKING_SPEED_FACTOR_ZERO`
: No seeking.

Playing orders

`BT_MCS_PLAYING_ORDER_SINGLE_ONCE`
: A single track is played once; there is no next track.

`BT_MCS_PLAYING_ORDER_SINGLE_REPEAT`
: A single track is played repeatedly; the next track is the current track.

`BT_MCS_PLAYING_ORDER_INORDER_ONCE`
: The tracks within a group are played once in track order.

`BT_MCS_PLAYING_ORDER_INORDER_REPEAT`
: The tracks within a group are played in track order repeatedly.

`BT_MCS_PLAYING_ORDER_OLDEST_ONCE`
: The tracks within a group are played once only from the oldest first.

`BT_MCS_PLAYING_ORDER_OLDEST_REPEAT`
: The tracks within a group are played from the oldest first repeatedly.

`BT_MCS_PLAYING_ORDER_NEWEST_ONCE`
: The tracks within a group are played once only from the newest first.

`BT_MCS_PLAYING_ORDER_NEWEST_REPEAT`
: The tracks within a group are played from the newest first repeatedly.

`BT_MCS_PLAYING_ORDER_SHUFFLE_ONCE`
: The tracks within a group are played in random order once.

`BT_MCS_PLAYING_ORDER_SHUFFLE_REPEAT`
: The tracks within a group are played in random order repeatedly.

Playing orders supported

A bitmap, in the same order as the playing orders above.

Note that playing order 1 corresponds to bit 0, and so on.

`BT_MCS_PLAYING_ORDERS_SUPPORTED_SINGLE_ONCE`
: A single track is played once; there is no next track.

`BT_MCS_PLAYING_ORDERS_SUPPORTED_SINGLE_REPEAT`
: A single track is played repeatedly; the next track is the current track.

`BT_MCS_PLAYING_ORDERS_SUPPORTED_INORDER_ONCE`
: The tracks within a group are played once in track order.

`BT_MCS_PLAYING_ORDERS_SUPPORTED_INORDER_REPEAT`
: The tracks within a group are played in track order repeatedly.

`BT_MCS_PLAYING_ORDERS_SUPPORTED_OLDEST_ONCE`
: The tracks within a group are played once only from the oldest first.

`BT_MCS_PLAYING_ORDERS_SUPPORTED_OLDEST_REPEAT`
: The tracks within a group are played from the oldest first repeatedly.

`BT_MCS_PLAYING_ORDERS_SUPPORTED_NEWEST_ONCE`
: The tracks within a group are played once only from the newest first.

`BT_MCS_PLAYING_ORDERS_SUPPORTED_NEWEST_REPEAT`
: The tracks within a group are played from the newest first repeatedly.

`BT_MCS_PLAYING_ORDERS_SUPPORTED_SHUFFLE_ONCE`
: The tracks within a group are played in random order once.

`BT_MCS_PLAYING_ORDERS_SUPPORTED_SHUFFLE_REPEAT`
: The tracks within a group are played in random order repeatedly.

Media states

`BT_MCS_MEDIA_STATE_INACTIVE`
: The current track is invalid, and no track has been selected.

`BT_MCS_MEDIA_STATE_PLAYING`
: The media player is playing the current track.

`BT_MCS_MEDIA_STATE_PAUSED`
: The current track is paused.

  The media player has a current track, but it is not being played

`BT_MCS_MEDIA_STATE_SEEKING`
: The current track is fast forwarding or fast rewinding.

Media control point opcodes

`BT_MCS_OPC_PLAY`
: Start playing the current track.

`BT_MCS_OPC_PAUSE`
: Pause playing the current track.

`BT_MCS_OPC_FAST_REWIND`
: Fast rewind the current track.

`BT_MCS_OPC_FAST_FORWARD`
: Fast forward the current track.

`BT_MCS_OPC_STOP`
: Stop current activity and return to the paused state and set the current track position to the start of the current track.

`BT_MCS_OPC_MOVE_RELATIVE`
: Set a new current track position relative to the current track position.

`BT_MCS_OPC_PREV_SEGMENT`
: Set the current track position to the starting position of the previous segment of the current track.

`BT_MCS_OPC_NEXT_SEGMENT`
: Set the current track position to the starting position of the next segment of the current track.

`BT_MCS_OPC_FIRST_SEGMENT`
: Set the current track position to the starting position of the first segment of the current track.

`BT_MCS_OPC_LAST_SEGMENT`
: Set the current track position to the starting position of the last segment of the current track.

`BT_MCS_OPC_GOTO_SEGMENT`
: Set the current track position to the starting position of the nth segment of the current track.

`BT_MCS_OPC_PREV_TRACK`
: Set the current track to the previous track based on the playing order.

`BT_MCS_OPC_NEXT_TRACK`
: Set the current track to the next track based on the playing order.

`BT_MCS_OPC_FIRST_TRACK`
: Set the current track to the first track based on the playing order.

BT_MCS_OPC_LAST_TRACK
: Set the current track to the last track based on the playing order.

BT_MCS_OPC_GOTO_TRACK
: Set the current track to the nth track based on the playing order.

BT_MCS_OPC_PREV_GROUP
: Set the current group to the previous group in the sequence of groups.

BT_MCS_OPC_NEXT_GROUP
: Set the current group to the next group in the sequence of groups.

BT_MCS_OPC_FIRST_GROUP
: Set the current group to the first group in the sequence of groups.

BT_MCS_OPC_LAST_GROUP
: Set the current group to the last group in the sequence of groups.

BT_MCS_OPC_GOTO_GROUP
: Set the current group to the nth group in the sequence of groups.

Media control point supported opcodes values

BT_MCS_OPC_SUP_PLAY
: Support the Play opcode.

BT_MCS_OPC_SUP_PAUSE
: Support the Pause opcode.

BT_MCS_OPC_SUP_FAST_REWIND
: Support the Fast Rewind opcode.

BT_MCS_OPC_SUP_FAST_FORWARD
: Support the Fast Forward opcode.

BT_MCS_OPC_SUP_STOP
: Support the Stop opcode.

BT_MCS_OPC_SUP_MOVE_RELATIVE
: Support the Move Relative opcode.

BT_MCS_OPC_SUP_PREV_SEGMENT
: Support the Previous Segment opcode.

BT_MCS_OPC_SUP_NEXT_SEGMENT
: Support the Next Segment opcode.

`BT_MCS_OPC_SUP_FIRST_SEGMENT`
: Support the First Segment opcode.

`BT_MCS_OPC_SUP_LAST_SEGMENT`
: Support the Last Segment opcode.

`BT_MCS_OPC_SUP_GOTO_SEGMENT`
: Support the Goto Segment opcode.

`BT_MCS_OPC_SUP_PREV_TRACK`
: Support the Previous Track opcode.

`BT_MCS_OPC_SUP_NEXT_TRACK`
: Support the Next Track opcode.

`BT_MCS_OPC_SUP_FIRST_TRACK`
: Support the First Track opcode.

`BT_MCS_OPC_SUP_LAST_TRACK`
: Support the Last Track opcode.

`BT_MCS_OPC_SUP_GOTO_TRACK`
: Support the Goto Track opcode.

`BT_MCS_OPC_SUP_PREV_GROUP`
: Support the Previous Group opcode.

`BT_MCS_OPC_SUP_NEXT_GROUP`
: Support the Next Group opcode.

`BT_MCS_OPC_SUP_FIRST_GROUP`
: Support the First Group opcode.

`BT_MCS_OPC_SUP_LAST_GROUP`
: Support the Last Group opcode.

`BT_MCS_OPC_SUP_GOTO_GROUP`
: Support the Goto Group opcode.

Media control point notification result codes

`BT_MCS_OPC_NTF_SUCCESS`
: Action requested by the opcode write was completed successfully.

`BT_MCS_OPC_NTF_NOT_SUPPORTED`
: An invalid or unsupported opcode was used for the Media Control Point write.

BT_MCS_OPC_NTF_PLAYER_INACTIVE
: The Media Player State characteristic value is Inactive when the opcode is received or the result of the requested action of the opcode results in the Media Player State characteristic being set to Inactive.

BT_MCS_OPC_NTF_CANNOT_BE_COMPLETED
: The requested action of any Media Control Point write cannot be completed successfully because of a condition within the player.

Search control point type values

Reference: Media Control Service spec v1.0 section 3.20.2

BT_MCS_SEARCH_TYPE_TRACK_NAME
: Search for Track Name.

BT_MCS_SEARCH_TYPE_ARTIST_NAME
: Search for Artist Name.

BT_MCS_SEARCH_TYPE_ALBUM_NAME
: Search for Album Name.

BT_MCS_SEARCH_TYPE_GROUP_NAME
: Search for Group Name.

BT_MCS_SEARCH_TYPE_EARLIEST_YEAR
: Search for Earliest Year.

BT_MCS_SEARCH_TYPE_LATEST_YEAR
: Search for Latest Year.

BT_MCS_SEARCH_TYPE_GENRE
: Search for Genre.

BT_MCS_SEARCH_TYPE_ONLY_TRACKS
: Search for Tracks only.

BT_MCS_SEARCH_TYPE_ONLY_GROUPS
: Search for Groups only.

Search notification result codes

Reference: Media Control Service spec v1.0 section 3.20.2

BT_MCS_SCP_NTF_SUCCESS
: Search request was accepted; search has started.

BT_MCS_SCP_NTF_FAILURE
: Search request was invalid; no search started.

Group object object types

Reference: Media Control Service spec v1.0 section 4.4.1

`BT_MCS_GROUP_OBJECT_TRACK_TYPE`
: Group object type is track.

`BT_MCS_GROUP_OBJECT_GROUP_TYPE`
: Group object type is group.

Defines

`BT_MCS_ERR_LONG_VAL_CHANGED`
: A characteristic value has changed while a Read Long Value Characteristic sub-procedure is in progress.

`BT_MCS_OPCODES_SUPPORTED_LEN`
: Media control point supported opcodes length.

`SEARCH_LEN_MIN`
: Search control point minimum length.

  At least one search control item (SCI), consisting of the length octet and the type octet. (The * parameter field may be empty.)

`SEARCH_LEN_MAX`
: Search control point maximum length.

`SEARCH_SCI_LEN_MIN`
: Search control point item (SCI) minimum length.

  An SCI length can be as little as one byte, for an SCI that has only the type field. (The SCI len is the length of type + param.)

`SEARCH_PARAM_MAX`
: Search parameters maximum length

Media Proxy

group `bt_media_proxy`
: Media proxy module.

  The media proxy module is the connection point between media players and media controllers.

  Since
  : 3.0

  Version
  : 0.8.0

A media player has (access to) media content and knows how to navigate and play this content. A media controller reads or gets information from a player and controls the player by setting player parameters and giving the player commands.

The media proxy module allows media player implementations to make themselves available to media controllers. And it allows controllers to access, and get updates from, any player.

The media proxy module allows both local and remote control of local player instances: A media controller may be a local application, or it may be a Media Control Service relaying requests from a remote Media Control Client. There may be either local or remote control, or both, or even multiple instances of each.

Playback speed parameters

All values from -128 to 127 allowed, only some defined

`MEDIA_PROXY_PLAYBACK_SPEED_MIN`
: Minimum playback speed, resulting in 25 % speed.

`MEDIA_PROXY_PLAYBACK_SPEED_QUARTER`
: Quarter playback speed, resulting in 25 % speed.

`MEDIA_PROXY_PLAYBACK_SPEED_HALF`
: Half playback speed, resulting in 50 % speed.

`MEDIA_PROXY_PLAYBACK_SPEED_UNITY`
: Unity playback speed, resulting in 100 % speed.

`MEDIA_PROXY_PLAYBACK_SPEED_DOUBLE`
: Double playback speed, resulting in 200 % speed.

`MEDIA_PROXY_PLAYBACK_SPEED_MAX`
: Max playback speed, resulting in 395.7 % speed (nearly 400 %)

Seeking speed factors

The allowed values for seeking speed are the range -64 to -4 (endpoints included), the value 0, and the range 4 to 64 (endpoints included).

`MEDIA_PROXY_SEEKING_SPEED_FACTOR_MAX`
: Maximum seeking speed - Can be negated.

`MEDIA_PROXY_SEEKING_SPEED_FACTOR_MIN`
: Minimum seeking speed - Can be negated.

`MEDIA_PROXY_SEEKING_SPEED_FACTOR_ZERO`
: No seeking.

Playing orders

`MEDIA_PROXY_PLAYING_ORDER_SINGLE_ONCE`
A single track is played once; there is no next track.

`MEDIA_PROXY_PLAYING_ORDER_SINGLE_REPEAT`
A single track is played repeatedly; the next track is the current track.

`MEDIA_PROXY_PLAYING_ORDER_INORDER_ONCE`
The tracks within a group are played once in track order.

`MEDIA_PROXY_PLAYING_ORDER_INORDER_REPEAT`
The tracks within a group are played in track order repeatedly.

`MEDIA_PROXY_PLAYING_ORDER_OLDEST_ONCE`
The tracks within a group are played once only from the oldest first.

`MEDIA_PROXY_PLAYING_ORDER_OLDEST_REPEAT`
The tracks within a group are played from the oldest first repeatedly.

`MEDIA_PROXY_PLAYING_ORDER_NEWEST_ONCE`
The tracks within a group are played once only from the newest first.

`MEDIA_PROXY_PLAYING_ORDER_NEWEST_REPEAT`
The tracks within a group are played from the newest first repeatedly.

`MEDIA_PROXY_PLAYING_ORDER_SHUFFLE_ONCE`
The tracks within a group are played in random order once.

`MEDIA_PROXY_PLAYING_ORDER_SHUFFLE_REPEAT`
The tracks within a group are played in random order repeatedly.

Playing orders supported

A bitmap, in the same order as the playing orders above.

Note that playing order 1 corresponds to bit 0, and so on.

`MEDIA_PROXY_PLAYING_ORDERS_SUPPORTED_SINGLE_ONCE`
A single track is played once; there is no next track.

`MEDIA_PROXY_PLAYING_ORDERS_SUPPORTED_SINGLE_REPEAT`
A single track is played repeatedly; the next track is the current track.

`MEDIA_PROXY_PLAYING_ORDERS_SUPPORTED_INORDER_ONCE`
The tracks within a group are played once in track order.

`MEDIA_PROXY_PLAYING_ORDERS_SUPPORTED_INORDER_REPEAT`
The tracks within a group are played in track order repeatedly.

MEDIA_PROXY_PLAYING_ORDERS_SUPPORTED_OLDEST_ONCE
: The tracks within a group are played once only from the oldest first.

MEDIA_PROXY_PLAYING_ORDERS_SUPPORTED_OLDEST_REPEAT
: The tracks within a group are played from the oldest first repeatedly.

MEDIA_PROXY_PLAYING_ORDERS_SUPPORTED_NEWEST_ONCE
: The tracks within a group are played once only from the newest first.

MEDIA_PROXY_PLAYING_ORDERS_SUPPORTED_NEWEST_REPEAT
: The tracks within a group are played from the newest first repeatedly.

MEDIA_PROXY_PLAYING_ORDERS_SUPPORTED_SHUFFLE_ONCE
: The tracks within a group are played in random order once.

MEDIA_PROXY_PLAYING_ORDERS_SUPPORTED_SHUFFLE_REPEAT
: The tracks within a group are played in random order repeatedly.

Media player states

MEDIA_PROXY_STATE_INACTIVE
: The current track is invalid, and no track has been selected.

MEDIA_PROXY_STATE_PLAYING
: The media player is playing the current track.

MEDIA_PROXY_STATE_PAUSED
: The current track is paused.

  The media player has a current track, but it is not being played

MEDIA_PROXY_STATE_SEEKING
: The current track is fast forwarding or fast rewinding.

MEDIA_PROXY_STATE_LAST
: Used internally as the last state value.

Media player command opcodes

MEDIA_PROXY_OP_PLAY
: Start playing the current track.

MEDIA_PROXY_OP_PAUSE
: Pause playing the current track.

MEDIA_PROXY_OP_FAST_REWIND
: Fast rewind the current track.

MEDIA_PROXY_OP_FAST_FORWARD
: Fast forward the current track.

MEDIA_PROXY_OP_STOP
: Stop current activity and return to the paused state and set the current track position to the start of the current track.

MEDIA_PROXY_OP_MOVE_RELATIVE
: Set a new current track position relative to the current track position.

MEDIA_PROXY_OP_PREV_SEGMENT
: Set the current track position to the starting position of the previous segment of the current track.

MEDIA_PROXY_OP_NEXT_SEGMENT
: Set the current track position to the starting position of the next segment of the current track.

MEDIA_PROXY_OP_FIRST_SEGMENT
: Set the current track position to the starting position of the first segment of the current track.

MEDIA_PROXY_OP_LAST_SEGMENT
: Set the current track position to the starting position of the last segment of the current track.

MEDIA_PROXY_OP_GOTO_SEGMENT
: Set the current track position to the starting position of the nth segment of the current track.

MEDIA_PROXY_OP_PREV_TRACK
: Set the current track to the previous track based on the playing order.

MEDIA_PROXY_OP_NEXT_TRACK
: Set the current track to the next track based on the playing order.

MEDIA_PROXY_OP_FIRST_TRACK
: Set the current track to the first track based on the playing order.

MEDIA_PROXY_OP_LAST_TRACK
: Set the current track to the last track based on the playing order.

MEDIA_PROXY_OP_GOTO_TRACK
: Set the current track to the nth track based on the playing order.

MEDIA_PROXY_OP_PREV_GROUP
: Set the current group to the previous group in the sequence of groups.

MEDIA_PROXY_OP_NEXT_GROUP
: Set the current group to the next group in the sequence of groups.

MEDIA_PROXY_OP_FIRST_GROUP
: Set the current group to the first group in the sequence of groups.

MEDIA_PROXY_OP_LAST_GROUP
: Set the current group to the last group in the sequence of groups.

MEDIA_PROXY_OP_GOTO_GROUP
: Set the current group to the nth group in the sequence of groups.

Unnamed Group

MEDIA_PROXY_OP_SUP_PLAY
: Media player supported command opcodes.

  Support the Play opcode

MEDIA_PROXY_OP_SUP_PAUSE
: Support the Pause opcode.

MEDIA_PROXY_OP_SUP_FAST_REWIND
: Support the Fast Rewind opcode.

MEDIA_PROXY_OP_SUP_FAST_FORWARD
: Support the Fast Forward opcode.

MEDIA_PROXY_OP_SUP_STOP
: Support the Stop opcode.

MEDIA_PROXY_OP_SUP_MOVE_RELATIVE
: Support the Move Relative opcode.

MEDIA_PROXY_OP_SUP_PREV_SEGMENT
: Support the Previous Segment opcode.

MEDIA_PROXY_OP_SUP_NEXT_SEGMENT
: Support the Next Segment opcode.

MEDIA_PROXY_OP_SUP_FIRST_SEGMENT
: Support the First Segment opcode.

MEDIA_PROXY_OP_SUP_LAST_SEGMENT
: Support the Last Segment opcode.

MEDIA_PROXY_OP_SUP_GOTO_SEGMENT
: Support the Goto Segment opcode.

MEDIA_PROXY_OP_SUP_PREV_TRACK
: Support the Previous Track opcode.

MEDIA_PROXY_OP_SUP_NEXT_TRACK
: Support the Next Track opcode.

MEDIA_PROXY_OP_SUP_FIRST_TRACK
: Support the First Track opcode.

MEDIA_PROXY_OP_SUP_LAST_TRACK
: Support the Last Track opcode.

MEDIA_PROXY_OP_SUP_GOTO_TRACK
: Support the Goto Track opcode.

MEDIA_PROXY_OP_SUP_PREV_GROUP
: Support the Previous Group opcode.

MEDIA_PROXY_OP_SUP_NEXT_GROUP
: Support the Next Group opcode.

MEDIA_PROXY_OP_SUP_FIRST_GROUP
: Support the First Group opcode.

MEDIA_PROXY_OP_SUP_LAST_GROUP
: Support the Last Group opcode.

MEDIA_PROXY_OP_SUP_GOTO_GROUP
: Support the Goto Group opcode.

Media player command result codes

MEDIA_PROXY_CMD_SUCCESS
: Action requested by the opcode write was completed successfully.

MEDIA_PROXY_CMD_NOT_SUPPORTED
: An invalid or unsupported opcode was used for the Media Control Point write.

MEDIA_PROXY_CMD_PLAYER_INACTIVE
: The Media Player State characteristic value is Inactive when the opcode is received or the result of the requested action of the opcode results in the Media Player State characteristic being set to Inactive.

MEDIA_PROXY_CMD_CANNOT_BE_COMPLETED
: The requested action of any Media Control Point write cannot be completed successfully because of a condition within the player.

Search operation type values

MEDIA_PROXY_SEARCH_TYPE_TRACK_NAME
: Search for Track Name.

MEDIA_PROXY_SEARCH_TYPE_ARTIST_NAME
: Search for Artist Name.

MEDIA_PROXY_SEARCH_TYPE_ALBUM_NAME
: Search for Album Name.

MEDIA_PROXY_SEARCH_TYPE_GROUP_NAME
: Search for Group Name.

MEDIA_PROXY_SEARCH_TYPE_EARLIEST_YEAR
: Search for Earliest Year.

MEDIA_PROXY_SEARCH_TYPE_LATEST_YEAR
: Search for Latest Year.

MEDIA_PROXY_SEARCH_TYPE_GENRE
: Search for Genre.

MEDIA_PROXY_SEARCH_TYPE_ONLY_TRACKS
: Search for Tracks only.

MEDIA_PROXY_SEARCH_TYPE_ONLY_GROUPS
: Search for Groups only.

Search notification result codes

MEDIA_PROXY_SEARCH_SUCCESS
: Search request was accepted; search has started.

MEDIA_PROXY_SEARCH_FAILURE
: Search request was invalid; no search started.

Group object object types

MEDIA_PROXY_GROUP_OBJECT_TRACK_TYPE
: Group object type is track.

MEDIA_PROXY_GROUP_OBJECT_GROUP_TYPE
: Group object type is group.

Defines

MEDIA_PROXY_OPCODES_SUPPORTED_LEN

Media player supported opcodes length.

Functions

int **media_proxy_ctrl_register**(struct *media_proxy_ctrl_cbs* *ctrl_cbs)

Register a controller with the media_proxy.

Parameters

- **ctrl_cbs** – Callbacks to the controller

Returns

0 if success, errno on failure

int **media_proxy_ctrl_discover_player**(struct bt_conn *conn)

Discover a remote media player.

Discover a remote media player instance. The remote player instance will be discovered, and accessed, using Bluetooth, via the media control client and a remote media control service. This call will start a GATT discovery of the Media Control Service on the peer, and setup handles and subscriptions.

This shall be called once before any other actions can be executed for the remote player. The remote player instance will be returned in the discover_player() callback.

Parameters

- **conn** – The connection to do discovery for

Returns

0 if success, errno on failure

int **media_proxy_ctrl_get_player_name**(struct media_player *player)

Read Media Player Name.

Parameters

- **player** – Media player instance pointer

Returns

0 if success, errno on failure.

int **media_proxy_ctrl_get_icon_id**(struct media_player *player)

Read Icon Object ID.

Get an ID (48 bit) that can be used to retrieve the Icon Object from an Object Transfer Service

See the Media Control Service spec v1.0 sections 3.2 and 4.1 for a description of the Icon Object.

Requires Object Transfer Service

Parameters

- **player** – Media player instance pointer

Returns

0 if success, errno on failure.

int media_proxy_ctrl_get_icon_url(struct media_player *player)

Read Icon URL.

Get a URL to the media player's icon.

Parameters

- player – Media player instance pointer

int media_proxy_ctrl_get_track_title(struct media_player *player)

Read Track Title.

Parameters

- player – Media player instance pointer

Returns

0 if success, errno on failure.

int media_proxy_ctrl_get_track_duration(struct media_player *player)

Read Track Duration.

The duration of a track is measured in hundredths of a second.

Parameters

- player – Media player instance pointer

Returns

0 if success, errno on failure.

int media_proxy_ctrl_get_track_position(struct media_player *player)

Read Track Position.

The position of the player (the playing position) is measured in hundredths of a second from the beginning of the track

Parameters

- player – Media player instance pointer

Returns

0 if success, errno on failure.

int media_proxy_ctrl_set_track_position(struct media_player *player, int32_t position)

Set Track Position.

Set the playing position of the media player in the current track. The position is given in hundredths of a second, from the beginning of the track of the track for positive values, and (backwards) from the end of the track for negative values.

Parameters

- player – Media player instance pointer
- position – The track position to set

Returns

0 if success, errno on failure.

int media_proxy_ctrl_get_playback_speed(struct media_player *player)

Get Playback Speed.

The playback speed parameter is related to the actual playback speed as follows: actual playback speed = 2^(speed_parameter/64)

A speed parameter of 0 corresponds to unity speed playback (i.e. playback at "normal" speed). A speed parameter of -128 corresponds to playback at one fourth of normal speed, 127 corresponds to playback at almost four times the normal speed.

Parameters

- `player` – Media player instance pointer

Returns
0 if success, errno on failure.

int `media_proxy_ctrl_set_playback_speed`(struct media_player *player, int8_t speed)

Set Playback Speed.

See the get_playback_speed() function for an explanation of the playback speed parameter.

Note that the media player may not support all possible values of the playback speed parameter. If the value given is not supported, and is higher than the current value, the player should set the playback speed to the next higher supported value. (And correspondingly to the next lower supported value for given values lower than the current value.)

Parameters

- `player` – Media player instance pointer
- `speed` – The playback speed parameter to set

Returns
0 if success, errno on failure.

int `media_proxy_ctrl_get_seeking_speed`(struct media_player *player)

Get Seeking Speed.

The seeking speed gives the speed with which the player is seeking. It is a factor, relative to real-time playback speed - a factor four means seeking happens at four times the real-time playback speed. Positive values are for forward seeking, negative values for backwards seeking.

The seeking speed is not settable - a non-zero seeking speed is the result of "fast rewind" of "fast forward" commands.

Parameters

- `player` – Media player instance pointer

Returns
0 if success, errno on failure.

int `media_proxy_ctrl_get_track_segments_id`(struct media_player *player)

Read Current Track Segments Object ID.

Get an ID (48 bit) that can be used to retrieve the Current Track Segments Object from an Object Transfer Service

See the Media Control Service spec v1.0 sections 3.10 and 4.2 for a description of the Track Segments Object.

Requires Object Transfer Service

Parameters

- `player` – Media player instance pointer

Returns
0 if success, errno on failure.

int `media_proxy_ctrl_get_current_track_id`(struct media_player *player)

Read Current Track Object ID.

Get an ID (48 bit) that can be used to retrieve the Current Track Object from an Object Transfer Service

See the Media Control Service spec v1.0 sections 3.11 and 4.3 for a description of the Current Track Object.

Requires Object Transfer Service

Parameters

- `player` – Media player instance pointer

Returns
0 if success, errno on failure.

int `media_proxy_ctrl_set_current_track_id`(struct media_player *player, uint64_t id)

Set Current Track Object ID.

Change the player's current track to the track given by the ID. (Behaves similarly to the goto track command.)

Requires Object Transfer Service

Parameters

- `player` – Media player instance pointer
- `id` – The ID of a track object

Returns
0 if success, errno on failure.

int `media_proxy_ctrl_get_next_track_id`(struct media_player *player)

Read Next Track Object ID.

Get an ID (48 bit) that can be used to retrieve the Next Track Object from an Object Transfer Service

Requires Object Transfer Service

Parameters

- `player` – Media player instance pointer

Returns
0 if success, errno on failure.

int `media_proxy_ctrl_set_next_track_id`(struct media_player *player, uint64_t id)

Set Next Track Object ID.

Change the player's next track to the track given by the ID.

Requires Object Transfer Service

Parameters

- `player` – Media player instance pointer
- `id` – The ID of a track object

Returns
0 if success, errno on failure.

int `media_proxy_ctrl_get_parent_group_id`(struct media_player *player)

Read Parent Group Object ID.

Get an ID (48 bit) that can be used to retrieve the Parent Track Object from an Object Transfer Service

The parent group is the parent of the current group.

See the Media Control Service spec v1.0 sections 3.14 and 4.4 for a description of the Current Track Object.

Requires Object Transfer Service

Parameters

- `player` – Media player instance pointer

Returns
0 if success, errno on failure.

int `media_proxy_ctrl_get_current_group_id`(struct media_player *player)

Read Current Group Object ID.

Get an ID (48 bit) that can be used to retrieve the Current Track Object from an Object Transfer Service

See the Media Control Service spec v1.0 sections 3.14 and 4.4 for a description of the Current Group Object.

Requires Object Transfer Service

Parameters

- `player` – Media player instance pointer

Returns
0 if success, errno on failure.

int `media_proxy_ctrl_set_current_group_id`(struct media_player *player, uint64_t id)

Set Current Group Object ID.

Change the player's current group to the group given by the ID, and the current track to the first track in that group.

Requires Object Transfer Service

Parameters

- `player` – Media player instance pointer
- `id` – The ID of a group object

Returns
0 if success, errno on failure.

int `media_proxy_ctrl_get_playing_order`(struct media_player *player)

Read Playing Order.

Parameters

- `player` – Media player instance pointer

Returns
0 if success, errno on failure.

int `media_proxy_ctrl_set_playing_order`(struct media_player *player, uint8_t order)

Set Playing Order.

Set the media player's playing order

Parameters

- `player` – Media player instance pointer
- `order` – The playing order to set

Returns
0 if success, errno on failure.

int `media_proxy_ctrl_get_playing_orders_supported`(struct media_player *player)

Read Playing Orders Supported.

Read a bitmap containing the media player's supported playing orders.

Parameters

- `player` – Media player instance pointer

Returns
0 if success, errno on failure.

int `media_proxy_ctrl_get_media_state`(struct media_player *player)

Read Media State.

Read the media player's state

Parameters

- `player` – Media player instance pointer

Returns
0 if success, errno on failure.

int `media_proxy_ctrl_send_command`(struct media_player *player, const struct *mpl_cmd* *command)

Send Command.

Send a command to the media player. Commands may cause the media player to change its state May result in two callbacks - one for the actual sending of the command to the player, one for the result of the command from the player.

Parameters

- `player` – Media player instance pointer
- `command` – The command to send

Returns
0 if success, errno on failure.

int `media_proxy_ctrl_get_commands_supported`(struct media_player *player)

Read Commands Supported.

Read a bitmap containing the media player's supported command opcodes.

Parameters

- `player` – Media player instance pointer

Returns
0 if success, errno on failure.

int `media_proxy_ctrl_send_search`(struct media_player *player, const struct *mpl_search* *search)

Set Search.

Write a search to the media player. If the search is successful, the search results will be available as a group object in the Object Transfer Service (OTS).

May result in up to three callbacks

- one for the actual sending of the search to the player
- one for the result code for the search from the player
- if the search is successful, one for the search results object ID in the OTs

Requires Object Transfer Service

Parameters

- `player` – Media player instance pointer
- `search` – The search to write

Returns
0 if success, errno on failure.

int **media_proxy_ctrl_get_search_results_id**(struct media_player *player)

Read Search Results Object ID.

Get an ID (48 bit) that can be used to retrieve the Search Results Object from an Object Transfer Service

The search results object is a group object. The search results object only exists if a successful search operation has been done.

Requires Object Transfer Service

Parameters

- **player** – Media player instance pointer

Returns
0 if success, errno on failure.

uint8_t **media_proxy_ctrl_get_content_ctrl_id**(struct media_player *player)

Read Content Control ID.

The content control ID identifies a content control service on a device, and links it to the corresponding audio stream.

Parameters

- **player** – Media player instance pointer

Returns
0 if success, errno on failure.

int **media_proxy_pl_register**(struct *media_proxy_pl_calls* *pl_calls)

Register a player with the media proxy.

Register a player with the media proxy module, for use by media controllers.

The media proxy may call any non-NULL function pointers in the supplied *media_proxy_pl_calls* structure.

Parameters

- **pl_calls** – Function pointers to the media player's calls

Returns
0 if success, errno on failure

int **media_proxy_pl_init**(void)

Initialize player.

TODO: Move to player header file

struct bt_ots ***bt_mcs_get_ots**(void)

Get the pointer of the Object Transfer Service used by the Media Control Service.

TODO: Find best location for this call, and move this one also

void **media_proxy_pl_name_cb**(const char *name)

Player name changed callback.

To be called when the player's name is changed.

Parameters

- **name** – The name of the player

void media_proxy_pl_icon_url_cb(const char *url)

Player icon URL changed callback.

To be called when the player's icon URL is changed.

Parameters

- url – The URL of the player's icon

void media_proxy_pl_track_changed_cb(void)

Track changed callback.

To be called when the player's current track is changed

void media_proxy_pl_track_title_cb(char *title)

Track title callback.

To be called when the player's current track is changed

Parameters

- title – The title of the track

void media_proxy_pl_track_duration_cb(int32_t duration)

Track duration callback.

To be called when the current track's duration is changed (e.g. due to a track change)

The track duration is given in hundredths of a second.

Parameters

- duration – The track duration

void media_proxy_pl_track_position_cb(int32_t position)

Track position callback.

To be called when the media player's position in the track is changed, or when the player is paused or similar.

Exception: This callback should not be called when the position changes during regular playback, i.e. while the player is playing and playback happens at a constant speed.

The track position is given in hundredths of a second from the start of the track.

Parameters

- position – The media player's position in the track

void media_proxy_pl_playback_speed_cb(int8_t speed)

Playback speed callback.

To be called when the playback speed is changed.

Parameters

- speed – The playback speed parameter

void media_proxy_pl_seeking_speed_cb(int8_t speed)

Seeking speed callback.

To be called when the seeking speed is changed.

Parameters

- speed – The seeking speed factor

void media_proxy_pl_current_track_id_cb(uint64_t id)

Current track object ID callback.

To be called when the ID of the current track is changed (e.g. due to a track change).

Parameters

- id – The ID of the current track object in the OTS

void media_proxy_pl_next_track_id_cb(uint64_t id)

Next track object ID callback.

To be called when the ID of the current track is changes

Parameters

- id – The ID of the next track object in the OTS

void media_proxy_pl_parent_group_id_cb(uint64_t id)

Parent group object ID callback.

To be called when the ID of the parent group is changed

Parameters

- id – The ID of the parent group object in the OTS

void media_proxy_pl_current_group_id_cb(uint64_t id)

Current group object ID callback.

To be called when the ID of the current group is changed

Parameters

- id – The ID of the current group object in the OTS

void media_proxy_pl_playing_order_cb(uint8_t order)

Playing order callback.

To be called when the playing order is changed

Parameters

- order – The playing order

void media_proxy_pl_media_state_cb(uint8_t state)

Media state callback.

To be called when the media state is changed

Parameters

- state – The media player's state

void media_proxy_pl_command_cb(const struct *mpl_cmd_ntf* *cmd_ntf)

Command callback.

To be called when a command has been sent, to notify whether the command was successfully performed or not. See the MEDIA_PROXY_CMD_* result code defines.

Parameters

- cmd_ntf – The result of the command

void media_proxy_pl_commands_supported_cb(uint32_t opcodes)

Commands supported callback.

To be called when the set of commands supported is changed

Parameters

- opcodes – The supported commands opcodes

void **media_proxy_pl_search_cb**(uint8_t result_code)

Search callback.

To be called when a search has been set to notify whether the search was successfully performed or not. See the MEDIA_PROXY_SEARCH_* result code defines.

The actual results of the search, if successful, can be found in the search results object.

Parameters

- **result_code** – The result (success or failure) of the search

void **media_proxy_pl_search_results_id_cb**(uint64_t id)

Search Results object ID callback.

To be called when the ID of the search results is changed (typically as the result of a new successful search).

Parameters

- **id** – The ID of the search results object in the OTS

struct **mpl_cmd**

#include <media_proxy.h> Media player command.

Public Members

uint8_t **opcode**

The opcode.

See the MEDIA_PROXY_OP_* values

bool **use_param**

Whether or not the *mpl_cmd::param* is used.

int32_t **param**

A 32-bit signed parameter.

The parameter value depends on the *mpl_cmd::opcode*

struct **mpl_cmd_ntf**

#include <media_proxy.h> Media command notification.

Public Members

uint8_t **requested_opcode**

The opcode that was sent.

uint8_t **result_code**

The result of the operation

struct **mpl_sci**

#include <media_proxy.h> Search control item.

Public Members

uint8_t **len**

Length of type and parameter.

uint8_t **type**

MEDIA_PROXY_SEARCH_TYPE_<...>

char **param**[62]

Search parameter.

struct **mpl_search**

#include <*media_proxy.h*> Search.

Public Members

uint8_t **len**

The length of the *mpl_search::search* value.

char **search**[64]

Concatenated search control items - (type, length, param)

struct **media_proxy_ctrl_cbs**

#include <*media_proxy.h*> Callbacks to a controller, from the media proxy.

Given by a controller when registering

Public Members

void (***local_player_instance**)(struct media_player *player, int err)

Media Player Instance callback.

Called when the local Media Player instance is registered or read (TODO). Also called if the local player instance is already registered when the controller is registered. Provides the controller with the pointer to the local player instance.

Param player
Media player instance pointer

Param err
Error value. 0 on success, or errno on negative value.

void (***player_name_recv**)(struct media_player *player, int err, const char *name)

Media Player Name receive callback.

Called when the Media Player Name is read or changed See also media_proxy_ctrl_name_get()

Param player
Media player instance pointer

Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.

Param name
The name of the media player

void (*`icon_id_recv`)(struct media_player *player, int err, uint64_t id)

Media Player Icon Object ID receive callback.

Called when the Media Player Icon Object ID is read See also *media_proxy_ctrl_get_icon_id()*

Param player
Media player instance pointer
Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.
Param id
The ID of the Icon object in the Object Transfer Service (48 bits)

void (*`icon_url_recv`)(struct media_player *player, int err, const char *url)

Media Player Icon URL receive callback.

Called when the Media Player Icon URL is read See also *media_proxy_ctrl_get_icon_url()*

Param player
Media player instance pointer
Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.
Param url
The URL of the icon

void (*`track_changed_recv`)(struct media_player *player, int err)

Track changed receive callback.

Called when the Current Track is changed

Param player
Media player instance pointer
Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.

void (*`track_title_recv`)(struct media_player *player, int err, const char *title)

Track Title receive callback.

Called when the Track Title is read or changed See also *media_proxy_ctrl_get_track_title()*

Param player
Media player instance pointer
Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.
Param title
The title of the current track

void (*`track_duration_recv`)(struct media_player *player, int err, int32_t duration)

Track Duration receive callback.

Called when the Track Duration is read or changed See also *media_proxy_ctrl_get_track_duration()*

Param player
Media player instance pointer
Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.
Param duration
The duration of the current track

void (*`track_position_recv`)(struct media_player *player, int err, int32_t position)

Track Position receive callback.

Called when the Track Position is read or changed See also *media_proxy_ctrl_get_track_position()* and *media_proxy_ctrl_set_track_position()*
Param player
Media player instance pointer
Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.
Param position
The player's position in the track

void (*`track_position_write`)(struct media_player *player, int err, int32_t position)

Track Position write callback.

Called when the Track Position is written See also *media_proxy_ctrl_set_track_position()*.
Param player
Media player instance pointer
Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.
Param position
The position given attempted to write

void (*`playback_speed_recv`)(struct media_player *player, int err, int8_t speed)

Playback Speed receive callback.

Called when the Playback Speed is read or changed See also *media_proxy_ctrl_get_playback_speed()* and *media_proxy_ctrl_set_playback_speed()*
Param player
Media player instance pointer
Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.
Param speed
The playback speed parameter

void (*`playback_speed_write`)(struct media_player *player, int err, int8_t speed)

Playback Speed write callback.

Called when the Playback Speed is written See also *media_proxy_ctrl_set_playback_speed()*
Param player
Media player instance pointer
Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.

Param speed
The playback speed parameter attempted to write

void (*`seeking_speed_recv`)(struct media_player *player, int err, int8_t speed)

Seeking Speed receive callback.

Called when the Seeking Speed is read or changed See also *media_proxy_ctrl_get_seeking_speed()*

Param player
Media player instance pointer
Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.
Param speed
The seeking speed factor

void (*`track_segments_id_recv`)(struct media_player *player, int err, uint64_t id)

Track Segments Object ID receive callback.

Called when the Track Segments Object ID is read See also *media_proxy_ctrl_get_track_segments_id()*

Param player
Media player instance pointer
Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.
Param id
The ID of the track segments object in Object Transfer Service (48 bits)

void (*`current_track_id_recv`)(struct media_player *player, int err, uint64_t id)

Current Track Object ID receive callback.

Called when the Current Track Object ID is read or changed See also *media_proxy_ctrl_get_current_track_id()* and *media_proxy_ctrl_set_current_track_id()*

Param player
Media player instance pointer
Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.
Param id
The ID of the current track object in Object Transfer Service (48 bits)

void (*`current_track_id_write`)(struct media_player *player, int err, uint64_t id)

Current Track Object ID write callback.

Called when the Current Track Object ID is written See also *media_proxy_ctrl_set_current_track_id()*

Param player
Media player instance pointer
Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.
Param id
The ID (48 bits) attempted to write

void (*`next_track_id_recv`)(struct media_player *player, int err, uint64_t id)

Next Track Object ID receive callback.

Called when the Next Track Object ID is read or changed See also *media_proxy_ctrl_get_next_track_id()* and *media_proxy_ctrl_set_next_track_id()*

Param player
Media player instance pointer

Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.

Param id
The ID of the next track object in Object Transfer Service (48 bits)

void (*`next_track_id_write`)(struct media_player *player, int err, uint64_t id)

Next Track Object ID write callback.

Called when the Next Track Object ID is written See also *media_proxy_ctrl_set_next_track_id()*

Param player
Media player instance pointer

Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.

Param id
The ID (48 bits) attempted to write

void (*`parent_group_id_recv`)(struct media_player *player, int err, uint64_t id)

Parent Group Object ID receive callback.

Called when the Parent Group Object ID is read or changed See also *media_proxy_ctrl_get_parent_group_id()*

Param player
Media player instance pointer

Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.

Param id
The ID of the parent group object in Object Transfer Service (48 bits)

void (*`current_group_id_recv`)(struct media_player *player, int err, uint64_t id)

Current Group Object ID receive callback.

Called when the Current Group Object ID is read or changed See also *media_proxy_ctrl_get_current_group_id()* and *media_proxy_ctrl_set_current_group_id()*

Param player
Media player instance pointer

Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.

Param id
The ID of the current group object in Object Transfer Service (48 bits)

void (*`current_group_id_write`)(struct media_player *player, int err, uint64_t id)

Current Group Object ID write callback.

Called when the Current Group Object ID is written See also *media_proxy_ctrl_set_current_group_id()*

Param player
Media player instance pointer

Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.
Param id
The ID (48 bits) attempted to write

void (*`playing_order_recv`)(struct media_player *player, int err, uint8_t order)

Playing Order receive callback.

Called when the Playing Order is read or changed See also *media_proxy_ctrl_get_playing_order()* and *media_proxy_ctrl_set_playing_order()*
Param player
Media player instance pointer
Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.
Param order
The playing order

void (*`playing_order_write`)(struct media_player *player, int err, uint8_t order)

Playing Order write callback.

Called when the Playing Order is written See also *media_proxy_ctrl_set_playing_order()*
Param player
Media player instance pointer
Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.
Param order
The playing order attempted to write

void (*`playing_orders_supported_recv`)(struct media_player *player, int err, uint16_t orders)

Playing Orders Supported receive callback.

Called when the Playing Orders Supported is read See also *media_proxy_ctrl_get_playing_orders_supported()*
Param player
Media player instance pointer
Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.
Param orders
The playing orders supported

void (*`media_state_recv`)(struct media_player *player, int err, uint8_t state)

Media State receive callback.

Called when the Media State is read or changed See also *media_proxy_ctrl_get_media_state()* and *media_proxy_ctrl_send_command()*
Param player
Media player instance pointer
Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.
Param state
The media player state

void (*command_send)(struct media_player *player, int err, const struct *mpl_cmd* *cmd)

Command send callback.

Called when a command has been sent See also *media_proxy_ctrl_send_command()*

Param player
Media player instance pointer

Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.

Param cmd
The command sent

void (*command_recv)(struct media_player *player, int err, const struct *mpl_cmd_ntf* *result)

Command result receive callback.

Called when a command result has been received See also *media_proxy_ctrl_send_command()*

Param player
Media player instance pointer

Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.

Param result
The result received

void (*commands_supported_recv)(struct media_player *player, int err, uint32_t opcodes)

Commands supported receive callback.

Called when the Commands Supported is read or changed See also *media_proxy_ctrl_get_commands_supported()*

Param player
Media player instance pointer

Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.

Param opcodes
The supported command opcodes (bitmap)

void (*search_send)(struct media_player *player, int err, const struct *mpl_search* *search)

Search send callback.

Called when a search has been sent See also *media_proxy_ctrl_send_search()*

Param player
Media player instance pointer

Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.

Param search
The search sent

void (*search_recv)(struct media_player *player, int err, uint8_t result_code)

Search result code receive callback.

Called when a search result code has been received See also *media_proxy_ctrl_send_search()*

The search result code tells whether the search was successful or not. For a successful search, the actual results of the search (i.e. what was found as a result of the search)can be accessed using the Search Results Object ID. The Search Results Object ID has a separate callback - *search_results_id_recv()*.

Param player
Media player instance pointer

Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.

Param result_code
Search result code

void (*`search_results_id_recv`)(struct media_player *player, int err, uint64_t id)

Search Results Object ID receive callback See also *media_proxy_ctrl_get_search_results_id()*

Called when the Search Results Object ID is read or changed

Param player
Media player instance pointer

Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.

Param id
The ID of the search results object in Object Transfer Service (48 bits)

void (*`content_ctrl_id_recv`)(struct media_player *player, int err, uint8_t ccid)

Content Control ID receive callback.

Called when the Content Control ID is read See also *media_proxy_ctrl_get_content_ctrl_id()*

Param player
Media player instance pointer

Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.

Param ccid
The content control ID

struct `media_proxy_pl_calls`

#include <media_proxy.h> Available calls in a player, that the media proxy can call.

Given by a player when registering.

Public Members

const char *(*`get_player_name`)(void)

Read Media Player Name.

Return
The name of the media player

uint64_t (*`get_icon_id`)(void)

Read Icon Object ID.

Get an ID (48 bit) that can be used to retrieve the Icon Object from an Object Transfer Service

See the Media Control Service spec v1.0 sections 3.2 and 4.1 for a description of the Icon Object.

Return
The Icon Object ID

const char *(***get_icon_url**)(void)

Read Icon URL.

Get a URL to the media player's icon.

Return
The URL of the Icon

const char *(***get_track_title**)(void)

Read Track Title.

Return
The title of the current track

int32_t (***get_track_duration**)(void)

Read Track Duration.

The duration of a track is measured in hundredths of a second.

Return
The duration of the current track

int32_t (***get_track_position**)(void)

Read Track Position.

The position of the player (the playing position) is measured in hundredths of a second from the beginning of the track

Return
The position of the player in the current track

void (***set_track_position**)(int32_t position)

Set Track Position.

Set the playing position of the media player in the current track. The position is given in hundredths of a second, from the beginning of the track of the track for positive values, and (backwards) from the end of the track for negative values.

Param position
The player position to set

int8_t (***get_playback_speed**)(void)

Get Playback Speed.

The playback speed parameter is related to the actual playback speed as follows: actual playback speed = 2^(speed_parameter/64)

A speed parameter of 0 corresponds to unity speed playback (i.e. playback at "normal" speed). A speed parameter of -128 corresponds to playback at one fourth of normal speed, 127 corresponds to playback at almost four times the normal speed.

Return
The playback speed parameter

void (***set_playback_speed**)(int8_t speed)

Set Playback Speed.

See the *get_playback_speed()* function for an explanation of the playback speed parameter.

Note that the media player may not support all possible values of the playback speed parameter. If the value given is not supported, and is higher than the current value, the player should set the playback speed to the next higher supported value. (And correspondingly to the next lower supported value for given values lower than the current value.)

Param speed
The playback speed parameter to set

int8_t (*`get_seeking_speed`)(void)

Get Seeking Speed.

The seeking speed gives the speed with which the player is seeking. It is a factor, relative to real-time playback speed - a factor four means seeking happens at four times the real-time playback speed. Positive values are for forward seeking, negative values for backwards seeking.

The seeking speed is not settable - a non-zero seeking speed is the result of "fast rewind" of "fast forward" commands.

Return
The seeking speed factor

uint64_t (*`get_track_segments_id`)(void)

Read Current Track Segments Object ID.

Get an ID (48 bit) that can be used to retrieve the Current Track Segments Object from an Object Transfer Service

See the Media Control Service spec v1.0 sections 3.10 and 4.2 for a description of the Track Segments Object.

Return
Current The Track Segments Object ID

uint64_t (*`get_current_track_id`)(void)

Read Current Track Object ID.

Get an ID (48 bit) that can be used to retrieve the Current Track Object from an Object Transfer Service

See the Media Control Service spec v1.0 sections 3.11 and 4.3 for a description of the Current Track Object.

Return
The Current Track Object ID

void (*`set_current_track_id`)(uint64_t id)

Set Current Track Object ID.

Change the player's current track to the track given by the ID. (Behaves similarly to the goto track command.)

Param id
The ID of a track object

uint64_t (*`get_next_track_id`)(void)

Read Next Track Object ID.

Get an ID (48 bit) that can be used to retrieve the Next Track Object from an Object Transfer Service

Return
The Next Track Object ID

void (*set_next_track_id)(uint64_t id)

Set Next Track Object ID.

Change the player's next track to the track given by the ID.

Param id
The ID of a track object

uint64_t (*get_parent_group_id)(void)

Read Parent Group Object ID.

Get an ID (48 bit) that can be used to retrieve the Parent Track Object from an Object Transfer Service

The parent group is the parent of the current group.

See the Media Control Service spec v1.0 sections 3.14 and 4.4 for a description of the Current Track Object.

Return
The Current Group Object ID

uint64_t (*get_current_group_id)(void)

Read Current Group Object ID.

Get an ID (48 bit) that can be used to retrieve the Current Track Object from an Object Transfer Service

See the Media Control Service spec v1.0 sections 3.14 and 4.4 for a description of the Current Group Object.

Return
The Current Group Object ID

void (*set_current_group_id)(uint64_t id)

Set Current Group Object ID.

Change the player's current group to the group given by the ID, and the current track to the first track in that group.

Param id
The ID of a group object

uint8_t (*get_playing_order)(void)

Read Playing Order.

return The media player's current playing order

void (*set_playing_order)(uint8_t order)

Set Playing Order.

Set the media player's playing order. See the MEDIA_PROXY_PLAYING_ORDER_* defines.

Param order
The playing order to set

uint16_t (*get_playing_orders_supported)(void)

Read Playing Orders Supported.

Read a bitmap containing the media player's supported playing orders. See the MEDIA_PROXY_PLAYING_ORDERS_SUPPORTED_* defines.

Return
The media player's supported playing orders

uint8_t (*`get_media_state`)(void)

Read Media State.

Read the media player's state See the MEDIA_PROXY_MEDIA_STATE_* defines.

Return
The media player's state

void (*`send_command`)(const struct *mpl_cmd* *command)

Send Command.

Send a command to the media player. For command opcodes (play, pause, ...) - see the MEDIA_PROXY_OP_* defines.

Param command
The command to send

uint32_t (*`get_commands_supported`)(void)

Read Commands Supported.

Read a bitmap containing the media player's supported command opcodes. See the MEDIA_PROXY_OP_SUP_* defines.

Return
The media player's supported command opcodes

void (*`send_search`)(const struct *mpl_search* *search)

Set Search.

Write a search to the media player. (For the formatting of a search, see the Media Control Service spec and the mcs.h file.)

Param search
The search to write

uint64_t (*`get_search_results_id`)(void)

Read Search Results Object ID.

Get an ID (48 bit) that can be used to retrieve the Search Results Object from an Object Transfer Service

The search results object is a group object. The search results object only exists if a successful search operation has been done.

Return
The Search Results Object ID

uint8_t (*`get_content_ctrl_id`)(void)

Read Content Control ID.

The content control ID identifies a content control service on a device, and links it to the corresponding audio stream.

Return
The content control ID for the media player

Media Control Client

group `bt_gatt_mcc`

Bluetooth Media Control Client (MCC) interface.

Updated to the Media Control Profile specification revision 1.0

Since
3.0

Version
0.8.0

Typedefs

typedef void (*`bt_mcc_discover_mcs_cb`)(struct bt_conn *conn, int err)

Callback function for *bt_mcc_discover_mcs()*

Called when a media control server is discovered

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

typedef void (*`bt_mcc_read_player_name_cb`)(struct bt_conn *conn, int err, const char *name)

Callback function for *bt_mcc_read_player_name()*

Called when the player name is read or notified

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param name
Player name

typedef void (*`bt_mcc_read_icon_obj_id_cb`)(struct bt_conn *conn, int err, uint64_t icon_id)

Callback function for *bt_mcc_read_icon_obj_id()*

Called when the icon object ID is read

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param icon_id
The ID of the Icon Object. This is a UINT48 in a uint64_t

typedef void (*`bt_mcc_read_icon_url_cb`)(struct bt_conn *conn, int err, const char *icon_url)

Callback function for *bt_mcc_read_icon_url()*

Called when the icon URL is read

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param icon_url
The URL of the Icon

typedef void (*bt_mcc_track_changed_ntf_cb)(struct bt_conn *conn, int err)

Callback function for track changed notifications.

Called when a track change is notified.

The track changed characteristic is a special case. It can not be read or set, it can only be notified.

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

typedef void (*bt_mcc_read_track_title_cb)(struct bt_conn *conn, int err, const char *title)

Callback function for *bt_mcc_read_track_title()*

Called when the track title is read or notified

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param title
The title of the track

typedef void (*bt_mcc_read_track_duration_cb)(struct bt_conn *conn, int err, int32_t dur)

Callback function for *bt_mcc_read_track_duration()*

Called when the track duration is read or notified

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param dur
The duration of the track

typedef void (*bt_mcc_read_track_position_cb)(struct bt_conn *conn, int err, int32_t pos)

Callback function for *bt_mcc_read_track_position()*

Called when the track position is read or notified

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param pos
The Track Position

typedef void (*bt_mcc_set_track_position_cb)(struct bt_conn *conn, int err, int32_t pos)

Callback function for *bt_mcc_set_track_position()*

Called when the track position is set

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param pos
The Track Position set (or attempted to set)

typedef void (*`bt_mcc_read_playback_speed_cb`)(struct bt_conn *conn, int err, int8_t speed)

Callback function for *bt_mcc_read_playback_speed()*

Called when the playback speed is read or notified

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param speed
The Playback Speed

typedef void (*`bt_mcc_set_playback_speed_cb`)(struct bt_conn *conn, int err, int8_t speed)

Callback function for *bt_mcc_set_playback_speed()*

Called when the playback speed is set

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param speed
The Playback Speed set (or attempted to set)

typedef void (*`bt_mcc_read_seeking_speed_cb`)(struct bt_conn *conn, int err, int8_t speed)

Callback function for *bt_mcc_read_seeking_speed()*

Called when the seeking speed is read or notified

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param speed
The Seeking Speed

typedef void (*`bt_mcc_read_segments_obj_id_cb`)(struct bt_conn *conn, int err, uint64_t id)

Callback function for *bt_mcc_read_segments_obj_id()*

Called when the track segments object ID is read

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param id
The Track Segments Object ID (UINT48)

typedef void (*bt_mcc_read_current_track_obj_id_cb)(struct bt_conn *conn, int err, uint64_t id)

Callback function for *bt_mcc_read_current_track_obj_id()*

Called when the current track object ID is read or notified

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param id
The Current Track Object ID (UINT48)

typedef void (*bt_mcc_set_current_track_obj_id_cb)(struct bt_conn *conn, int err, uint64_t id)

Callback function for *bt_mcc_set_current_track_obj_id()*

Called when the current track object ID is set

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param id
The Object ID (UINT48) set (or attempted to set)

typedef void (*bt_mcc_read_next_track_obj_id_cb)(struct bt_conn *conn, int err, uint64_t id)

Callback function for bt_mcc_read_next_track_obj_id_obj()

Called when the next track object ID is read or notified

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param id
The Next Track Object ID (UINT48)

typedef void (*bt_mcc_set_next_track_obj_id_cb)(struct bt_conn *conn, int err, uint64_t id)

Callback function for *bt_mcc_set_next_track_obj_id()*

Called when the next track object ID is set

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param id
The Object ID (UINT48) set (or attempted to set)

typedef void (*bt_mcc_read_parent_group_obj_id_cb)(struct bt_conn *conn, int err, uint64_t id)

Callback function for *bt_mcc_read_parent_group_obj_id()*

Called when the parent group object ID is read or notified

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param id
The Parent Group Object ID (UINT48)

typedef void (*`bt_mcc_read_current_group_obj_id_cb`)(struct bt_conn *conn, int err, uint64_t id)

Callback function for *bt_mcc_read_current_group_obj_id()*

Called when the current group object ID is read or notified

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param id
The Current Group Object ID (UINT48)

typedef void (*`bt_mcc_set_current_group_obj_id_cb`)(struct bt_conn *conn, int err, uint64_t obj_id)

Callback function for *bt_mcc_set_current_group_obj_id()*

Called when the current group object ID is set

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param obj_id
The Object ID (UINT48) set (or attempted to set)

typedef void (*`bt_mcc_read_playing_order_cb`)(struct bt_conn *conn, int err, uint8_t order)

Callback function for *bt_mcc_read_playing_order()*

Called when the playing order is read or notified

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param order
The playback order

typedef void (*`bt_mcc_set_playing_order_cb`)(struct bt_conn *conn, int err, uint8_t order)

Callback function for *bt_mcc_set_playing_order()*

Called when the playing order is set

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param order
The Playing Order set (or attempted to set)

typedef void (*`bt_mcc_read_playing_orders_supported_cb`)(struct bt_conn *conn, int err, uint16_t orders)

Callback function for *bt_mcc_read_playing_orders_supported()*

Called when the supported playing orders are read or notified

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param orders
The playing orders supported (bitmap)

typedef void (*`bt_mcc_read_media_state_cb`)(struct bt_conn *conn, int err, uint8_t state)

Callback function for *bt_mcc_read_media_state()*

Called when the media state is read or notified

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param state
The Media State

typedef void (*`bt_mcc_send_cmd_cb`)(struct bt_conn *conn, int err, const struct *mpl_cmd* *cmd)

Callback function for *bt_mcc_send_cmd()*

Called when a command is sent, i.e. when the media control point is set

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param cmd
The command sent

typedef void (*`bt_mcc_cmd_ntf_cb`)(struct bt_conn *conn, int err, const struct *mpl_cmd_ntf* *ntf)

Callback function for command notifications.

Called when the media control point is notified

Notifications for commands (i.e. for writes to the media control point) use a different parameter structure than what is used for sending commands (writing to the media control point)

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param ntf
The command notification

typedef void (*`bt_mcc_read_opcodes_supported_cb`)(struct bt_conn *conn, int err, uint32_t opcodes)

Callback function for *bt_mcc_read_opcodes_supported()*

Called when the supported opcodes (commands) are read or notified

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param opcodes
The supported opcodes

typedef void (*`bt_mcc_send_search_cb`)(struct bt_conn *conn, int err, const struct *mpl_search* *search)

Callback function for *bt_mcc_send_search()*

Called when a search is sent, i.e. when the search control point is set

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param search
The search set (or attempted to set)

typedef void (*`bt_mcc_search_ntf_cb`)(struct bt_conn *conn, int err, uint8_t result_code)

Callback function for search notifications.

Called when the search control point is notified

Notifications for searches (i.e. for writes to the search control point) use a different parameter structure than what is used for sending searches (writing to the search control point)

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param result_code
The search notification

typedef void (*`bt_mcc_read_search_results_obj_id_cb`)(struct bt_conn *conn, int err, uint64_t id)

Callback function for *bt_mcc_read_search_results_obj_id()*

Called when the search results object ID is read or notified

Note that the Search Results Object ID value may be zero, in case the characteristic does not exist on the server. (This will be the case if there has not been a successful search.)

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param id
The Search Results Object ID (UINT48)

typedef void (*`bt_mcc_read_content_control_id_cb`)(struct bt_conn *conn, int err, uint8_t ccid)

Callback function for *bt_mcc_read_content_control_id()*

Called when the content control ID is read

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param ccid
The Content Control ID

typedef void (*`bt_mcc_otc_obj_selected_cb`)(struct bt_conn *conn, int err)

Callback function for object selected.

Called when an object is selected

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

typedef void (*`bt_mcc_otc_obj_metadata_cb`)(struct bt_conn *conn, int err)

Callback function for *bt_mcc_otc_read_object_metadata()*

Called when object metadata is read

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

typedef void (*`bt_mcc_otc_read_icon_object_cb`)(struct bt_conn *conn, int err, struct *net_buf_simple* *buf)

Callback function for *bt_mcc_otc_read_icon_object()*

Called when the icon object is read

If err is EMSGSIZE, the object contents have been truncated.

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param buf
Buffer containing the object contents

typedef void (*bt_mcc_otc_read_track_segments_object_cb)(struct bt_conn *conn, int err, struct *net_buf_simple* *buf)

Callback function for *bt_mcc_otc_read_track_segments_object()*

Called when the track segments object is read

If err is EMSGSIZE, the object contents have been truncated.

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param buf
Buffer containing the object contents

typedef void (*bt_mcc_otc_read_current_track_object_cb)(struct bt_conn *conn, int err, struct *net_buf_simple* *buf)

Callback function for *bt_mcc_otc_read_current_track_object()*

Called when the current track object is read

If err is EMSGSIZE, the object contents have been truncated.

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param buf
Buffer containing the object contents

typedef void (*bt_mcc_otc_read_next_track_object_cb)(struct bt_conn *conn, int err, struct *net_buf_simple* *buf)

Callback function for *bt_mcc_otc_read_next_track_object()*

Called when the next track object is read

If err is EMSGSIZE, the object contents have been truncated.

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param buf
Buffer containing the object contents

typedef void (*bt_mcc_otc_read_parent_group_object_cb)(struct bt_conn *conn, int err, struct *net_buf_simple* *buf)

Callback function for *bt_mcc_otc_read_parent_group_object()*

Called when the parent group object is read

If err is EMSGSIZE, the object contents have been truncated.

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param buf
Buffer containing the object contents

typedef void (*`bt_mcc_otc_read_current_group_object_cb`)(struct bt_conn *conn, int err, struct *net_buf_simple* *buf)

Callback function for *bt_mcc_otc_read_current_group_object()*

Called when the current group object is read

If err is EMSGSIZE, the object contents have been truncated.

Param conn
The connection that was used to initialise the media control client

Param err
Error value. 0 on success, GATT error or errno on fail

Param buf
Buffer containing the object contents

Functions

int `bt_mcc_init`(struct *bt_mcc_cb* *cb)

Initialize Media Control Client.

Parameters

- `cb` – Callbacks to be used

Returns
0 if success, errno on failure.

int `bt_mcc_discover_mcs`(struct bt_conn *conn, bool subscribe)

Discover Media Control Service.

Discover Media Control Service (MCS) on the server given by the connection Optionally subscribe to notifications.

Shall be called once, after media control client initialization and before using other media control client functionality.

Parameters

- `conn` – Connection to the peer device
- `subscribe` – Whether to subscribe to notifications

Returns
0 if success, errno on failure.

int `bt_mcc_read_player_name`(struct bt_conn *conn)

Read Media Player Name.

Parameters

- `conn` – Connection to the peer device

Returns
0 if success, errno on failure.

int bt_mcc_read_icon_obj_id(struct bt_conn *conn)
Read Icon Object ID.

Parameters

- conn – Connection to the peer device

Returns
0 if success, errno on failure.

int bt_mcc_read_icon_url(struct bt_conn *conn)
Read Icon Object URL.

Parameters

- conn – Connection to the peer device

Returns
0 if success, errno on failure.

int bt_mcc_read_track_title(struct bt_conn *conn)
Read Track Title.

Parameters

- conn – Connection to the peer device

Returns
0 if success, errno on failure.

int bt_mcc_read_track_duration(struct bt_conn *conn)
Read Track Duration.

Parameters

- conn – Connection to the peer device

Returns
0 if success, errno on failure.

int bt_mcc_read_track_position(struct bt_conn *conn)
Read Track Position.

Parameters

- conn – Connection to the peer device

Returns
0 if success, errno on failure.

int bt_mcc_set_track_position(struct bt_conn *conn, int32_t pos)
Set Track position.

Parameters

- conn – Connection to the peer device
- pos – Track position

Returns
0 if success, errno on failure.

int bt_mcc_read_playback_speed(struct bt_conn *conn)
Read Playback speed.

Parameters

- conn – Connection to the peer device

Returns
0 if success, errno on failure.

int **bt_mcc_set_playback_speed**(struct bt_conn *conn, int8_t speed)

Set Playback Speed.

Parameters

- **conn** – Connection to the peer device
- **speed** – Playback speed

Returns

0 if success, errno on failure.

int **bt_mcc_read_seeking_speed**(struct bt_conn *conn)

Read Seeking speed.

Parameters

- **conn** – Connection to the peer device

Returns

0 if success, errno on failure.

int **bt_mcc_read_segments_obj_id**(struct bt_conn *conn)

Read Track Segments Object ID.

Parameters

- **conn** – Connection to the peer device

Returns

0 if success, errno on failure.

int **bt_mcc_read_current_track_obj_id**(struct bt_conn *conn)

Read Current Track Object ID.

Parameters

- **conn** – Connection to the peer device

Returns

0 if success, errno on failure.

int **bt_mcc_set_current_track_obj_id**(struct bt_conn *conn, uint64_t id)

Set Current Track Object ID.

Set the Current Track to the track given by the `id` parameter

Parameters

- **conn** – Connection to the peer device
- **id** – Object Transfer Service ID (UINT48) of the track to set as the current track

Returns

0 if success, errno on failure.

int **bt_mcc_read_next_track_obj_id**(struct bt_conn *conn)

Read Next Track Object ID.

Parameters

- **conn** – Connection to the peer device

Returns

0 if success, errno on failure.

int **bt_mcc_set_next_track_obj_id**(struct bt_conn *conn, uint64_t id)

Set Next Track Object ID.

Set the Next Track to the track given by the `id` parameter

Parameters

- **conn** – Connection to the peer device
- **id** – Object Transfer Service ID (UINT48) of the track to set as the next track

Returns

0 if success, errno on failure.

int **bt_mcc_read_current_group_obj_id**(struct bt_conn *conn)

Read Current Group Object ID.

Parameters

- **conn** – Connection to the peer device

Returns

0 if success, errno on failure.

int **bt_mcc_set_current_group_obj_id**(struct bt_conn *conn, uint64_t id)

Set Current Group Object ID.

Set the Current Group to the group given by the `id` parameter

Parameters

- **conn** – Connection to the peer device
- **id** – Object Transfer Service ID (UINT48) of the group to set as the current group

Returns

0 if success, errno on failure.

int **bt_mcc_read_parent_group_obj_id**(struct bt_conn *conn)

Read Parent Group Object ID.

Parameters

- **conn** – Connection to the peer device

Returns

0 if success, errno on failure.

int **bt_mcc_read_playing_order**(struct bt_conn *conn)

Read Playing Order.

Parameters

- **conn** – Connection to the peer device

Returns

0 if success, errno on failure.

int **bt_mcc_set_playing_order**(struct bt_conn *conn, uint8_t order)

Set Playing Order.

Parameters

- **conn** – Connection to the peer device
- **order** – Playing order

Returns

0 if success, errno on failure.

int **bt_mcc_read_playing_orders_supported**(struct bt_conn *conn)

Read Playing Orders Supported.

Parameters

- **conn** – Connection to the peer device

Returns

0 if success, errno on failure.

int **bt_mcc_read_media_state**(struct bt_conn *conn)

Read Media State.

Parameters

- **conn** – Connection to the peer device

Returns

0 if success, errno on failure.

int **bt_mcc_send_cmd**(struct bt_conn *conn, const struct *mpl_cmd* *cmd)

Send a command.

Write a command (e.g. “play”, “pause”) to the server’s media control point.

Parameters

- **conn** – Connection to the peer device
- **cmd** – The command to send

Returns

0 if success, errno on failure.

int **bt_mcc_read_opcodes_supported**(struct bt_conn *conn)

Read Opcodes Supported.

Parameters

- **conn** – Connection to the peer device

Returns

0 if success, errno on failure.

int **bt_mcc_send_search**(struct bt_conn *conn, const struct *mpl_search* *search)

Send a Search command.

Write a search to the server’s search control point.

Parameters

- **conn** – Connection to the peer device
- **search** – The search

Returns

0 if success, errno on failure.

int **bt_mcc_read_search_results_obj_id**(struct bt_conn *conn)

Search Results Group Object ID.

Parameters

- **conn** – Connection to the peer device

Returns

0 if success, errno on failure.

int bt_mcc_read_content_control_id(struct bt_conn *conn)

Read Content Control ID.

Parameters

- conn – Connection to the peer device

Returns

0 if success, errno on failure.

int bt_mcc_otc_read_object_metadata(struct bt_conn *conn)

Read the current object metadata.

Parameters

- conn – Connection to the peer device

Returns

0 if success, errno on failure.

int bt_mcc_otc_read_icon_object(struct bt_conn *conn)

Read the Icon Object.

Parameters

- conn – Connection to the peer device

Returns

0 if success, errno on failure.

int bt_mcc_otc_read_track_segments_object(struct bt_conn *conn)

Read the Track Segments Object.

Parameters

- conn – Connection to the peer device

Returns

0 if success, errno on failure.

int bt_mcc_otc_read_current_track_object(struct bt_conn *conn)

Read the Current Track Object.

Parameters

- conn – Connection to the peer device

Returns

0 if success, errno on failure.

int bt_mcc_otc_read_next_track_object(struct bt_conn *conn)

Read the Next Track Object.

Parameters

- conn – Connection to the peer device

Returns

0 if success, errno on failure.

int bt_mcc_otc_read_current_group_object(struct bt_conn *conn)

Read the Current Group Object.

Parameters

- conn – Connection to the peer device

Returns

0 if success, errno on failure.

int **bt_mcc_otc_read_parent_group_object**(struct bt_conn *conn)
Read the Parent Group Object.

Parameters

- **conn** – Connection to the peer device

Returns
0 if success, errno on failure.

struct *bt_ots_client* ***bt_mcc_otc_inst**(struct bt_conn *conn)
Look up MCC OTC instance.

Parameters

- **conn** – The connection to the MCC server.

Returns
Pointer to a MCC OTC instance if found else NULL.

struct **bt_mcc_cb**
#include <mcc.h> Media control client callbacks.

Public Members

bt_mcc_discover_mcs_cb **discover_mcs**
Callback when discovery has finished.

bt_mcc_read_player_name_cb **read_player_name**
Callback when reading the player name.

bt_mcc_read_icon_obj_id_cb **read_icon_obj_id**
Callback when reading the icon object ID.

bt_mcc_read_icon_url_cb **read_icon_url**
Callback when reading the icon URL.

bt_mcc_track_changed_ntf_cb **track_changed_ntf**
Callback when getting a track changed notification.

bt_mcc_read_track_title_cb **read_track_title**
Callback when reading the track title.

bt_mcc_read_track_duration_cb **read_track_duration**
Callback when reading the track duration.

bt_mcc_read_track_position_cb **read_track_position**
Callback when reading the track position.

bt_mcc_set_track_position_cb **set_track_position**
Callback when setting the track position.

bt_mcc_read_playback_speed_cb **read_playback_speed**
Callback when reading the playback speed.

bt_mcc_set_playback_speed_cb `set_playback_speed`
Callback when setting the playback speed.

bt_mcc_read_seeking_speed_cb `read_seeking_speed`
Callback when reading the seeking speed.

bt_mcc_read_segments_obj_id_cb `read_segments_obj_id`
Callback when reading the segments object ID.

bt_mcc_read_current_track_obj_id_cb `read_current_track_obj_id`
Callback when reading the current track object ID.

bt_mcc_set_current_track_obj_id_cb `set_current_track_obj_id`
Callback when setting the current track object ID.

bt_mcc_read_next_track_obj_id_cb `read_next_track_obj_id`
Callback when reading the next track object ID.

bt_mcc_set_next_track_obj_id_cb `set_next_track_obj_id`
Callback when setting the next track object ID.

bt_mcc_read_current_group_obj_id_cb `read_current_group_obj_id`
Callback when reading the current group object ID.

bt_mcc_set_current_group_obj_id_cb `set_current_group_obj_id`
Callback when setting the current group object ID.

bt_mcc_read_parent_group_obj_id_cb `read_parent_group_obj_id`
Callback when reading the parent group object ID.

bt_mcc_read_playing_order_cb `read_playing_order`
Callback when reading the playing order.

bt_mcc_set_playing_order_cb `set_playing_order`
Callback when setting the playing order.

bt_mcc_read_playing_orders_supported_cb `read_playing_orders_supported`
Callback when reading the supported playing orders.

bt_mcc_read_media_state_cb `read_media_state`
Callback when reading the media state.

bt_mcc_send_cmd_cb `send_cmd`
Callback when sending a command.

bt_mcc_cmd_ntf_cb `cmd_ntf`
Callback command notifications.

bt_mcc_read_opcodes_supported_cb `read_opcodes_supported`
Callback when reading the supported opcodes.

bt_mcc_send_search_cb `send_search`
Callback when sending the a search query.

bt_mcc_search_ntf_cb `search_ntf`
Callback when receiving a search notification.

bt_mcc_read_search_results_obj_id_cb `read_search_results_obj_id`
Callback when reading the search results object ID.

bt_mcc_read_content_control_id_cb `read_content_control_id`
Callback when reading the content control ID.

bt_mcc_otc_obj_selected_cb `otc_obj_selected`
Callback when selecting an object.

bt_mcc_otc_obj_metadata_cb `otc_obj_metadata`
Callback when receiving the current object metadata.

bt_mcc_otc_read_icon_object_cb `otc_icon_object`
Callback when reading the current icon object.

bt_mcc_otc_read_track_segments_object_cb `otc_track_segments_object`
Callback when reading the track segments object.

bt_mcc_otc_read_current_track_object_cb `otc_current_track_object`
Callback when reading the current track object.

bt_mcc_otc_read_next_track_object_cb `otc_next_track_object`
Callback when reading the next track object.

bt_mcc_otc_read_current_group_object_cb `otc_current_group_object`
Callback when reading the current group object.

bt_mcc_otc_read_parent_group_object_cb `otc_parent_group_object`
Callback when reading the parent group object.

Bluetooth Microphone Control

API Reference

group `bt_gatt_micp`
Microphone Control Profile (MICP)

Since
2.7

Version
0.8.0

Application error codes

`BT_MICP_ERR_MUTE_DISABLED`
Mute/unmute commands are disabled.

Microphone Control Profile mute states

`BT_MICP_MUTE_UNMUTED`
The microphone state is unmuted.

`BT_MICP_MUTE_MUTED`
The microphone state is muted.

`BT_MICP_MUTE_DISABLED`
The microphone state is disabled and cannot be muted or unmuted.

Defines

`BT_MICP_MIC_DEV_AICS_CNT`
Defines the maximum number of Microphone Control Service instances for the Microphone Control Profile Microphone Device.

Functions

int **bt_micp_mic_dev_register**(struct *bt_micp_mic_dev_register_param* *param)
Initialize the Microphone Control Profile Microphone Device.

This will enable the Microphone Control Service instance and make it discoverable by Microphone Controllers.

Parameters

- `param` – Pointer to an initialization structure.

Returns
0 if success, errno on failure.

int **bt_micp_mic_dev_included_get**(struct *bt_micp_included* *included)
Get Microphone Device included services.

Returns a pointer to a struct that contains information about the Microphone Device included Audio Input Control Service instances.

Requires that `CONFIG_BT_MICP_MIC_DEV_AICS` is enabled.

Parameters

- `included` – Pointer to store the result in.

Returns
0 if success, errno on failure.

int **bt_micp_mic_dev_unmute**(void)

Unmute the Microphone Device.

Returns
0 on success, GATT error value on fail.

int **bt_micp_mic_dev_mute**(void)

Mute the Microphone Device.

Returns
0 on success, GATT error value on fail.

int **bt_micp_mic_dev_mute_disable**(void)

Disable the mute functionality on the Microphone Device.

Can be reenabled by called *bt_micp_mic_dev_mute* or *bt_micp_mic_dev_unmute*.

Returns
0 on success, GATT error value on fail.

int **bt_micp_mic_dev_mute_get**(void)

Read the mute state on the Microphone Device.

Returns
0 on success, GATT error value on fail.

int **bt_micp_mic_ctlr_included_get**(struct bt_micp_mic_ctlr *mic_ctlr, struct *bt_micp_included* *included)

Get Microphone Control Profile included services.

Returns a pointer to a struct that contains information about the Microphone Control Profile included services instances, such as pointers to the Audio Input Control Service instances.

Requires that `CONFIG_BT_MICP_MIC_CTLR_AICS` is enabled.

Parameters

- **mic_ctlr** – Microphone Controller instance pointer.
- **included** – **[out]** Pointer to store the result in.

Returns
0 if success, errno on failure.

int **bt_micp_mic_ctlr_conn_get**(const struct bt_micp_mic_ctlr *mic_ctlr, struct bt_conn **conn)

Get the connection pointer of a Microphone Controller instance.

Get the Bluetooth connection pointer of a Microphone Controller instance.

Parameters

- **mic_ctlr** – Microphone Controller instance pointer.
- **conn** – Connection pointer.

Returns
0 if success, errno on failure.

struct bt_micp_mic_ctlr ***bt_micp_mic_ctlr_get_by_conn**(const struct bt_conn *conn)

Get the volume controller from a connection pointer.

Get the Volume Control Profile Volume Controller pointer from a connection pointer. Only volume controllers that have been initiated via *bt_micp_mic_ctlr_discover()* can be retrieved.

Parameters

- **conn** – Connection pointer.

Return values

- **Pointer** – to a Microphone Control Profile Microphone Controller instance
- **NULL** – if conn is NULL or if the connection has not done discovery yet

int **bt_micp_mic_ctlr_discover**(struct bt_conn *conn, struct bt_micp_mic_ctlr **mic_ctlr)

Discover Microphone Control Service.

This will start a GATT discovery and setup handles and subscriptions. This shall be called once before any other actions can be executed for the peer device, and the *bt_micp_mic_ctlr_cb::discover* callback will notify when it is possible to start remote operations.

-

Parameters

- **conn** – The connection to initialize the profile for.
- **mic_ctlr** – **[out]** Valid remote instance object on success.

Returns

0 on success, GATT error value on fail.

int **bt_micp_mic_ctlr_unmute**(struct bt_micp_mic_ctlr *mic_ctlr)

Unmute a remote Microphone Device.

Parameters

- **mic_ctlr** – Microphone Controller instance pointer.

Returns

0 on success, GATT error value on fail.

int **bt_micp_mic_ctlr_mute**(struct bt_micp_mic_ctlr *mic_ctlr)

Mute a remote Microphone Device.

Parameters

- **mic_ctlr** – Microphone Controller instance pointer.

Returns

0 on success, GATT error value on fail.

int **bt_micp_mic_ctlr_mute_get**(struct bt_micp_mic_ctlr *mic_ctlr)

Read the mute state of a remote Microphone Device.

Parameters

- **mic_ctlr** – Microphone Controller instance pointer.

Returns

0 on success, GATT error value on fail.

int **bt_micp_mic_ctlr_cb_register**(struct *bt_micp_mic_ctlr_cb* *cb)

Registers the callbacks used by Microphone Controller.

This can only be done as the client.

Parameters

- **cb** – The callback structure.

Returns

0 if success, errno on failure.

struct **bt_micp_mic_dev_register_param**

#include <micp.h> Register parameters structure for Microphone Control Service.

Public Members

struct bt_aics_register_param **aics_param**[0]

Register parameter structure for Audio Input Control Services.

struct *bt_micp_mic_dev_cb* ***cb**

Microphone Control Profile callback structure.

struct **bt_micp_included**

#include <micp.h> Microphone Control Profile included services.

Used for to represent the Microphone Control Profile included service instances, for either a Microphone Controller or a Microphone Device. The instance pointers either represent local service instances, or remote service instances.

Public Members

uint8_t **aics_cnt**

Number of Audio Input Control Service instances.

struct bt_aics ****aics**

Array of pointers to Audio Input Control Service instances.

struct **bt_micp_mic_dev_cb**

#include <micp.h> Struct to hold the Microphone Device callbacks.

These can be registered for usage with *bt_micp_mic_dev_register()*.

Public Members

void (***mute**)(uint8_t mute)

Callback function for Microphone Device mute.

Called when the value is read with *bt_micp_mic_dev_mute_get()*, or if the value is changed by either the Microphone Device or a Microphone Controller.

Param mute
The mute setting of the Microphone Control Service.

struct **bt_micp_mic_ctlr_cb**

#include <micp.h> Struct to hold the Microphone Controller callbacks.

These can be registered for usage with *bt_micp_mic_ctlr_cb_register()*.

Public Members

void (*`mute`)(struct bt_micp_mic_ctlr *mic_ctlr, int err, uint8_t mute)

Callback function for Microphone Control Profile mute.

Called when the value is read, or if the value is changed by either the Microphone Device or a Microphone Controller.

Param mic_ctlr
Microphone Controller instance pointer.

Param err
Error value. 0 on success, GATT error or errno on fail. For notifications, this will always be 0.

Param mute
The mute setting of the Microphone Control Service.

void (*`discover`)(struct bt_micp_mic_ctlr *mic_ctlr, int err, uint8_t aics_count)

Callback function for *bt_micp_mic_ctlr_discover()*.

Param mic_ctlr
Microphone Controller instance pointer.

Param err
Error value. 0 on success, GATT error or errno on fail.

Param aics_count
Number of Audio Input Control Service instances on peer device.

void (*`mute_written`)(struct bt_micp_mic_ctlr *mic_ctlr, int err)

Callback function for Microphone Control Profile mute/unmute.

Param mic_ctlr
Microphone Controller instance pointer.

Param err
Error value. 0 on success, GATT error or errno on fail.

void (*`unmute_written`)(struct bt_micp_mic_ctlr *mic_ctlr, int err)

Callback function for Microphone Control Profile mute/unmute.

Param mic_ctlr
Microphone Controller instance pointer.

Param err
Error value. 0 on success, GATT error or errno on fail.

Bluetooth Audio Volume Control

API Reference

group `bt_gatt_vcp`

Volume Control Profile (VCP)

The Volume Control Profile (VCP) provides procedures to control the volume level and mute state on audio devices.

Since
2.7

Version
0.8.0

Volume Control Service Error codes

`BT_VCP_ERR_INVALID_COUNTER`
: The Change_Counter operand value does not match the Change_Counter field value of the Volume State characteristic.

`BT_VCP_ERR_OP_NOT_SUPPORTED`
: An invalid opcode has been used in a control point procedure.

Volume Control Service Mute Values

`BT_VCP_STATE_UNMUTED`
: The volume state is unmuted.

`BT_VCP_STATE_MUTED`
: The volume state is muted.

Defines

`BT_VCP_VOL_REND_VOCS_CNT`
: Defines the maximum number of Volume Offset Control service instances for the Volume Control Profile Volume Renderer.

`BT_VCP_VOL_REND_AICS_CNT`
: Defines the maximum number of Audio Input Control service instances for the Volume Control Profile Volume Renderer.

Functions

int **bt_vcp_vol_rend_included_get**(struct *bt_vcp_included* *included)
: Get Volume Control Service included services.

  Returns a pointer to a struct that contains information about the Volume Control Service included service instances, such as pointers to the Volume Offset Control Service (Volume Offset Control Service) or Audio Input Control Service (AICS) instances.

  Parameters

  - `included` – **[out]** Pointer to store the result in.

  Returns
  0 if success, errno on failure.

int **bt_vcp_vol_rend_register**(struct *bt_vcp_vol_rend_register_param* *param)
: Register the Volume Control Service.

  This will register and enable the service and make it discoverable by clients.

  Parameters

  - `param` – Volume Control Service register parameters.

  Returns
  0 if success, errno on failure.

int **bt_vcp_vol_rend_set_step**(uint8_t volume_step)

Set the Volume Control Service volume step size.

Set the value that the volume changes, when changed relatively with e.g. *bt_vcp_vol_rend_vol_down* or *bt_vcp_vol_rend_vol_up*.

This can only be done as the server.

Parameters

- **volume_step** – The volume step size (1-255).

Returns

0 if success, errno on failure.

int **bt_vcp_vol_rend_get_state**(void)

Get the Volume Control Service volume state.

Returns

0 if success, errno on failure.

int **bt_vcp_vol_rend_get_flags**(void)

Get the Volume Control Service flags.

Returns

0 if success, errno on failure.

int **bt_vcp_vol_rend_vol_down**(void)

Turn the volume down by one step on the server.

Returns

0 if success, errno on failure.

int **bt_vcp_vol_rend_vol_up**(void)

Turn the volume up by one step on the server.

Returns

0 if success, errno on failure.

int **bt_vcp_vol_rend_unmute_vol_down**(void)

Turn the volume down and unmute the server.

Returns

0 if success, errno on failure.

int **bt_vcp_vol_rend_unmute_vol_up**(void)

Turn the volume up and unmute the server.

Returns

0 if success, errno on failure.

int **bt_vcp_vol_rend_set_vol**(uint8_t volume)

Set the volume on the server.

Parameters

- **volume** – The absolute volume to set.

Returns

0 if success, errno on failure.

int **bt_vcp_vol_rend_unmute**(void)

Unmute the server.

Returns

0 if success, errno on failure.

int **bt_vcp_vol_rend_mute**(void)

Mute the server.

Returns

0 if success, errno on failure.

int **bt_vcp_vol_ctlr_cb_register**(struct *bt_vcp_vol_ctlr_cb* *cb)

Registers the callbacks used by the Volume Controller.

Parameters

- **cb** – The callback structure.

Return values

- **0** – on success
- **-EINVAL** – if cb is NULL
- **-EALREADY** – if cb was already registered

int **bt_vcp_vol_ctlr_cb_unregister**(struct *bt_vcp_vol_ctlr_cb* *cb)

Unregisters the callbacks used by the Volume Controller.

Parameters

- **cb** – The callback structure.

Return values

- **0** – on success
- **-EINVAL** – if cb is NULL
- **-EALREADY** – if cb was not registered

int **bt_vcp_vol_ctlr_discover**(struct bt_conn *conn, struct bt_vcp_vol_ctlr **vol_ctlr)

Discover Volume Control Service and included services.

This will start a GATT discovery and setup handles and subscriptions. This shall be called once before any other actions can be executed for the peer device, and the *bt_vcp_vol_ctlr_cb::discover* callback will notify when it is possible to start remote operations.

This shall only be done as the client,

Parameters

- **conn** – The connection to discover Volume Control Service for.
- **vol_ctlr** – **[out]** Valid remote instance object on success.

Returns

0 if success, errno on failure.

struct bt_vcp_vol_ctlr ***bt_vcp_vol_ctlr_get_by_conn**(const struct bt_conn *conn)

Get the volume controller from a connection pointer.

Get the Volume Control Profile Volume Controller pointer from a connection pointer. Only volume controllers that have been initiated via *bt_vcp_vol_ctlr_discover()* can be retrieved.

Parameters

- **conn** – Connection pointer.

Return values

- **Pointer** – to a Volume Control Profile Volume Controller instance
- **NULL** – if conn is NULL or if the connection has not done discovery yet

int **bt_vcp_vol_ctlr_conn_get**(const struct bt_vcp_vol_ctlr *vol_ctlr, struct bt_conn **conn)

Get the connection pointer of a client instance.

Get the Bluetooth connection pointer of a Volume Control Service client instance.

Parameters

- **vol_ctlr** – Volume Controller instance pointer.
- **conn** – **[out]** Connection pointer.

Returns
0 if success, errno on failure.

int **bt_vcp_vol_ctlr_included_get**(struct bt_vcp_vol_ctlr *vol_ctlr, struct *bt_vcp_included* *included)

Get Volume Control Service included services.

Returns a pointer to a struct that contains information about the Volume Control Service included service instances, such as pointers to the Volume Offset Control Service (Volume Offset Control Service) or Audio Input Control Service (AICS) instances.

Requires that CONFIG_BT_VCP_VOL_CTLR_VOCS or CONFIG_BT_VCP_VOL_CTLR_AICS is enabled.

Parameters

- **vol_ctlr** – Volume Controller instance pointer.
- **included** – **[out]** Pointer to store the result in.

Returns
0 if success, errno on failure.

int **bt_vcp_vol_ctlr_read_state**(struct bt_vcp_vol_ctlr *vol_ctlr)

Read the volume state of a remote Volume Renderer.

Parameters

- **vol_ctlr** – Volume Controller instance pointer.

Returns
0 if success, errno on failure.

int **bt_vcp_vol_ctlr_read_flags**(struct bt_vcp_vol_ctlr *vol_ctlr)

Read the volume flags of a remote Volume Renderer.

Parameters

- **vol_ctlr** – Volume Controller instance pointer.

Returns
0 if success, errno on failure.

int **bt_vcp_vol_ctlr_vol_down**(struct bt_vcp_vol_ctlr *vol_ctlr)

Turn the volume down one step on a remote Volume Renderer.

Parameters

- **vol_ctlr** – Volume Controller instance pointer.

Returns
0 if success, errno on failure.

int **bt_vcp_vol_ctlr_vol_up**(struct bt_vcp_vol_ctlr *vol_ctlr)

Turn the volume up one step on a remote Volume Renderer.

Parameters

- **vol_ctlr** – Volume Controller instance pointer.

Returns
0 if success, errno on failure.

int **bt_vcp_vol_ctlr_unmute_vol_down**(struct bt_vcp_vol_ctlr *vol_ctlr)
Turn the volume down one step and unmute on a remote Volume Renderer.

Parameters

- **vol_ctlr** – Volume Controller instance pointer.

Returns
0 if success, errno on failure.

int **bt_vcp_vol_ctlr_unmute_vol_up**(struct bt_vcp_vol_ctlr *vol_ctlr)
Turn the volume up one step and unmute on a remote Volume Renderer.

Parameters

- **vol_ctlr** – Volume Controller instance pointer.

Returns
0 if success, errno on failure.

int **bt_vcp_vol_ctlr_set_vol**(struct bt_vcp_vol_ctlr *vol_ctlr, uint8_t volume)
Set the absolute volume on a remote Volume Renderer.

Parameters

- **vol_ctlr** – Volume Controller instance pointer.
- **volume** – The absolute volume to set.

Returns
0 if success, errno on failure.

int **bt_vcp_vol_ctlr_unmute**(struct bt_vcp_vol_ctlr *vol_ctlr)
Unmute a remote Volume Renderer.

Parameters

- **vol_ctlr** – Volume Controller instance pointer.

Returns
0 if success, errno on failure.

int **bt_vcp_vol_ctlr_mute**(struct bt_vcp_vol_ctlr *vol_ctlr)
Mute a remote Volume Renderer.

Parameters

- **vol_ctlr** – Volume Controller instance pointer.

Returns
0 if success, errno on failure.

struct **bt_vcp_vol_rend_register_param**
#include <vcp.h> Register structure for Volume Control Service.

Public Members

uint8_t **step**
Initial step size (1-255)

uint8_t **mute**
: Initial mute state (0-1)

uint8_t **volume**
: Initial volume level (0-255)

struct bt_vocs_register_param **vocs_param**[0]
: Register parameters for Volume Offset Control Services.

struct bt_aics_register_param **aics_param**[0]
: Register parameters for Audio Input Control Services.

struct *bt_vcp_vol_rend_cb* ***cb**
: Volume Control Service callback structure.

struct **bt_vcp_included**

#include <*vcp.h*> Volume Control Service included services.

Used for to represent the Volume Control Service included service instances, for either a client or a server. The instance pointers either represent local server instances, or remote service instances.

Public Members

uint8_t **vocs_cnt**
: Number of Volume Offset Control Service instances.

struct bt_vocs ****vocs**
: Array of pointers to Volume Offset Control Service instances.

uint8_t **aics_cnt**
: Number of Audio Input Control Service instances.

struct bt_aics ****aics**
: Array of pointers to Audio Input Control Service instances.

struct **bt_vcp_vol_rend_cb**

#include <*vcp.h*> Struct to hold the Volume Renderer callbacks.

These can be registered for usage with *bt_vcp_vol_rend_register()*.

Public Members

void (***state**)(int err, uint8_t volume, uint8_t mute)
: Callback function for Volume Control Service volume state.

  Called when the value is locally read with *bt_vcp_vol_rend_get_state()*, or if the state is changed by either the Volume Renderer or a remote Volume Controller.

Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.
Param volume
The volume of the Volume Control Service server.
Param mute
The mute setting of the Volume Control Service server.

void (*`flags`)(int err, uint8_t flags)

Callback function for Volume Control Service flags.

Called when the value is locally read as the server. Called when the value is remotely read as the client. Called if the value is changed by either the server or client.
Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.
Param flags
The flags of the Volume Control Service server.

struct `bt_vcp_vol_ctlr_cb`

#include <*vcp.h*> Struct to hold the Volume Controller callbacks.

These can be registered for usage with *bt_vcp_vol_ctlr_cb_register()*.

Public Members

void (*`state`)(struct bt_vcp_vol_ctlr *vol_ctlr, int err, uint8_t volume, uint8_t mute)

Callback function for Volume Control Profile volume state.

Called when the value is remotely read as the Volume Controller. Called if the value is changed by either the Volume Renderer or Volume Controller, and notified to the to Volume Controller.
Param vol_ctlr
Volume Controller instance pointer.
Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.
Param volume
The volume of the Volume Renderer.
Param mute
The mute setting of the Volume Renderer.

void (*`flags`)(struct bt_vcp_vol_ctlr *vol_ctlr, int err, uint8_t flags)

Callback function for Volume Control Profile volume flags.

Called when the value is remotely read as the Volume Controller. Called if the value is changed by the Volume Renderer.

A non-zero value indicates the volume has been changed on the Volume Renderer since it was booted.
Param vol_ctlr
Volume Controller instance pointer.
Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.

Param flags
The flags of the Volume Renderer.

void (*`discover`)(struct bt_vcp_vol_ctlr *vol_ctlr, int err, uint8_t vocs_count, uint8_t aics_count)

Callback function for *bt_vcp_vol_ctlr_discover()*.

This callback is called once the discovery procedure is completed.

Param vol_ctlr
Volume Controller instance pointer.

Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.

Param vocs_count
Number of Volume Offset Control Service instances on the remote Volume Renderer.

Param aics_count
Number of Audio Input Control Service instances the remote Volume Renderer.

void (*`vol_down`)(struct bt_vcp_vol_ctlr *vol_ctlr, int err)

Callback function for *bt_vcp_vol_ctlr_vol_down()*.

Called when the volume down procedure is completed.

Param vol_ctlr
Volume Controller instance pointer.

Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.

void (*`vol_up`)(struct bt_vcp_vol_ctlr *vol_ctlr, int err)

Callback function for *bt_vcp_vol_ctlr_vol_up()*.

Called when the volume up procedure is completed.

Param vol_ctlr
Volume Controller instance pointer.

Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.

void (*`mute`)(struct bt_vcp_vol_ctlr *vol_ctlr, int err)

Callback function for *bt_vcp_vol_ctlr_mute()*.

Called when the mute procedure is completed.

Param vol_ctlr
Volume Controller instance pointer.

Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.

void (*`unmute`)(struct bt_vcp_vol_ctlr *vol_ctlr, int err)

Callback function for *bt_vcp_vol_ctlr_unmute()*.

Called when the unmute procedure is completed.

Param vol_ctlr
Volume Controller instance pointer.

Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.

void (*vol_down_unmute)(struct bt_vcp_vol_ctlr *vol_ctlr, int err)

Callback function for bt_vcp_vol_ctlr_vol_down_unmute().

Called when the volume down and unmute procedure is completed.

Param vol_ctlr
Volume Controller instance pointer.

Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.

void (*vol_up_unmute)(struct bt_vcp_vol_ctlr *vol_ctlr, int err)

Callback function for bt_vcp_vol_ctlr_vol_up_unmute().

Called when the volume up and unmute procedure is completed.

Param vol_ctlr
Volume Controller instance pointer.

Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.

void (*vol_set)(struct bt_vcp_vol_ctlr *vol_ctlr, int err)

Callback function for bt_vcp_vol_ctlr_vol_set().

Called when the set absolute volume procedure is completed.

Param vol_ctlr
Volume Controller instance pointer.

Param err
Error value. 0 on success, GATT error on positive value or errno on negative value.

struct bt_vocs_cb vocs_cb

Volume Offset Control Service callbacks.

struct bt_aics_cb aics_cb

Audio Input Control Service callbacks.

Bluetooth: Basic Audio Profile This document describes how to run Basic Audio Profile functionality which includes:

- Capabilities and Endpoint discovery
- Audio Stream Endpoint procedures

Commands

```
bap --help
Subcommands:
   init
   select_broadcast       : <stream>
   create_broadcast       : [preset <preset_name>] [enc <broadcast_code>]
   start_broadcast        :
   stop_broadcast         :
   delete_broadcast       :
```

(continues on next page)

(continued from previous page)

```
  create_broadcast_sink  : 0x<broadcast_id>
  sync_broadcast         : 0x<bis_index> [[[0x<bis_index>] 0x<bis_index>] ...]
                           [bcode <broadcast code> || bcode_str <broadcast code
                           as string>]
  stop_broadcast_sink    : Stops broadcast sink
  term_broadcast_sink    :
  discover               : [dir: sink, source]
  config                 : <direction: sink, source> <index> [loc <loc_bits>]
                           [preset <preset_name>]
  stream_qos             : interval [framing] [latency] [pd] [sdu] [phy] [rtn]
  qos                    : Send QoS configure for Unicast Group
  enable                 : [context]
  connect                : Connect the CIS of the stream
  stop
  list
  print_ase_info         : Print ASE info for default connection
  metadata               : [context]
  start
  disable
  release
  select_unicast         : <stream>
  preset                 : <sink, source, broadcast> [preset]
                           [config
                                 [freq <frequency>]
                                 [dur <duration>]
                                 [chan_alloc <location>]
                                 [frame_len <frame length>]
                                 [frame_blks <frame blocks>]]
                           [meta
                                 [pref_ctx <context>]
                                 [stream_ctx <context>]
                                 [program_info <program info>]
                                 [lang <ISO 639-3 lang>]
                                 [ccid_list <ccids>]
                                 [parental_rating <rating>]
                                 [program_info_uri <URI>]
                                 [audio_active_state <state>]
                                 [bcast_flag]
                                 [extended <meta>]
                                 [vendor <meta>]]
  send                   : Send to Audio Stream [data]
  bap_stats              : Sets or gets the statistics reporting interval in # of
                           packets
  set_location           : <direction: sink, source> <location bitmask>
  set_context            : <direction: sink, source><context bitmask> <type:
                           supported, available>
```

Table 14: State Machine Transitions

Command	Depends	Allowed States	Next States
init	none	any	none
discover	init	any	any
config	discover	idle/codec-configured/qos-configured	codec-configured
qos	config	codec-configured/qos-configured	qos-configured
enable	qos	qos-configured	enabling
connect	qos/enable	qos-configured/enabling	qos-configured/enabling
[start]	enable/connect	enabling	streaming
disable	enable	enabling/streaming	disabling
[stop]	disable	disabling	qos-configure/idle
release	config	any	releasing/codec-configure/idle
list	none	any	none
select_unicast	none	any	none
send	enable	streaming	none

Example Central Connect and establish a sink stream:

```
uart:~$ bt init
uart:~$ bap init
uart:~$ bt connect <address>
uart:~$ gatt exchange-mtu
uart:~$ bap discover sink
uart:~$ bap config sink 0
uart:~$ bap qos
uart:~$ bap enable
uart:~$ bap connect
```

Connect and establish a source stream:

```
uart:~$ bt init
uart:~$ bap init
uart:~$ bt connect <address>
uart:~$ gatt exchange-mtu
uart:~$ bap discover source
uart:~$ bap config source 0
uart:~$ bap qos
uart:~$ bap enable
uart:~$ bap connect
uart:~$ bap start
```

Disconnect and release:

```
uart:~$ bap disable
uart:~$ bap release
```

Example Peripheral Listen:

```
uart:~$ bt init
uart:~$ bap init
uart:~$ bt advertise on
```

Server initiated disable and release:

```
uart:~$ bap disable
uart:~$ bap release
```

Example Broadcast Source Create and establish a broadcast source stream:

```
uart:~$ bap init
uart:~$ bap create_broadcast
uart:~$ bap start_broadcast
```

Stop and release a broadcast source stream:

```
uart:~$ bap stop_broadcast
uart:~$ bap delete_broadcast
```

Example Broadcast Sink Scan for and establish a broadcast sink stream. The command `bap create_broadcast_sink` will either use existing periodic advertising sync (if exist) or start scanning and sync to the periodic advertising with the provided broadcast ID before syncing to the BIG.

```
uart:~$ bap init
uart:~$ bap create_broadcast_sink 0xEF6716
No PA sync available, starting scanning for broadcast_id
Found broadcaster with ID 0xEF6716 and addr 03:47:95:75:C0:08 (random) and sid 0x00
Attempting to PA sync to the broadcaster
PA synced to broadcast with broadcast ID 0xEF6716
Attempting to sync to the BIG
Received BASE from sink 0x20019080:
Presentation delay: 40000
Subgroup count: 1
Subgroup 0x20024182:
   Codec Format: 0x06
   Company ID  : 0x0000
   Vendor ID   : 0x0000
   codec cfg id 0x06 cid 0x0000 vid 0x0000 count 16
      Codec specific configuration:
      Sampling frequency: 16000 Hz (3)
      Frame duration: 10000 us (1)
      Channel allocation:
               Front left (0x00000001)
               Front right (0x00000002)
      Octets per codec frame: 40
      Codec specific metadata:
      Streaming audio contexts:
         Unspecified (0x0001)
      BIS index: 0x01
         codec cfg id 0x06 cid 0x0000 vid 0x0000 count 6
         Codec specific configuration:
            Channel allocation:
               Front left (0x00000001)
         Codec specific metadata:
            None
      BIS index: 0x02
         codec cfg id 0x06 cid 0x0000 vid 0x0000 count 6
         Codec specific configuration:
            Channel allocation:
               Front right (0x00000002)
         Codec specific metadata:
            None
Possible indexes: 0x01 0x02
```

(continues on next page)

(continued from previous page)

```
Sink 0x20019110 is ready to sync without encryption
uart:~$ bap sync_broadcast 0x01
```

Syncing to encrypted broadcast If the broadcast is encrypted, the broadcast code can be entered with the `bap sync_broadcast` command as such:

```
Sink 0x20019110 is ready to sync with encryption
uart:~$ bap sync_broadcast 0x01 bcode 0102030405060708090a0b0c0d0e0f
```

The broadcast code can be 1-16 values, either as a string or a hexadecimal value.

```
Sink 0x20019110 is ready to sync with encryption
uart:~$ bap sync_broadcast 0x01 bcode_str thisismycode
```

Stop and release a broadcast sink stream:

```
uart:~$ bap stop_broadcast_sink
uart:~$ bap term_broadcast_sink
```

Init The `init` command register local PAC records which are necessary to be able to configure stream and properly manage capabilities in use.

Table 15: State Machine Transitions

Depends	Allowed States	Next States
none	any	none

```
uart:~$ bap init
```

Discover PAC(s) and ASE(s) Once connected the `discover` command discover PAC records and ASE characteristics representing remote endpoints.

Table 16: State Machine Transitions

Depends	Allowed States	Next States
init	any	any

Note

Use command `gatt exchange-mtu` to make sure the MTU is configured properly.

```
uart:~$ gatt exchange-mtu
Exchange pending
Exchange successful
uart:~$ bap discover [type: sink, source]
uart:~$ bap discover sink
conn 0x2000b168: codec_cap 0x2001f8ec dir 0x02
codec cap id 0x06 cid 0x0000 vid 0x0000
   Codec specific capabilities:
      Supported sampling frequencies:
         8000 Hz (0x0001)
```

(continues on next page)

(continued from previous page)

```
         11025 Hz (0x0002)
         16000 Hz (0x0004)
         22050 Hz (0x0008)
         24000 Hz (0x0010)
         32000 Hz (0x0020)
         44100 Hz (0x0040)
         48000 Hz (0x0080)
         88200 Hz (0x0100)
         96000 Hz (0x0200)
         176400 Hz (0x0400)
         192000 Hz (0x0800)
         384000 Hz (0x1000)
      Supported frame durations:
         10 ms (0x02)
      Supported channel counts:
         1 channel (0x01)
      Supported octets per codec frame counts:
         Min: 40
         Max: 120
      Supported max codec frames per SDU: 1
   Codec capabilities metadata:
      Preferred audio contexts:
         Conversational (0x0002)
         Media (0x0004)
ep 0x81754e0
ep 0x81755d4
Discover complete: err 0
```

Select preset The `preset` command can be used to either print the default preset configuration or set a different one, it is worth noting that it doesn't change any stream previously configured.

```
uart:~$ bap preset
preset - <sink, source, broadcast> [preset]
         [config
               [freq <frequency>]
               [dur <duration>]
               [chan_alloc <location>]
               [frame_len <frame length>]
               [frame_blks <frame blocks>]]
         [meta
               [pref_ctx <context>]
               [stream_ctx <context>]
               [program_info <program info>]
               [lang <ISO 639-3 lang>]
               [ccid_list <ccids>]
               [parental_rating <rating>]
               [program_info_uri <URI>]
               [audio_active_state <state>]
               [bcast_flag]
               [extended <meta>]
               [vendor <meta>]]
uart:~$ bap preset sink
16_2_1
codec cfg id 0x06 cid 0x0000 vid 0x0000 count 16
   Codec specific configuration:
      Sampling frequency: 16000 Hz (3)
      Frame duration: 10000 us (1)
      Channel allocation:
                  Front left (0x00000001)
```

(continues on next page)

(continued from previous page)

```
                  Front right (0x00000002)
      Octets per codec frame: 40
   Codec specific metadata:
      Streaming audio contexts:
         Game (0x0008)
QoS: interval 10000 framing 0x00 phy 0x02 sdu 40 rtn 2 latency 10 pd 40000

uart:~$ bap preset sink 32_2_1
32_2_1
codec cfg id 0x06 cid 0x0000 vid 0x0000 count 16
   Codec specific configuration:
      Sampling frequency: 32000 Hz (6)
      Frame duration: 10000 us (1)
      Channel allocation:
                  Front left (0x00000001)
                  Front right (0x00000002)
      Octets per codec frame: 80
   Codec specific metadata:
      Streaming audio contexts:
         Game (0x0008)
   QoS: interval 10000 framing 0x00 phy 0x02 sdu 80 rtn 2 latency 10 pd 40000
```

Configure preset The `bap preset` command can also be used to configure the preset used for the subsequent commands. It is possible to add or set (or reset) any value. To reset the preset, the command can simply be run without the `config` or `meta` parameter. The parameters are using the assigned numbers values.

```
uart:~$ bap preset sink 32_2_1
32_2_1
codec cfg id 0x06 cid 0x0000 vid 0x0000 count 16
data #0: type 0x01 value_len 1
00000000: 06                                              |.               |
data #1: type 0x02 value_len 1
00000000: 01                                              |.               |
data #2: type 0x03 value_len 4
00000000: 03 00 00 00                                     |....            |
data #3: type 0x04 value_len 2
00000000: 50 00                                           |P.              |
meta #0: type 0x02 value_len 2
00000000: 08 00                                           |..              |
QoS: interval 10000 framing 0x00 phy 0x02 sdu 80 rtn 2 latency 10 pd 40000

uart:~$ bap preset sink 32_2_1 config freq 10
32_2_1
codec cfg id 0x06 cid 0x0000 vid 0x0000 count 16
data #0: type 0x01 value_len 1
00000000: 0a                                              |.               |
data #1: type 0x02 value_len 1
00000000: 01                                              |.               |
data #2: type 0x03 value_len 4
00000000: 03 00 00 00                                     |....            |
data #3: type 0x04 value_len 2
00000000: 50 00                                           |P.              |
meta #0: type 0x02 value_len 2
00000000: 08 00                                           |..              |
QoS: interval 10000 framing 0x00 phy 0x02 sdu 80 rtn 2 latency 10 pd 40000

uart:~$ bap preset sink 32_2_1 config freq 10 meta lang "eng" stream_ctx 4
32_2_1
```

(continues on next page)

(continued from previous page)

```
codec cfg id 0x06 cid 0x0000 vid 0x0000 count 16
data #0: type 0x01 value_len 1
00000000: 0a                                              |.               |
data #1: type 0x02 value_len 1
00000000: 01                                              |.               |
data #2: type 0x03 value_len 4
00000000: 03 00 00 00                                     |....            |
data #3: type 0x04 value_len 2
00000000: 50 00                                           |P.              |
meta #0: type 0x02 value_len 2
00000000: 04 00                                           |..              |
meta #1: type 0x04 value_len 3
00000000: 65 6e 67                                        |eng             |
QoS: interval 10000 framing 0x00 phy 0x02 sdu 80 rtn 2 latency 10 pd 40000
```

Configure Codec The `config` command attempts to configure a stream for the given direction using a preset codec configuration which can either be passed directly or in case it is omitted the default preset is used.

Table 17: State Machine Transitions

Depends	Allowed States	Next States
discover	idle/codec-configured/qos-configured	codec-configured

```
uart:~$ bap config <direction: sink, source> <index> [loc <loc_bits>] [preset <preset_name>]
uart:~$ bap config sink 0
Setting location to 0x00000000
ASE config: preset 16_2_1
stream 0x2000df70 config operation rsp_code 0 reason 0
```

Configure Stream QoS The `stream_qos` Sets a new stream QoS.

```
uart:~$ bap stream_qos <interval> [framing] [latency] [pd] [sdu] [phy] [rtn]
uart:~$ bap stream_qos 10
```

Configure QoS The `qos` command attempts to configure the stream QoS using the preset configuration, each individual QoS parameter can be set with use optional parameters.

Table 18: State Machine Transitions

Depends	Allowed States	Next States
config	qos-configured/codec-configured	qos-configured

```
uart:~$ bap qos
```

Enable The `enable` command attempts to enable the stream previously configured.

Table 19: State Machine Transitions

Depends	Allowed States	Next States
qos	qos-configured	enabling

```
uart:~$ bap enable [context]
uart:~$ bap enable Media
```

Connect The `connect` command attempts to connect the stream previously configured. Sink streams will have to be started by the unicast server, and source streams will have to be started by the unicast client.

Table 20: State Machine Transitions

Depends	Allowed States	Next States
qos/enable	qos-configured/enabling	qos-configured/enabling

```
uart:~$ bap connect
```

Start The `start` command is only necessary when starting a source stream.

Table 21: State Machine Transitions

Depends	Allowed States	Next States
enable/connect	enabling	streaming

```
uart:~$ bap start
```

Disable The `disable` command attempts to disable the stream previously enabled, if the remote peer accepts then the ISO disconnection procedure is also initiated.

Table 22: State Machine Transitions

Depends	Allowed States	Next States
enable	enabling/streaming	disabling

```
uart:~$ bap disable
```

Stop The `stop` command is only necessary when acting as a sink as it indicates to the source the stack is ready to stop receiving data.

Table 23: State Machine Transitions

Depends	Allowed States	Next States
disable	disabling	qos-configure/idle

```
uart:~$ bap stop
```

Release The `release` command releases the current stream and its configuration.

Table 24: State Machine Transitions

Depends	Allowed States	Next States
config	any	releasing/codec-configure/idle

```
uart:~$ bap release
```

List The `list` command list the available streams.

Table 25: State Machine Transitions

Depends	Allowed States	Next States
none	any	none

```
uart:~$ bap list
*0: ase 0x01 dir 0x01 state 0x01
```

Select Unicast The `select_unicast` command set a unicast stream as default.

Table 26: State Machine Transitions

Depends	Allowed States	Next States
none	any	none

```
uart:~$ bap select <ase>
uart:~$ bap select 0x01
Default stream: 1
```

To select a broadcast stream:

```
uart:~$ bap select 0x01 broadcast
Default stream: 1 (broadcast)
```

Send The send command sends data over BAP Stream.

Table 27: State Machine Transitions

Depends	Allowed States	Next States
enable	streaming	none

```
uart:~$ bap send [count]
uart:~$ bap send
Audio sending...
```

Bluetooth: Broadcast Audio Profile Broadcast Assistant This document describes how to run the BAP Broadcast Assistant functionality. Note that in the examples below, some lines of debug have been removed to make this shorter and provide a better overview.

The Broadcast Assistant is responsible for offloading scan for a resource restricted device, such that scanning does not drain the battery. The Broadcast Assistant shall support scanning for periodic advertisements and may optionally support the periodic advertisements synchronization transfer (PAST) protocol.

The Broadcast Assistant will typically be phones or laptops. The Broadcast Assistant scans for periodic advertisements and transfer information to the server.

It is necessary to have `CONFIG_BT_BAP_BROADCAST_ASSISTANT_LOG_LEVEL_DBG` enabled for using the Broadcast Assistant interactively.

When the Bluetooth stack has been initialized (`bt init`), and a device has been connected, the Broadcast Assistant can discover BASS on the connected device calling `bap_broadcast_assistant discover`, which will start a discovery for the BASS UUIDs and store the handles, and subscribe to all notifications.

```
uart:~$ bap_broadcast_assistant --help
bap_broadcast_assistant - Bluetooth BAP broadcast assistant client shell
                          commands
Subcommands:
discover          : Discover BASS on the server
scan_start        : Start scanning for broadcasters
scan_stop         : Stop scanning for BISs
add_src           : Add a source <address: XX:XX:XX:XX:XX:XX> <type:
                     public/random> <adv_sid> <sync_pa> <broadcast_id>
                     [<pa_interval>] [<sync_bis>] [<metadata>]
add_broadcast_id  : Add a source by broadcast ID <broadcast_id> <sync_pa>
                     [<sync_bis>] [<metadata>]
add_pa_sync       : Add a PA sync as a source <sync_pa> <broadcast_id>
                     [bis_index [bis_index [bix_index [...]]]]>
mod_src           : Set sync <src_id> <sync_pa> [<pa_interval> | "unknown"] [<sync_bis>]
                     [<metadata>]
broadcast_code    : Send a string-based broadcast code of up to 16 bytes
                     <src_id> <broadcast code>
rem_src           : Remove a source <src_id>
read_state        : Remove a source <index>
```

Example usage

Setup

```
uart:~$ bt init
uart:~$ bap init
uart:~$ bt connect xx:xx:xx:xx:xx:xx public
```

When connected

> **Note**
>
> The Broadcast Assistant will not actually start scanning for periodic advertisements, as that feature is still, at the time of writing, not implemented.

Start scanning for periodic advertisements for a server:

```
uart:~$ bap_broadcast_assistant discover
BASS discover done with 1 recv states
```

(continues on next page)

(continued from previous page)

```
uart:~$ bap_broadcast_assistant scan_start true
BASS scan start successful
Found broadcaster with ID 0x05BD38 and addr 1E:4D:0A:AA:6E:49 (random) and sid 0x00
```

Adding a source to the receive state with add_src:

```
uart:~$ bap_broadcast_assistant add_src 11:22:33:44:55:66 public 5 1 1
BASS recv state: src_id 0, addr 11:22:33:44:55:66 (public), sid 5, sync_state 1, encrypt_
↪state 00000000000000000000000000000000
     [0]: BIS sync 0, metadata_len 0
```

Adding a source to the receive state with add_broadcast_id (recommended):

```
uart:~$ bap_broadcast_assistant add_broadcast_id 0x05BD38 true
[DEVICE]: 1E:4D:0A:AA:6E:49 (random), AD evt type 5, RSSI -28 Broadcast Audio Source C:0␣
↪S:0 D:0 SR:0 E:1 Prim: LE 1M, Secn: LE 2M, Interval: 0x03c0 (1200000 us), SID: 0x0
Found BAP broadcast source with address 1E:4D:0A:AA:6E:49 (random) and ID 0x05BD38
BASS recv state: src_id 0, addr 1E:4D:0A:AA:6E:49 (random), sid 0, sync_state 0, encrypt_
↪state 0
      [0]: BIS sync 0x0000, metadata_len 0
BASS add source successful
BASS recv state: src_id 0, addr 1E:4D:0A:AA:6E:49 (random), sid 0, sync_state 2, encrypt_
↪state 0
      [0]: BIS sync 0x0000, metadata_len 0
BASS recv state: src_id 0, addr 1E:4D:0A:AA:6E:49 (random), sid 0, sync_state 2, encrypt_
↪state 0
      [0]: BIS sync 0x0000, metadata_len 4
               Metadata length 2, type 2, data: 0100
```

Modifying a receive state:

```
uart:~$ bap_broadcast_assistant mod_src 0 true 0x03c0 0x02
BASS modify source successful
BASS recv state: src_id 0, addr 1E:4D:0A:AA:6E:49 (random), sid 0, sync_state 2, encrypt_
↪state 0
      [0]: BIS sync 0x0001, metadata_len 4
               Metadata length 2, type 2, data: 0100
```

Supplying a broadcast code:

```
uart:~$ bap_broadcast_assistant broadcast_code 0 secretCode
Sending broadcast code:
00000000: 73 65 63 72 65 74 43 6f 64 65 00 00 00 00 00 00 |secretCo de....|
uart:~$ BASS broadcast code successful
```

Bluetooth: Broadcast Audio Profile Scan Delegator This document describes how to run the Scan Delegator functionality, Note that in the examples below, some lines of debug have been removed to make this shorter and provide a better overview.

The Scan Delegator may optionally support the periodic advertisements synchronization transfer (PAST) protocol.

The Scan Delegator server typically resides on devices that have inputs or outputs.

It is necessary to have `CONFIG_BT_BAP_SCAN_DELEGATOR_LOG_LEVEL_DBG` enabled for using the Scan Delegator interactively.

The Scan Delegator can currently only set the sync state of a receive state, but does not actually support syncing with periodic advertisements yet.

```
bap_scan_delegator --help
bap_scan_delegator - Bluetooth BAP Scan Delegator shell commands
Subcommands:
   init    :Initialize the service and register callbacks
   synced  :Set server scan state <src_id> <pa_synced> <bis_syncs> <enc_state>
```

Example Usage

Setup

```
uart:~$ bt init
uart:~$ bt advertise on
Advertising started
```

When connected Set sync state for a source:

```
uart:~$ bap_scan_delegator synced 0 1 3 0
```

Bluetooth: Common Audio Profile Shell This document describes how to run the Common Audio Profile functionality.

CAP Acceptor The Acceptor will typically be a resource-constrained device, such as a headset, earbud or hearing aid. The Acceptor can initialize a Coordinated Set Identification Service instance, if it is in a pair with one or more other CAP Acceptors.

Using the CAP Acceptor When the Bluetooth stack has been initialized (`bt init`), the Acceptor can be registered by calling `cap_acceptor init`, which will register the CAS and CSIS services, as well as register callbacks.

```
cap_acceptor --help
cap_acceptor - Bluetooth CAP acceptor shell commands
Subcommands:
  init          :Initialize the service and register callbacks [size <int>]
                 [rank <int>] [not-lockable] [sirk <data>]
  lock          :Lock the set
  release       :Release the set [force]
  sirk          :Set the currently used SIRK <sirk>
  get_sirk      :Get the currently used SIRK
  sirk_rsp      :Set the response used in SIRK requests <accept, accept_enc,
                 reject, oob>
```

Besides initializing the CAS and the CSIS, there are also commands to lock and release the CSIS instance, as well as printing and modifying access to the SIRK of the CSIS.

Setting a new SIRK This command can modify the currently used SIRK. To get the new RSI to advertise on air, `bt adv-data` or `bt advertise` must be called again to set the new advertising data. If `CONFIG_BT_CSIP_SET_MEMBER_NOTIFIABLE` is enabled, this will also notify connected clients.

```
uart:~$ cap_acceptor sirk 00112233445566778899aabbccddeeff
SIRK updated
```

Getting the current SIRK This command can get the currently used SIRK.

```
uart:~$ cap_acceptor get_sirk
SIRK
36 04 9a dc 66 3a a1 a1 |6...f:..
1d 9a 2f 41 01 73 3e 01 |../A.s>.
```

CAP Initiator The Initiator will typically be a resource-rich device, such as a phone or PC. The Initiator can discover CAP Acceptors's CAS and optional CSIS services. The CSIS service can be read to provide information about other CAP Acceptors in the same Coordinated Set. The Initiator can execute stream control procedures on sets of devices, either ad-hoc or Coordinated, and thus provides an easy way to setup multiple streams on multiple devices at once.

Using the CAP Initiator When the Bluetooth stack has been initialized (`bt init`), the Initiator can discover CAS and the optionally included CSIS instance by calling (`cap_initiator discover`). The CAP initiator also supports broadcast audio as a source.

```
uart:~$ cap_initiator --help
cap_initiator - Bluetooth CAP initiator shell commands
Subcommands:
  discover          : Discover CAS
  unicast_start     : Unicast Start [csip] [sinks <cnt> (default 1)] [sources
                      <cnt> (default 1)] [conns (<cnt> | all) (default 1)]
  unicast_list      : Unicast list streams
  unicast_update    : Unicast Update <all | stream [stream [stream...]]>
  unicast_stop      : Unicast stop streams [stream [stream [stream...]]] (all by default)
  unicast_cancel    : Unicast cancel current procedure
  ac_1              : Unicast audio configuration 1
  ac_2              : Unicast audio configuration 2
  ac_3              : Unicast audio configuration 3
  ac_4              : Unicast audio configuration 4
  ac_5              : Unicast audio configuration 5
  ac_6_i            : Unicast audio configuration 6(i)
  ac_6_ii           : Unicast audio configuration 6(ii)
  ac_7_i            : Unicast audio configuration 7(i)
  ac_7_ii           : Unicast audio configuration 7(ii)
  ac_8_i            : Unicast audio configuration 8(i)
  ac_8_ii           : Unicast audio configuration 8(ii)
  ac_9_i            : Unicast audio configuration 9(i)
  ac_9_ii           : Unicast audio configuration 9(ii)
  ac_10             : Unicast audio configuration 10
  ac_11_i           : Unicast audio configuration 11(i)
  ac_11_ii          : Unicast audio configuration 11(ii)
  broadcast_start   :
  broadcast_update  : <meta>
  broadcast_stop    :
  broadcast_delete  :
  ac_12             : Broadcast audio configuration 12
  ac_13             : Broadcast audio configuration 13
  ac_14             : Broadcast audio configuration 14
```

Before being able to perform any stream operation, the device must also perform the `bap discover` operation to discover the ASEs and PAC records. The `bap init` command also needs to be called.

When connected Discovering CAS and CSIS on a device:

```
uart:~$ cap_initiator discover
discovery completed with CSIS
```

Discovering ASEs and PAC records on a device:

```
uart:~$ bap discover
conn 0x81cc260: #0: codec 0x81d5b28 dir 0x01
codec 0x06 cid 0x0000 vid 0x0000 count 5
data #0: type 0x01 len 2
00000000: f5                                                |.               |
data #1: type 0x02 len 1
data #2: type 0x03 len 1
data #3: type 0x04 len 4
00000000: 1e 00 f0                                          |...             |
data #4: type 0x05 len 1
meta #0: type 0x01 len 2
00000000: 06                                                |.               |
dir 1 loc 1
snk ctx 6 src ctx 6
Conn: 0x81cc260, Sink #0: ep 0x81e4248
Conn: 0x81cc260, Sink #1: ep 0x81e46a8
conn 0x81cc260: #0: codec 0x81d5f00 dir 0x02
codec 0x06 cid 0x0000 vid 0x0000 count 5
data #0: type 0x01 len 2
00000000: f5                                                |.               |
data #1: type 0x02 len 1
data #2: type 0x03 len 1
data #3: type 0x04 len 4
00000000: 1e 00 f0                                          |...             |
data #4: type 0x05 len 1
meta #0: type 0x01 len 2
00000000: 06                                                |.               |
dir 2 loc 1
snk ctx 6 src ctx 6
Conn: 0x81cc260, Source #0: ep 0x81e5c88
Conn: 0x81cc260, Source #1: ep 0x81e60e8
Discover complete: err 0
```

Both of the above commands should be done for each device that you want to use in the set. To use multiple devices, simply connect to more and then use `bt select` the device to execute the commands on.

Once all devices have been connected and the respective discovery commands have been called, the `cap_initiator unicast_start` command can be used to put one or more streams into the streaming state.

```
uart:~$ cap_initiator unicast_start sinks 1 sources 0 conns all
Setting up 1 sinks and 0 sources on each (2) conn
Starting 1 streams
Unicast start completed
```

To stop all the streams that has been started, the `cap_initiator unicast_stop` command can be used.

```
uart:~$ cap_initiator unicast_stop all
Unicast stop completed
```

When doing broadcast To start a broadcast as the CAP initiator there are a few steps to be done:

1. Create and configure an extended advertising set with periodic advertising
2. Create and configure a broadcast source
3. Setup extended and periodic advertising data

The following commands will setup a CAP broadcast source using the 16_2_1 preset (defined by BAP):

```
bt init
bap init
bt adv-create nconn-nscan ext-adv name
bt per-adv-param
bap preset broadcast 16_2_1
cap_initiator ac_12
bt adv-data discov
bt per-adv-data
cap_initiator broadcast_start
```

The broadcast source is created by the `cap_initiator ac_12`, `cap_initiator ac_13`, and `cap_initiator ac_14` commands, configuring the broadcast source for the defined audio configurations from BAP. The broadcast source can then be stopped with `cap_initiator broadcast_stop` or deleted with `cap_initiator broadcast_delete`.

The metadata of the broadcast source can be updated at any time, including when it is already streaming. To update the metadata the `cap_initiator broadcast_update` command can be used. The command takes an array of data, and the only requirement (besides having valid data) is that the streaming context shall be set. For example to set the streaming context to media, the command can be used as

```
cap_initiator broadcast_update 03020400
CAP Broadcast source updated with new metadata. Update the advertised base via `bt per-adv-
↪data`
bt per-adv-data
```

The `bt per-adv-data` command should be used afterwards to update the data is the advertised BASE. The data must be little-endian, so in the above example the metadata `03020400` is setting the metadata entry with `03` as the length, `02` as the type (streaming context) and `0400` as the value `BT_AUDIO_CONTEXT_TYPE_MEDIA` (which has the numeric value of 0x).

CAP Commander The Commander will typically be a either co-located with a CAP Initiator or be on a separate resource-rich mobile device, such as a phone or smartwatch. The Commander can discover CAP Acceptors's CAS and optional CSIS services. The CSIS service can be read to provide information about other CAP Acceptors in the same Coordinated Set. The Commander can provide information about broadcast sources to CAP Acceptors or coordinate capture and rendering information such as mute or volume states.

Using the CAP Commander When the Bluetooth stack has been initialized (`bt init`), the Commander can discover CAS and the optionally included CSIS instance by calling (`cap_commander discover`).

```
cap_commander --help
cap_commander - Bluetooth CAP commander shell commands
Subcommands:
  discover                :Discover CAS
  cancel                  :CAP commander cancel current procedure
  change_volume           :Change volume on all connections <volume>
  change_volume_mute      :Change volume mute state on all connections <mute>
  change_volume_offset    :Change volume offset per connection <volume_offset
                           [volume_offset [...]]>
```

(continues on next page)

(continued from previous page)

```
  change_microphone_mute  :Change microphone mute state on all connections <mute>
  change_microphone_gain  :Change microphone gain per connection <gain
                           [gain [...]]>
```

Before being able to perform any stream operation, the device must also perform the `bap discover` operation to discover the ASEs and PAC records. The `bap init` command also needs to be called.

When connected

Discovering CAS and CSIS on a device

```
uart:~$ cap_commander discover
discovery completed with CSIS
```

Setting the volume on all connected devices

```
uart:~$ vcp_vol_ctlr discover
VCP discover done with 1 VOCS and 1 AICS
uart:~$ cap_commander change_volume 15
uart:~$ cap_commander change_volume 15
Setting volume to 15 on 2 connections
VCP volume 15, mute 0
VCP vol_set done
VCP volume 15, mute 0
VCP flags 0x01
VCP vol_set done
Volume change completed
```

Setting the volume offset on one or more devices The offsets are set by connection index, so connection index 0 gets the first offset, and index 1 gets the second offset, etc.:

```
uart:~$ bt connect <device A>
Connected: <device A>
uart:~$ cap_commander discover
discovery completed with CSIS
uart:~$ vcp_vol_ctlr discover
VCP discover done with 1 VOCS and 1 AICS
uart:~$
uart:~$ bt connect <device B>
Connected: <device B>
uart:~$ cap_commander discover
discovery completed with CSIS
uart:~$ vcp_vol_ctlr discover
VCP discover done with 1 VOCS and 1 AICS
uart:~$
uart:~$ cap_commander change_volume_offset 10
Setting volume offset on 1 connections
VOCS inst 0x200140a4 offset 10
Offset set for inst 0x200140a4
Volume offset change completed
uart:~$
uart:~$ cap_commander change_volume_offset 10 15
Setting volume offset on 2 connections
Offset set for inst 0x200140a4
VOCS inst 0x20014188 offset 15
```

(continues on next page)

(continued from previous page)

```
Offset set for inst 0x20014188
Volume offset change completed
```

Setting the volume mute on all connected devices

```
uart:~$ bt connect <device A>
Connected: <device A>
uart:~$ cap_commander discover
discovery completed with CSIS
uart:~$ vcp_vol_ctlr discover
VCP discover done with 1 VOCS and 1 AICS
uart:~$
uart:~$ bt connect <device B>
Connected: <device B>
uart:~$ cap_commander discover
discovery completed with CSIS
uart:~$ vcp_vol_ctlr discover
VCP discover done with 1 VOCS and 1 AICS
uart:~$
uart:~$ cap_commander change_volume_mute 1
Setting volume mute to 1 on 2 connections
VCP volume 100, mute 1
VCP mute done
VCP volume 100, mute 1
VCP mute done
Volume mute change completed
uart:~$ cap_commander change_volume_mute 0
Setting volume mute to 0 on 2 connections
VCP volume 100, mute 0
VCP unmute done
VCP volume 100, mute 0
VCP unmute done
Volume mute change completed
```

Setting the microphone mute on all connected devices

```
uart:~$ bt connect <device A>
Connected: <device A>
uart:~$ cap_commander discover
discovery completed with CSIS
uart:~$ micp_mic_ctlr discover
MICP discover done with 1 VOCS and 1 AICS
uart:~$
uart:~$ bt connect <device B>
Connected: <device B>
uart:~$ cap_commander discover
discovery completed with CSIS
uart:~$ micp_mic_ctlr discover
MICP discover done with 1 VOCS and 1 AICS
uart:~$
uart:~$ cap_commander change_microphone_mute 1
Setting microphone mute to 1 on 2 connections
MICP microphone 100, mute 1
MICP mute done
MICP microphone 100, mute 1
MICP mute done
Microphone mute change completed
uart:~$ cap_commander change_microphone_mute 0
Setting microphone mute to 0 on 2 connections
```

(continues on next page)

(continued from previous page)

```
MICP microphone 100, mute 0
MICP unmute done
MICP microphone 100, mute 0
MICP unmute done
Microphone mute change completed
```

Setting the microphone gain on one or more devices The gains are set by connection index, so connection index 0 gets the first offset, and index 1 gets the second offset, etc.:

```
uart:~$ bt connect <device A>
Connected: <device A>
uart:~$ cap_commander discover
discovery completed with CSIS
uart:~$ micp_mic_ctlr discover
MICP discover done with 1 AICS
uart:~$
uart:~$ bt connect <device B>
Connected: <device B>
uart:~$ cap_commander discover
discovery completed with CSIS
uart:~$ micp_mic_ctlr discover
MICP discover done with 1 AICS
uart:~$
uart:~$ cap_commander change_microphone_gain 10
Setting microphone gain on 1 connections
AICS inst 0x200140a4 state gain 10, mute 0, mode 0
Gain set for inst 0x200140a4
Microphone gain change completed
uart:~$
uart:~$ cap_commander change_microphone_gain 10 15
Setting microphone gain on 2 connections
Gain set for inst 0x200140a4
AICS inst 0x20014188 state gain 15, mute 0, mode 0
Gain set for inst 0x20014188
Microphone gain change completed
```

Starting a broadcast reception

```
uart:~$ bt connect <device A>
Connected: <device A>
uart:~$ bap_init
uart:~$ cap_commander discover
discovery completed with CSIS
uart:~$ bap_broadcast_assistant discover
BASS discover done with 1 recv states
uart:~$ cap_commander broadcast_reception_start <device B> 0 4
Starting broadcast reception on 1 connection(s)
```

Bluetooth: Call Control Profile This document describes how to run the call control functionality, both as a client and as a (telephone bearer service (TBS)) server. Note that in the examples below, some lines of debug have been removed to make this shorter and provide a better overview.

Telephone Bearer Service Client The telephone bearer service client will typically exist on a resource restricted device, such as headphones, but may also exist on e.g. phones or laptops.

The call control client will also thus typically be the advertiser. The client can control the states of calls on a server using the call control point.

It is necessary to have `CONFIG_BT_TBS_CLIENT_LOG_LEVEL_DBG` enabled for using the client interactively.

Using the telephone bearer service client When the Bluetooth stack has been initialized (`bt init`), and a device has been connected, the telephone bearer service client can discover TBS on the connected device calling `tbs_client discover`, which will start a discovery for the TBS UUIDs and store the handles, and optionally subscribe to all notifications (default is to subscribe to all).

Since a server may have multiple TBS instances, most of the tbs_client commands will take an index (starting from 0) as input. Joining calls require at least 2 call IDs, and all call indexes shall be on the same TBS instance.

A server may also have a GTBS instance, which is an abstraction layer for all the telephone bearers on the server. If the server has both GTBS and TBS, the client may subscribe and use either when sending requests if `BT_TBS_CLIENT_GTBS` is enabled.

```
tbs_client --help
tbs_client - Bluetooth TBS_CLIENT shell commands
Subcommands:
   discover                       :Discover TBS [subscribe]
   set_signal_reporting_interval  :Set the signal reporting interval
                                   [<{instance_index, gtbs}>] <interval>
   originate                      :Originate a call [<{instance_index, gtbs}>]
                                   <uri>
   terminate                      :terminate a call [<{instance_index, gtbs}>]
                                   <id>
   accept                         :Accept a call [<{instance_index, gtbs}>] <id>
   hold                           :Place a call on hold [<{instance_index,
                                   gtbs}>] <id>
   retrieve                       :Retrieve a held call [<{instance_index,
                                   gtbs}>] <id>
   read_provider_name             :Read the bearer name [<{instance_index,
                                   gtbs}>]
   read_bearer_uci                :Read the bearer UCI [<{instance_index, gtbs}>]
   read_technology                :Read the bearer technology [<{instance_index,
                                   gtbs}>]
   read_uri_list                  :Read the bearer's supported URI list
                                   [<{instance_index, gtbs}>]
   read_signal_strength           :Read the bearer signal strength
                                   [<{instance_index, gtbs}>]
   read_signal_interval           :Read the bearer signal strength reporting
                                    interval [<{instance_index, gtbs}>]
   read_current_calls             :Read the current calls [<{instance_index,
                                   gtbs}>]
   read_ccid                      :Read the CCID [<{instance_index, gtbs}>]
   read_status_flags              :Read the in feature and status value
                                   [<{instance_index, gtbs}>]
   read_uri                       :Read the incoming call target URI
                                   [<{instance_index, gtbs}>]
   read_call_state                :Read the call state [<{instance_index, gtbs}>]
   read_remote_uri                :Read the incoming remote URI
                                   [<{instance_index, gtbs}>]
   read_friendly_name             :Read the friendly name of an incoming call
                                   [<{instance_index, gtbs}>]
   read_optional_opcodes          :Read the optional opcodes [<{instance_index,
                                   gtbs}>]
```

In the following examples, notifications from GTBS is ignored, unless otherwise specified.

Example usage

Setup

```
uart:~$ bt init
uart:~$ bt advertise on
Advertising started
```

When connected Placing a call:

```
uart:~$ tbs_client discover
<dbg> bt_tbs_client.primary_discover_func: Discover complete, found 1 instances (GTBS found)
<dbg> bt_tbs_client.discover_func: Setup complete for 1 / 1 TBS
<dbg> bt_tbs_client.discover_func: Setup complete GTBS
uart:~$ tbs_client originate 0 tel:123
<dbg> bt_tbs_client.notify_handler: Index 0
<dbg> bt_tbs_client.current_calls_notify_handler: Call 0x01 is in the dialing state with␣
↪URI tel:123
<dbg> bt_tbs_client.call_cp_notify_handler: Status: success for the originate opcode for␣
↪call 0x00
<dbg> bt_tbs_client.notify_handler: Index 0
<dbg> bt_tbs_client.current_calls_notify_handler: Call 0x01 is in the alerting state with␣
↪URI tel:123
<call answered by peer device, and status notified by TBS server>
<dbg> bt_tbs_client.notify_handler: Index 0
<dbg> bt_tbs_client.current_calls_notify_handler: Call 0x01 is in the active state with URI␣
↪tel:123
```

Placing a call on GTBS:

```
uart:~$ tbs_client originate 0 tel:123
<dbg> bt_tbs_client.notify_handler: Index 0
<dbg> bt_tbs_client.current_calls_notify_handler: Call 0x01 is in the dialing state with␣
↪URI tel:123
<dbg> bt_tbs_client.call_cp_notify_handler: Status: success for the originate opcode for␣
↪call 0x00
<dbg> bt_tbs_client.notify_handler: Index 0
<dbg> bt_tbs_client.current_calls_notify_handler: Call 0x01 is in the alerting state with␣
↪URI tel:123
<call answered by peer device, and status notified by TBS server>
<dbg> bt_tbs_client.notify_handler: Index 0
<dbg> bt_tbs_client.current_calls_notify_handler: Call 0x01 is in the active state with URI␣
↪tel:123
```

It is necessary to set an outgoing caller ID before placing a call.

Accepting incoming call from peer device:

```
<dbg> bt_tbs_client.incoming_uri_notify_handler: tel:123
<dbg> bt_tbs_client.in_call_notify_handler: tel:456
<dbg> bt_tbs_client.friendly_name_notify_handler: Peter
<dbg> bt_tbs_client.current_calls_notify_handler: Call 0x05 is in the incoming state with␣
↪URI tel:456
uart:~$ tbs_client accept 0 5
<dbg> bt_tbs_client.call_cp_callback_handler: Status: success for the accept opcode for␣
↪call 0x05
<dbg> bt_tbs_client.current_calls_notify_handler: Call 0x05 is in the active state with URI␣
↪tel
```

Terminate call:

```
uart:~$ tbs_client terminate 0 5
<dbg> bt_tbs_client.termination_reason_notify_handler: ID 0x05, reason 0x06
<dbg> bt_tbs_client.call_cp_notify_handler: Status: success for the terminate opcode for␣
↪call 0x05
<dbg> bt_tbs_client.current_calls_notify_handler:
```

Telephone Bearer Service (TBS) The telephone bearer service is a service that typically resides on devices that can make calls, including calls from apps such as Skype, e.g. (smart)phones and PCs.

It is necessary to have `CONFIG_BT_TBS_LOG_LEVEL_DBG` enabled for using the TBS server interactively.

Using the telephone bearer service TBS can be controlled locally, or by a remote device (when in a call). For example a remote device may initiate a call to the device with the TBS server, or the TBS server may initiate a call to remote device, without a TBS_CLIENT client. The TBS implementation is capable of fully controlling any call.

```
tbs --help
tbs - Bluetooth TBS shell commands
Subcommands:
   init                        :Initialize TBS
   authorize                   :Authorize the current connection
   accept                      :Accept call <call_index>
   terminate                   :Terminate call <call_index>
   hold                        :Hold call <call_index>
   retrieve                    :Retrieve call <call_index>
   originate                   :Originate call [<instance_index>] <uri>
   join                        :Join calls <id> <id> [<id> [<id> [...]]]
   incoming                    :Simulate incoming remote call [<{instance_index,
                                gtbs}>] <local_uri> <remote_uri>
                                <remote_friendly_name>
   remote_answer               :Simulate remote answer outgoing call <call_index>
   remote_retrieve             :Simulate remote retrieve <call_index>
   remote_terminate            :Simulate remote terminate <call_index>
   remote_hold                 :Simulate remote hold <call_index>
   set_bearer_provider_name    :Set the bearer provider name [<{instance_index,
                                gtbs}>] <name>
   set_bearer_technology       :Set the bearer technology [<{instance_index,
                                gtbs}>] <technology>
   set_bearer_signal_strength  :Set the bearer signal strength [<{instance_index,
                                gtbs}>] <strength>
   set_status_flags            :Set the bearer feature and status value
                                [<{instance_index, gtbs}>] <feature_and_status>
   set_uri_scheme              :Set the URI prefix list <bearer_idx> <uri1 [uri2
                                [uri3 [...]]]>
   print_calls                 :Output all calls in the debug log
```

Example Usage

Setup

```
uart:~$ bt init
uart:~$ bt connect xx:xx:xx:xx:xx:xx public
```

When connected Answering a call for a peer device originated by a client:

```
<dbg> bt_tbs.write_call_cp: Index 0: Processing the originate opcode
<dbg> bt_tbs.originate_call: New call with call index 1
<dbg> bt_tbs.write_call_cp: Index 0: Processed the originate opcode with status success for␣
↪call index 1
uart:~$ tbs remote_answer 1
TBS succeeded for call_id: 1
```

Incoming call from a peer device, accepted by client:

```
uart:~$ tbs incoming 0 tel:123 tel:456 Peter
TBS succeeded for call_id: 4
<dbg> bt_tbs.bt_tbs_remote_incoming: New call with call index 4
<dbg> bt_tbs.write_call_cp: Index 0: Processed the accept opcode with status success for␣
↪call index 4
```

Bluetooth: Coordinated Set Identification Profile This document describes how to run the coordinated set identification functionality, both as a client and as a server. Note that in the examples below, some lines of debug have been removed to make this shorter and provide a better overview.

Set Coordinator (Client) The client will typically be a resource-rich device, such as a smartphone or a laptop. The client is able to lock and release members of a coordinated set. While the coordinated set is locked, no other clients may lock the set.

To lock a set, the client must connect to each of the set members it wants to lock. This implementation will always try to connect to all the members of the set, and at the same time. Thus if the set size is 3, then `BT_MAX_CONN` shall be at least 3.

If the locks on set members shall persists through disconnects, it is necessary to bond with the set members. If you need to bond with multiple set members, make sure that `BT_MAX_PAIRED` is correctly configured.

Using the Set Coordinator When the Bluetooth stack has been initialized (`bt init`), and a set member device has been connected, the call control client can be initialized by calling `csip_set_coordinator init`, which will start a discovery for the TBS uuids and store the handles, and optionally subscribe to all notifications (default is to subscribe to all).

Once the client has connected and discovered the handles, then it can read the set information, which is needed to identify other set members. The client can then scan for and connect to the remaining set members, and once all the members has been connected to, it can lock and release the set.

It is necessary to enable `CONFIG_BT_CSIP_SET_COORDINATOR_LOG_LEVEL_DBG` to properly use the set coordinator.

```
csip_set_coordinator --help
csip_set_coordinator - Bluetooth CSIP_SET_COORDINATOR shell commands
Subcommands:
   init              :Initialize CSIP_SET_COORDINATOR
   discover          :Run discover for CSIS on peer device [member_index]
   discover_members  :Scan for set members <set_pointer>
   lock_set          :Lock set
   release_set       :Release set
   lock              :Lock specific member [member_index]
   release           :Release specific member [member_index]
   lock_get          :Get the lock value of the specific member and instance
                      [member_index [inst_idx]]
```

Example usage

Setup

```
uart:~$ init
uart:~$ bt connect xx:xx:xx:xx:xx:xx public
```

When connected Discovering sets on a device:

```
uart:~$ csip_set_coordinator init
<dbg> bt_csip_set_coordinator.primary_discover_func: [ATTRIBUTE] handle 0x0048
<dbg> bt_csip_set_coordinator.primary_discover_func: Discover complete, found 1 instances
<dbg> bt_csip_set_coordinator.discover_func: Setup complete for 1 / 1
Found 1 sets on device
uart:~$ csip_set_coordinator discover_sets
<dbg> bt_csip_set_coordinator.SIRK
36 04 9a dc 66 3a a1 a1 |6...f:..
1d 9a 2f 41 01 73 3e 01 |../A.s>.
<dbg> bt_csip_set_coordinator.csip_set_coordinator_discover_sets_read_set_size_cb: 2
<dbg> bt_csip_set_coordinator.csip_set_coordinator_discover_sets_read_set_lock_cb: 1
<dbg> bt_csip_set_coordinator.csip_set_coordinator_discover_sets_read_rank_cb: 1
Set size 2 (pointer: 0x566fdfe8)
```

Discover set members, based on the set pointer above:

```
uart:~$ csip_set_coordinator discover_members 0x566fdfe8
<dbg> bt_csip_set_coordinator.csip_found: Found CSIS advertiser with address␣
↪34:02:86:03:86:c0 (public)
<dbg> bt_csip_set_coordinator.is_set_member: hash: 0x33ccb1, prand 0x5bfe6a
<dbg> bt_csip_set_coordinator.is_discovered: 34:02:86:03:86:c0 (public)
<dbg> bt_csip_set_coordinator.is_discovered: 34:13:e8:b3:7f:9e (public)
<dbg> bt_csip_set_coordinator.csip_found: Found member (2 / 2)
Discovered 2/2 set members
```

Lock set members:

```
uart:~$ csip_set_coordinator lock_set
<dbg> bt_csip_set_coordinator.bt_csip_set_coordinator_lock_set: Connecting to␣
↪34:02:86:03:86:c0 (public)
<dbg> bt_csip_set_coordinator.csip_set_coordinator_connected: Connected to␣
↪34:02:86:03:86:c0 (public)
<dbg> bt_csip_set_coordinator.discover_func: Setup complete for 1 / 1
<dbg> bt_csip_set_coordinator.csip_set_coordinator_lock_set_init_cb:
<dbg> bt_csip_set_coordinator.SIRK
36 04 9a dc 66 3a a1 a1 |6...f:..
1d 9a 2f 41 01 73 3e 01 |../A.s>.
<dbg> bt_csip_set_coordinator.csip_set_coordinator_discover_sets_read_set_size_cb: 2
<dbg> bt_csip_set_coordinator.csip_set_coordinator_discover_sets_read_set_lock_cb: 1
<dbg> bt_csip_set_coordinator.csip_set_coordinator_discover_sets_read_rank_cb: 2
<dbg> bt_csip_set_coordinator.csip_set_coordinator_write_lowest_rank: Locking member with␣
↪rank 1
<dbg> bt_csip_set_coordinator.notify_func: Instance 0 lock was locked
<dbg> bt_csip_set_coordinator.csip_set_coordinator_write_lowest_rank: Locking member with␣
↪rank 2
<dbg> bt_csip_set_coordinator.notify_func: Instance 0 lock was locked
Set locked
```

Release set members:

```
uart:~$ csip_set_coordinator release_set
<dbg> bt_csip_set_coordinator.csip_set_coordinator_release_highest_rank: Releasing member␣
↪with rank 2
<dbg> bt_csip_set_coordinator.notify_func: Instance 0 lock was released
<dbg> bt_csip_set_coordinator.csip_set_coordinator_release_highest_rank: Releasing member␣
↪with rank 1
<dbg> bt_csip_set_coordinator.notify_func: Instance 0 lock was released
Set released
```

Coordinated Set Member (Server) The server on devices that are part of a set, consisting of at least two devices, e.g. a pair of earbuds.

Using the Set Member

```
csip_set_member --help
csip_set_member - Bluetooth CSIP set member shell commands
Subcommands:
   register      :Initialize the service and register callbacks [size <int>]
                  [rank <int>] [not-lockable] [sirk <data>]
   lock          :Lock the set
   release       :Release the set [force]
   sirk          :Set the currently used SIRK <sirk>
   get_sirk      :Get the currently used SIRK
   sirk_rsp      :Set the response used in SIRK requests <accept, accept_enc,
                  reject, oob>
```

Example Usage

Setup

```
uart:~$ bt init
uart:~$ csip_set_member register
```

Setting a new SIRK This command can modify the currently used SIRK. To get the new RSI to advertise on air, `bt adv-data` or `bt advertise` must be called again to set the new advertising data. If `CONFIG_BT_CSIP_SET_MEMBER_NOTIFIABLE` is enabled, this will also notify connected clients.

```
uart:~$ csip_set_member sirk 00112233445566778899aabbccddeeff
SIRK updated
```

Getting the current SIRK This command can get the currently used SIRK.

```
uart:~$ csip_set_member get_sirk
SIRK
36 04 9a dc 66 3a a1 a1 |6...f:..
1d 9a 2f 41 01 73 3e 01 |../A.s>.
```

Bluetooth: Gaming Audio Profile Shell This document describes how to run the Gaming Audio Profile shell functionality. Unlike most other low-layer profiles, GMAP is a profile that exists and has a service (GMAS) on all devices. Thus both the initiator and acceptor (or central and peripheral) should do a discovery of the remote device's GMAS to see what GMAP roles and features they support.

Using the GMAP Shell When the Bluetooth stack has been initialized (`bt init`), the GMAS can be registered by by calling `gmap init`. It is also strongly suggested to enable BAP via `bap init`.

```
uart:~$ gmap --help
gmap - Bluetooth GMAP shell commands
Subcommands:
  init      : [none]
  set_role  : [ugt | ugg | bgr | bgs]
  discover  : [none]
  ac_1      : Unicast audio configuration 1
  ac_2      : Unicast audio configuration 2
  ac_3      : Unicast audio configuration 3
  ac_4      : Unicast audio configuration 4
  ac_5      : Unicast audio configuration 5
  ac_6_i    : Unicast audio configuration 6(i)
  ac_6_ii   : Unicast audio configuration 6(ii)
  ac_7_ii   : Unicast audio configuration 7(ii)
  ac_8_i    : Unicast audio configuration 8(i)
  ac_8_ii   : Unicast audio configuration 8(ii)
  ac_11_i   : Unicast audio configuration 11(i)
  ac_11_ii  : Unicast audio configuration 11(ii)
  ac_12     : Broadcast audio configuration 12
  ac_13     : Broadcast audio configuration 13
  ac_14     : Broadcast audio configuration 14
```

The `set_role` command can be used to change the role at runtime, assuming that the device supports the role (the GMAP roles depend on some BAP configurations).

Example Central with GMAP UGT role Connect and establish Gaming Audio streams using Audio Configuration (AC) 3 (some logging has been omitted for clarity):

```
uart:~$ bt init
uart:~$ bap init
uart:~$ gmap init
uart:~$ bt connect <address>
uart:~$ gatt exchange-mtu
uart:~$ bap discover
Discover complete: err 0
uart:~$ cap_initiator discover
discovery completed with CSIS
uart:~$ gmap discover
gmap discovered for conn 0x2001c7d8:
     role 0x0f
     ugg_feat 0x07
     ugt_feat 0x6f
     bgs_feat 0x01
     bgr_feat 0x03
uart:~$ bap preset sink 32_2_gr
uart:~$ bap preset source 32_2_gs
uart:~$ gmap ac_3
Starting 2 streams for AC_3
stream 0x20020060 config operation rsp_code 0 reason 0
stream 0x200204d0 config operation rsp_code 0 reason 0
stream 0x200204d0 qos operation rsp_code 0 reason 0
stream 0x20020060 qos operation rsp_code 0 reason 0
Stream 0x20020060 enabled
stream 0x200204d0 enable operation rsp_code 0 reason 0
Stream 0x200204d0 enabled
stream 0x20020060 enable operation rsp_code 0 reason 0
Stream 0x20020060 started
stream 0x200204d0 start operation rsp_code 0 reason 0
```

(continues on next page)

(continued from previous page)

```
Stream 0x200204d0 started
Unicast start completed
uart:~$ bap start_sine
Started transmitting on default_stream 0x20020060
[0]: stream 0x20020060 : TX LC3: 80 (seq_num 24800)
```

Media control for Generic Audio Content Control This document describes how to run the media control functionality, using the shell, both as a client and as a server.

The media control server consists of to parts. There is a media player (mpl) that contains the logic to handle media, and there is a media control service (mcs) that serves as a GATT-based interface to the player. The media control client consists of one part, the GATT based client (mcc).

The media control server may include an object transfer service (ots) and the media control client may include an object transfer client (otc). When these are included, a richer set of functionality is available.

The media control server and client both implement the Generic Media Control Service (only), and do not use any underlying Media Control Services.

Note that in the examples below, in many cases the debug output has been removed and long outputs may have been shortened to make the examples shorter and clearer.

Also note that this documentation does not list all shell commands, it just shows examples of some of them. The set of commands is explorable from the mcc shell and the mpl shell, by typing `mcc` or `mpl` and pressing TAB. A help text for each command can be found by doing `mcc <command> help` or `mpl <command> help`.

Overview A media player has a *name* and an *icon* that allows identification of the player for the user.

The content of the media player is structured into tracks and groups. A media player has a number of groups. A group contains tracks and other groups. (In this implementation, a group only contains tracks, not other groups.) Tracks can be divided into segments.

An active player will have a *current track*. This is the track that is playing now (if the player is playing). The current track has a *title*, a *duration* (given in hundredths of a second) and a *position* - the current position of the player within the track.

There is also a *current group* (the group of the current track), a *parent group* (the parent group of the current group) and a *next track*.

The media player is in a *state*, which will be one of playing, paused, seeking or inactive. When playing, playback happens at a given *playback speed*, and the tracks are played according to the *playing order*, which is one of the *playing orders supported*. Track changes are signalled as notifications of the *track changed* characteristic. When seeking (fast forward or fast rewind), the track position is moved according to the *seeking speed*.

The *opcodes supported* tells which operations are supported by the player by writing to the *media control point*. There is also a *search control point* that allows to search for groups and tracks according to various criteria, with the result returned in the *search results*.

Finally, the *content control ID* is used to associate the media player with an audio stream.

Media Control Client (MCP) The media control client is used to control, and to get information from, a media control server. Control is done by writing to one of the two control points, or by writing to other writable characteristics. Getting information is done by reading characteristics, or by configuring the server to send notifications.

Using the media control client Before use, the media control client must be initialized by the command `mcc init`.

To achieve a connection to the peer, the `bt` commands must be used - `bt init` followed by `bt advertise on` (or `bt connect` if the server is advertising).

When the media control client is connected to a media control server, the client can discover the server's Generic Media Control Service, by giving the command `mcc discover_mcs`. This will store the handles of the service, and (optionally, but default) subscribe to all notifications.

After discovery, the media control client can read and write characteristics, including the media control point and the search control point.

Example usage

Setup

```
uart:~$ bt init
Bluetooth initialized

uart:~$ mcc init
MCC init complete

uart:~$ bt advertise on
Advertising started
Connected: F6:58:DC:27:F3:57 (random)
```

When connected Service discovery (GMCS and included OTS):

```
uart:~$ mcc discover_mcs
<dbg> bt_mcc.bt_mcc_discover_mcs: start discovery of MCS primary service
<dbg> bt_mcc.discover_primary_func: [ATTRIBUTE] handle 0x00ae
<dbg> bt_mcc.discover_primary_func: Primary discovery complete
<dbg> bt_mcc.discover_primary_func: UUID: 2800
<dbg> bt_mcc.discover_primary_func: UUID: 8fd7
<dbg> bt_mcc.discover_primary_func: Start discovery of MCS characteristics
<dbg> bt_mcc.discover_mcs_char_func: [ATTRIBUTE] handle 0x00b0
<dbg> bt_mcc.discover_mcs_char_func: Player name, UUID: 8fa0
<dbg> bt_mcc.discover_mcs_char_func: [ATTRIBUTE] handle 0x00b2
<dbg> bt_mcc.discover_mcs_char_func: Icon Object, UUID: 8fa1
<dbg> bt_mcc.discover_mcs_char_func: [ATTRIBUTE] handle 0x00b4
<dbg> bt_mcc.discover_mcs_char_func: Icon URI, UUID: 8fa2
<dbg> bt_mcc.discover_mcs_char_func: [ATTRIBUTE] handle 0x00b6
<dbg> bt_mcc.discover_mcs_char_func: Track Changed, UUID: 8fa3
<dbg> bt_mcc.discover_mcs_char_func: Subscribing - handle: 0x00b6
[...]
<dbg> bt_mcc.discover_mcs_char_func: [ATTRIBUTE] handle 0x00ea
<dbg> bt_mcc.discover_mcs_char_func: Content Control ID, UUID: 8fb5
<dbg> bt_mcc.discover_mcs_char_func: Setup complete for MCS
<dbg> bt_mcc.discover_mcs_char_func: Start discovery of included services
<dbg> bt_mcc.discover_include_func: [ATTRIBUTE] handle 0x00af
<dbg> bt_mcc.discover_include_func: Include UUID 1825
<dbg> bt_mcc.discover_include_func: Discover include complete for MCS: OTS
<dbg> bt_mcc.discover_include_func: Start discovery of OTS characteristics
<dbg> bt_mcc.discover_otc_char_func: [ATTRIBUTE] handle 0x009c
<dbg> bt_mcc.discover_otc_char_func: OTS Features
[...]
<dbg> bt_mcc.discover_otc_char_func: [ATTRIBUTE] handle 0x00ac
<dbg> bt_mcc.discover_otc_char_func: Object Size
Discovery complete
```

(continues on next page)

(continued from previous page)

```
<dbg> bt_otc.bt_otc_register: 0
<dbg> bt_otc.bt_otc_register: L2CAP psm 0x  25 sec_level 1 registered
<dbg> bt_mcc.discover_otc_char_func: Setup complete for OTS 1 / 1
uart:~$
```

Reading characteristics - the player name and the track duration as examples:

```
uart:~$ mcc read_player_name
Player name: My media player
4d 79 20 6d 65 64 69 61  20 70 6c 61 79 65 72    |My media  player

uart:~$ mcc read_track_duration
Track duration: 6300
```

Note that the value of some characteristics may be truncated due to being too long to fit in the ATT packet. Increasing the ATT MTU may help:

```
uart:~$ mcc read_track_title
Track title: Interlude #1 (Song for

uart:~$ gatt exchange-mtu
Exchange pending
Exchange successful

uart:~$ mcc read_track_title
Track title: Interlude #1 (Song for Alison)
```

Writing characteristics - track position as an example:

The track position is where the player “is” in the current track. Read the track position, change it by writing to it, confirm by reading it again.

```
uart:~$ mcc read_track_position
Track Position: 0

uart:~$ mcc set_track_position 500
Track Position: 500

uart:~$ mcc read_track_position
Track Position: 500
```

Controlling the player via the control point:

Writing to the control point allows the client to request the server to do operations like play, pause, fast forward, change track, change group and so on. Some operations (e.g. goto track) take an argument. Currently, raw opcode values are used as input to the control point shell command. These opcode values can be found in the mpl.h header file.

Send the play command (opcode “1”), the command to go to track (opcode “52”) number three, and the pause command (opcode “2”):

```
uart:~$ mcc set_cp 1
Media State: 1
Operation: 1, result: 1
Operation: 1, param: 0

uart:~$ mcc set_cp 52 3
Track changed
Track title: Interlude #3 (Levanto Seventy)
Track duration: 7800
Track Position: 0
Current Track Object ID: 0x000000000104
```

(continues on next page)

(continued from previous page)

```
Next Track Object ID: 0x000000000105
Operation: 52, result: 1
Operation: 52, param: 3

uart:~$ mcc set_cp 2
Media State: 2
Operation: 2, result: 1
Operation: 2, param: 0
```

Using the included object transfer client When object transfer is supported by both the client and the server, a larger set of characteristics is available. These include object IDs for the various track and group objects. These IDs can be used to select and download the corresponding objects from the server's object transfer service.

Read the object ID of the current group object:

```
uart:~$ mcc read_current_group_obj_id
Current Group Object ID: 0x000000000107
```

Select the object with that ID:

```
uart:~$ mcc ots_select 0x107
Selecting object succeeded
```

Read the object's metadata:

```
uart:~$ mcc ots_read_metadata
Reading object metadata succeeded
<inf> bt_mcc: Object's meta data:
<inf> bt_mcc:        Current size    :35
<inf> bt_otc: --- Displaying 1 metadata records ---
<inf> bt_otc: Object ID: 0x000000000107
<inf> bt_otc: Object name: Joe Pass - Guitar Inte
<inf> bt_otc: Object Current Size: 35
<inf> bt_otc: Object Allocate Size: 35
<inf> bt_otc: Type: Group Obj Type
<inf> bt_otc: Properties:0x4
<inf> bt_otc:  - read permitted
```

Read the object itself:

The object received is a group object. It consists of a series of records consisting of a type (track or group) and an object ID.

```
uart:~$ mcc ots_read_current_group_object
<dbg> bt_mcc.on_group_content: Object type: 0, object  ID: 0x000000000102
<dbg> bt_mcc.on_group_content: Object type: 0, object  ID: 0x000000000103
<dbg> bt_mcc.on_group_content: Object type: 0, object  ID: 0x000000000104
<dbg> bt_mcc.on_group_content: Object type: 0, object  ID: 0x000000000105
<dbg> bt_mcc.on_group_content: Object type: 0, object  ID: 0x000000000106
```

Search The search control point takes as its input a sequence of search control items, each consisting of length, type (e.g. track name or artist name) and parameter (the track name or artist name to search for). If the result is successful, the search results are stored in an object in the object transfer service. The ID of the search results ID object can be read from the search results object ID characteristic. The search result object can then be downloaded as for the current group object above. (Note that the search results object ID is empty until a search has been done.)

This implementation has a working implementation of the search functionality interface and the server-side search control point parameter parsing. But the **actual searching is faked**, the same results are returned no matter what is searched for.

There are two commands for search, one (`mcc set_scp_raw`) allows to input the search control point parameter (the sequence of search control items) as a string. The other (`mcc set_scp_ioptest`) does preset IOP test searches and takes the round number of the IOP search control point test as a parameter.

Before the search, the search results object ID is empty

```
uart:~$ mcc read_search_results_obj_id
Search Results Object ID: 0x000000000000
<dbg> bt_mcc.mcc_read_search_results_obj_id_cb: Zero-length Search Results Object ID
```

Run the search corresponding to the fourth round of the IOP test:

The search control point parameter generated by this command and parameter has one search control item. The length field (first octet) is 16 (0x10). (The length of the length field itself is not included.) The type field (second octet) is 0x04 (search for a group name). The parameter (the group name to search for) is “TSPX_Group_Name”.

```
uart:~$ mcc set_scp_ioptest 4
Search string:
00000000: 10 04 54 53 50 58 5f 47  72 6f 75 70 5f 4e 61 6d |..TSPX_G roup_Nam|
00000010: 65                                              |e                |
Search control point notification result code: 1
Search Results Object ID: 0x000000000107
Search Control Point set
```

After the successful search, the search results object ID has a value:

```
uart:~$ mcc read_search_results_obj_id
Search Results Object ID: 0x000000000107
```

Media Control Service (MCS) The media control service (mcs) and the associated media player (mpl) typically reside on devices that can provide access to, and serve, media content, like PCs and smartphones.

As mentioned above, the media player (mpl) has the player logic, while the media control service (mcs) has the GATT-based interface. This separation is done so that the media player can also be used without the GATT-based interface.

Using the media control service and the media player The media control service and the media player are in general controlled remotely, from the media control client.

Before use, the media control client must be initialized by the command `mpl init`.

As for the client, the `bt` commands are used for connecting - `bt init` followed by `bt connect <address> <address type>` (or `bt advertise on` if the server is advertising).

Example Usage

Setup

```
uart:~$ bt init
Bluetooth initialized

uart:~$ mpl init
```

(continues on next page)

(continued from previous page)

```
[Large amounts of debug output]

uart:~$ bt connect F9:33:3B:67:D2:A7 (random)
Connection pending
Connected: F9:33:3B:67:D2:A7 (random)
```

When connected Control is done from the client.

The server will give debug output related to the various operations performed by the client.

Example: Debug output by the server when the client gives the "next track" command:

```
[00:13:29.932,373] <dbg> bt_mcs.control_point_write: Opcode: 49
[00:13:29.932,403] <dbg> bt_mpl.mpl_operation_set: opcode: 49, param: 536880068
[00:13:29.932,403] <dbg> bt_mpl.paused_state_operation_handler: Operation opcode: 49
[00:13:29.932,495] <dbg> bt_mpl.do_next_track: Track ID before: 0x000000000104
[00:13:29.932,586] <dbg> bt_mpl.do_next_track: Track ID after: 0x000000000105
[00:13:29.932,617] <dbg> bt_mcs.mpl_track_changed_cb: Notifying track change
[00:13:29.932,708] <dbg> bt_mcs.mpl_track_title_cb: Notifying track title: Interlude #4␣
↪(Vesper Dreams)
[00:13:29.932,800] <dbg> bt_mcs.mpl_track_duration_cb: Notifying track duration: 13500
[00:13:29.932,861] <dbg> bt_mcs.mpl_track_position_cb: Notifying track position: 0
[00:13:29.933,044] <dbg> bt_mcs.mpl_current_track_id_cb: Notifying current track ID:␣
↪0x000000000105
[00:13:29.933,258] <dbg> bt_mcs.mpl_next_track_id_cb: Notifying next track ID:␣
↪0x000000000106
[00:13:29.933,380] <dbg> bt_mcs.mpl_operation_cb: Notifying control point - opcode: 49,␣
↪result: 1
```

Some server commands are available. These commands force notifications of the various characteristics, for testing that the client receives notifications. The values sent in the notifications caused by these testing commands are independent of the media player, so they do not correspond the actual values of the characteristics nor to the actual state of the media player.

Example: Force (fake value) notification of the track duration:

```
uart:~$ mpl duration_changed_cb
[00:15:17.491,058] <dbg> bt_mcs.mpl_track_duration_cb: Notifying track duration: 12000
```

Bluetooth: Telephone and Media Audio Profile Shell This document describes how to run the Telephone and Media Audio Profile functionality. Unlike most other low-layer profiles, TMAP is a profile that exists and has a service (TMAS) on all devices. Thus both the initiator and acceptor (or central and peripheral) should do a discovery of the remote device's TMAS to see what TMAP roles they support.

Using the TMAP Shell When the Bluetooth stack has been initialized (`bt init`), the TMAS can be registered by calling `tmap init`.

```
tmap --help
tmap - Bluetooth TMAP shell commands
Subcommands:
  init          :Initialize and register the TMAS
  discover      :Discover TMAS on remote device
```

Bluetooth: Public Broadcast Profile Shell This document describes how to run the Public Broadcast Profile functionality. PBP does not have an associated service. Its purpose is to enable

a faster, more efficient discovery of Broadcast Sources that are transmitting audio with commonly used codec configurations.

Using the PBP Shell When the Bluetooth stack has been initialized (`bt init`), the Public Broadcast Profile is ready to run. To set the Public Broadcast Announcement features call `pbp set_features`.

```
pbp --help
pbp - Bluetooth PBP shell commands
Subcommands:
  set_features    :Set the Public Broadcast Announcement features
```

Bluetooth LE Host

Bluetooth standard services

Battery Service

group `bt_bas`

Battery Service (BAS)

[Experimental] Users should note that the APIs can change as a part of ongoing development.

Functions

uint8_t **bt_bas_get_battery_level**(void)

Read battery level value.

Read the characteristic value of the battery level

Returns
The battery level in percent.

int **bt_bas_set_battery_level**(uint8_t level)

Update battery level value.

Update the characteristic value of the battery level This will send a GATT notification to all current subscribers.

Parameters

- `level` – The battery level in percent.

Returns
Zero in case of success and error code in case of error.

Heart Rate Service

group `bt_hrs`

Heart Rate Service (HRS)

[Experimental] Users should note that the APIs can change as a part of ongoing development.

Functions

int **bt_hrs_cb_register**(struct *bt_hrs_cb* *cb)

Heart rate service callback register.

This function will register callbacks that will be called in certain events related to Heart rate service.

Parameters

- **cb** – Pointer to callbacks structure. Must point to memory that remains valid until unregistered.

Returns
0 on success

Returns
-EINVAL in case cb is NULL

int **bt_hrs_cb_unregister**(struct *bt_hrs_cb* *cb)

Heart rate service callback unregister.

This function will unregister callback from Heart rate service.

Parameters

- **cb** – Pointer to callbacks structure

Returns
0 on success

Returns
-EINVAL in case cb is NULL

Returns
-ENOENT in case the cb was not found in registered callbacks

int **bt_hrs_notify**(uint16_t heartrate)

Notify heart rate measurement.

This will send a GATT notification to all current subscribers.

Parameters

- **heartrate** – The heartrate measurement in beats per minute.

Returns
Zero in case of success and error code in case of error.

struct **bt_hrs_cb**

#include <*hrs.h*> Heart rate service callback structure.

Public Members

void (***ntf_changed**)(bool enabled)

Heart rate notifications changed.

Param enabled
Flag that is true if notifications were enabled, false if they were disabled.

Immediate Alert Service

group `bt_ias`

Immediate Alert Service (IAS)

[Experimental] Users should note that the APIs can change as a part of ongoing development.

Defines

`BT_IAS_CB_DEFINE`(_name)

Register a callback structure for immediate alert events.

Parameters

- `_name` – Name of callback structure.

Enums

enum `bt_ias_alert_lvl`

Values:

enumerator `BT_IAS_ALERT_LVL_NO_ALERT`

No alerting should be done on device.

enumerator `BT_IAS_ALERT_LVL_MILD_ALERT`

Device shall alert.

enumerator `BT_IAS_ALERT_LVL_HIGH_ALERT`

Device should alert in strongest possible way.

Functions

int `bt_ias_local_alert_stop`(void)

Method for stopping alert locally.

Returns

Zero in case of success and error code in case of error.

int `bt_ias_client_alert_write`(struct bt_conn *conn, enum *bt_ias_alert_lvl*)

Set alert level.

Parameters

- `conn` – Bluetooth connection object
- `bt_ias_alert_lvl` – Level of alert to write

Returns

Zero in case of success and error code in case of error.

int `bt_ias_discover`(struct bt_conn *conn)

Discover Immediate Alert Service.

Parameters

- `conn` – Bluetooth connection object

Returns
Zero in case of success and error code in case of error.

int `bt_ias_client_cb_register`(const struct *bt_ias_client_cb* *cb)
Register Immediate Alert Client callbacks.

Parameters
- `cb` – The callback structure

Returns
Zero in case of success and error code in case of error.

struct `bt_ias_cb`
#include <ias.h> Immediate Alert Service callback structure.

Public Members

void (*`no_alert`)(void)
Callback function to stop alert.

This callback is called when peer commands to disable alert.

void (*`mild_alert`)(void)
Callback function for alert level value.

This callback is called when peer commands to alert.

void (*`high_alert`)(void)
Callback function for alert level value.

This callback is called when peer commands to alert in the strongest possible way.

struct `bt_ias_client_cb`
#include <ias.h>

Public Members

void (*`discover`)(struct bt_conn *conn, int err)
Callback function for bt_ias_discover.

This callback is called when discovery procedure is complete.
Param conn
Bluetooth connection object.
Param err
0 on success, ATT error or negative errno otherwise

Object Transfer Service

group `bt_ots`
Object Transfer Service (OTS)

[Experimental] Users should note that the APIs can change as a part of ongoing development.

Defines

BT_OTS_OBJ_ID_SIZE
Size of OTS object ID (in bytes).

BT_OTS_OBJ_ID_MIN
Minimum allowed value for object ID (except ID for directory listing)

BT_OTS_OBJ_ID_MAX
Maximum allowed value for object ID (except ID for directory listing)

OTS_OBJ_ID_DIR_LIST
ID of the Directory Listing Object.

BT_OTS_OBJ_ID_MASK
Mask for OTS object IDs, preserving the 48 bits.

BT_OTS_OBJ_ID_STR_LEN
Length of OTS object ID string (in bytes).

BT_OTS_OBJ_SET_PROP_DELETE(prop)
Set *BT_OTS_OBJ_PROP_DELETE* property.

Parameters

- **prop** – Object properties.

BT_OTS_OBJ_SET_PROP_EXECUTE(prop)
Set *BT_OTS_OBJ_PROP_EXECUTE* property.

Parameters

- **prop** – Object properties.

BT_OTS_OBJ_SET_PROP_READ(prop)
Set *BT_OTS_OBJ_PROP_READ* property.

Parameters

- **prop** – Object properties.

BT_OTS_OBJ_SET_PROP_WRITE(prop)
Set *BT_OTS_OBJ_PROP_WRITE* property.

Parameters

- **prop** – Object properties.

BT_OTS_OBJ_SET_PROP_APPEND(prop)
Set *BT_OTS_OBJ_PROP_APPEND* property.

Parameters

- **prop** – Object properties.

BT_OTS_OBJ_SET_PROP_TRUNCATE(prop)
Set *BT_OTS_OBJ_PROP_TRUNCATE* property.

Parameters

- **prop** – Object properties.

BT_OTS_OBJ_SET_PROP_PATCH(prop)
Set *BT_OTS_OBJ_PROP_PATCH* property.

Parameters

- **prop** – Object properties.

BT_OTS_OBJ_SET_PROP_MARKED(prop)
Set *BT_OTS_OBJ_SET_PROP_MARKED* property.

Parameters

- **prop** – Object properties.

BT_OTS_OBJ_GET_PROP_DELETE(prop)
Get *BT_OTS_OBJ_PROP_DELETE* property.

Parameters

- **prop** – Object properties.

BT_OTS_OBJ_GET_PROP_EXECUTE(prop)
Get *BT_OTS_OBJ_PROP_EXECUTE* property.

Parameters

- **prop** – Object properties.

BT_OTS_OBJ_GET_PROP_READ(prop)
Get *BT_OTS_OBJ_PROP_READ* property.

Parameters

- **prop** – Object properties.

BT_OTS_OBJ_GET_PROP_WRITE(prop)
Get *BT_OTS_OBJ_PROP_WRITE* property.

Parameters

- **prop** – Object properties.

BT_OTS_OBJ_GET_PROP_APPEND(prop)
Get *BT_OTS_OBJ_PROP_APPEND* property.

Parameters

- **prop** – Object properties.

BT_OTS_OBJ_GET_PROP_TRUNCATE(prop)
Get *BT_OTS_OBJ_PROP_TRUNCATE* property.

Parameters

- **prop** – Object properties.

BT_OTS_OBJ_GET_PROP_PATCH(prop)
Get *BT_OTS_OBJ_PROP_PATCH* property.

Parameters

- **prop** – Object properties.

BT_OTS_OBJ_GET_PROP_MARKED(prop)
Get *BT_OTS_OBJ_PROP_MARKED* property.

Parameters

- **prop** – Object properties.

BT_OTS_OACP_SET_FEAT_CREATE(feat)
Set *BT_OTS_OACP_SET_FEAT_CREATE* feature.

Parameters

- **feat** – OTS features.

BT_OTS_OACP_SET_FEAT_DELETE(feat)
Set *BT_OTS_OACP_FEAT_DELETE* feature.

Parameters

- **feat** – OTS features.

BT_OTS_OACP_SET_FEAT_CHECKSUM(feat)
Set *BT_OTS_OACP_FEAT_CHECKSUM* feature.

Parameters

- **feat** – OTS features.

BT_OTS_OACP_SET_FEAT_EXECUTE(feat)
Set *BT_OTS_OACP_FEAT_EXECUTE* feature.

Parameters

- **feat** – OTS features.

BT_OTS_OACP_SET_FEAT_READ(feat)
Set *BT_OTS_OACP_FEAT_READ* feature.

Parameters

- **feat** – OTS features.

BT_OTS_OACP_SET_FEAT_WRITE(feat)
Set *BT_OTS_OACP_FEAT_WRITE* feature.

Parameters

- **feat** – OTS features.

BT_OTS_OACP_SET_FEAT_APPEND(feat)
Set *BT_OTS_OACP_FEAT_APPEND* feature.

Parameters

- **feat** – OTS features.

BT_OTS_OACP_SET_FEAT_TRUNCATE(feat)
Set *BT_OTS_OACP_FEAT_TRUNCATE* feature.

Parameters

- **feat** – OTS features.

BT_OTS_OACP_SET_FEAT_PATCH(feat)
Set *BT_OTS_OACP_FEAT_PATCH* feature.

Parameters

- **feat** – OTS features.

BT_OTS_OACP_SET_FEAT_ABORT(feat)
Set *BT_OTS_OACP_FEAT_ABORT* feature.

Parameters

- **feat** – OTS features.

BT_OTS_OACP_GET_FEAT_CREATE(feat)
Get *BT_OTS_OACP_FEAT_CREATE* feature.

Parameters

- **feat** – OTS features.

BT_OTS_OACP_GET_FEAT_DELETE(feat)
Get *BT_OTS_OACP_FEAT_DELETE* feature.

Parameters

- **feat** – OTS features.

BT_OTS_OACP_GET_FEAT_CHECKSUM(feat)
Get *BT_OTS_OACP_FEAT_CHECKSUM* feature.

Parameters

- **feat** – OTS features.

BT_OTS_OACP_GET_FEAT_EXECUTE(feat)
Get *BT_OTS_OACP_FEAT_EXECUTE* feature.

Parameters

- **feat** – OTS features.

BT_OTS_OACP_GET_FEAT_READ(feat)
Get *BT_OTS_OACP_FEAT_READ* feature.

Parameters

- **feat** – OTS features.

BT_OTS_OACP_GET_FEAT_WRITE(feat)
Get *BT_OTS_OACP_FEAT_WRITE* feature.

Parameters

- **feat** – OTS features.

BT_OTS_OACP_GET_FEAT_APPEND(feat)
Get *BT_OTS_OACP_FEAT_APPEND* feature.

Parameters

- **feat** – OTS features.

BT_OTS_OACP_GET_FEAT_TRUNCATE(feat)
Get *BT_OTS_OACP_FEAT_TRUNCATE* feature.

Parameters

- **feat** – OTS features.

BT_OTS_OACP_GET_FEAT_PATCH(feat)
Get *BT_OTS_OACP_FEAT_PATCH* feature.

Parameters

- **feat** – OTS features.

BT_OTS_OACP_GET_FEAT_ABORT(feat)
Get *BT_OTS_OACP_FEAT_ABORT* feature.

Parameters

- **feat** – OTS features.

BT_OTS_OLCP_SET_FEAT_GO_TO(feat)
Set *BT_OTS_OLCP_FEAT_GO_TO* feature.

Parameters

- **feat** – OTS features.

BT_OTS_OLCP_SET_FEAT_ORDER(feat)
Set *BT_OTS_OLCP_FEAT_ORDER* feature.

Parameters

- **feat** – OTS features.

BT_OTS_OLCP_SET_FEAT_NUM_REQ(feat)
Set *BT_OTS_OLCP_FEAT_NUM_REQ* feature.

Parameters

- **feat** – OTS features.

BT_OTS_OLCP_SET_FEAT_CLEAR(feat)
Set *BT_OTS_OLCP_FEAT_CLEAR* feature.

Parameters

- **feat** – OTS features.

BT_OTS_OLCP_GET_FEAT_GO_TO(feat)
Get *BT_OTS_OLCP_GET_FEAT_GO_TO* feature.

Parameters

- **feat** – OTS features.

BT_OTS_OLCP_GET_FEAT_ORDER(feat)
Get *BT_OTS_OLCP_GET_FEAT_ORDER* feature.

Parameters

- **feat** – OTS features.

BT_OTS_OLCP_GET_FEAT_NUM_REQ(feat)
Get *BT_OTS_OLCP_GET_FEAT_NUM_REQ* feature.

Parameters

- **feat** – OTS features.

BT_OTS_OLCP_GET_FEAT_CLEAR(feat)
Get *BT_OTS_OLCP_GET_FEAT_CLEAR* feature.

Parameters

- **feat** – OTS features.

BT_OTS_DATE_TIME_FIELD_SIZE

BT_OTS_STOP

BT_OTS_CONTINUE

Typedefs

typedef int (*bt_ots_client_dirlisting_cb)(struct *bt_ots_obj_metadata* *meta)

Directory listing object metadata callback.

If a directory listing is decoded using *bt_ots_client_decode_dirlisting()*, this callback will be called for each object in the directory listing.

Param meta
The metadata of the decoded object

Return
int BT_OTS_STOP or BT_OTS_CONTINUE. BT_OTS_STOP can be used to stop the decoding.

Enums

Properties of an OTS object.

Values:

enumerator BT_OTS_OBJ_PROP_DELETE = 0

Bit 0 Deletion of this object is permitted.

enumerator BT_OTS_OBJ_PROP_EXECUTE = 1

Bit 1 Execution of this object is permitted.

enumerator BT_OTS_OBJ_PROP_READ = 2

Bit 2 Reading this object is permitted.

enumerator BT_OTS_OBJ_PROP_WRITE = 3

Bit 3 Writing data to this object is permitted.

enumerator BT_OTS_OBJ_PROP_APPEND = 4

Bit 4 Appending data to this object is permitted.

```
Appending data increases its Allocated Size.
```

enumerator BT_OTS_OBJ_PROP_TRUNCATE = 5

Bit 5 Truncation of this object is permitted.

enumerator BT_OTS_OBJ_PROP_PATCH = 6

Bit 6 Patching this object is permitted.

```
Patching this object overwrites some of
the object's existing contents.
```

enumerator BT_OTS_OBJ_PROP_MARKED = 7

Bit 7 This object is a marked object.

Object Action Control Point Feature bits.

Values:

enumerator **BT_OTS_OACP_FEAT_CREATE** = 0
Bit 0 OACP Create Op Code Supported.

enumerator **BT_OTS_OACP_FEAT_DELETE** = 1
Bit 1 OACP Delete Op Code Supported

enumerator **BT_OTS_OACP_FEAT_CHECKSUM** = 2
Bit 2 OACP Calculate Checksum Op Code Supported.

enumerator **BT_OTS_OACP_FEAT_EXECUTE** = 3
Bit 3 OACP Execute Op Code Supported.

enumerator **BT_OTS_OACP_FEAT_READ** = 4
Bit 4 OACP Read Op Code Supported.

enumerator **BT_OTS_OACP_FEAT_WRITE** = 5
Bit 5 OACP Write Op Code Supported.

enumerator **BT_OTS_OACP_FEAT_APPEND** = 6
Bit 6 Appending Additional Data to Objects Supported

enumerator **BT_OTS_OACP_FEAT_TRUNCATE** = 7
Bit 7 Truncation of Objects Supported.

enumerator **BT_OTS_OACP_FEAT_PATCH** = 8
Bit 8 Patching of Objects Supported

enumerator **BT_OTS_OACP_FEAT_ABORT** = 9
Bit 9 OACP Abort Op Code Supported.

enum **bt_ots_oacp_write_op_mode**

Values:

enumerator **BT_OTS_OACP_WRITE_OP_MODE_NONE** = 0

enumerator **BT_OTS_OACP_WRITE_OP_MODE_TRUNCATE** = *BIT*(1)

Object List Control Point Feature bits.

Values:

enumerator **BT_OTS_OLCP_FEAT_GO_TO** = 0
Bit 0 OLCP Go To Op Code Supported.

enumerator BT_OTS_OLCP_FEAT_ORDER = 1
Bit 1 OLCP Order Op Code Supported.

enumerator BT_OTS_OLCP_FEAT_NUM_REQ = 2
Bit 2 OLCP Request Number of Objects Op Code Supported.

enumerator BT_OTS_OLCP_FEAT_CLEAR = 3
Bit 3 OLCP Clear Marking Op Code Supported.

Object metadata request bit field values.

Values:

enumerator BT_OTS_METADATA_REQ_NAME = *BIT*(0)
Request object name.

enumerator BT_OTS_METADATA_REQ_TYPE = *BIT*(1)
Request object type.

enumerator BT_OTS_METADATA_REQ_SIZE = *BIT*(2)
Request object size.

enumerator BT_OTS_METADATA_REQ_CREATED = *BIT*(3)
Request object first created time.

enumerator BT_OTS_METADATA_REQ_MODIFIED = *BIT*(4)
Request object last modified time.

enumerator BT_OTS_METADATA_REQ_ID = *BIT*(5)
Request object ID.

enumerator BT_OTS_METADATA_REQ_PROPS = *BIT*(6)
Request object properties.

enumerator BT_OTS_METADATA_REQ_ALL = 0x7F
Request all object metadata.

Functions

int **bt_ots_obj_add**(struct bt_ots *ots, const struct *bt_ots_obj_add_param* *param)
Add an object to the OTS instance.

This function adds an object to the OTS database. When the object is being added, a callback obj_created() is called to notify the user about a new object ID.

Parameters

- **ots** – OTS instance.
- **param** – Object addition parameters.

Returns
ID of created object in case of success.

Returns
negative value in case of error.

int **bt_ots_obj_delete**(struct bt_ots *ots, uint64_t id)

Delete an object from the OTS instance.

This function deletes an object from the OTS database. When the object is deleted a callback obj_deleted() is called to notify the user about this event. At this point, it is possible to free allocated buffer for object data.

Parameters

- **ots** – OTS instance.
- **id** – ID of the object to be deleted (uint48).

Returns
0 in case of success or negative value in case of error.

void ***bt_ots_svc_decl_get**(struct bt_ots *ots)

Get the service declaration attribute.

This function is enabled for CONFIG_BT_OTS_SECONDARY_SVC configuration. The first service attribute can be included in any other GATT service.

Parameters

- **ots** – OTS instance.

Returns
The first OTS attribute instance.

int **bt_ots_init**(struct bt_ots *ots, struct *bt_ots_init_param* *ots_init)

Initialize the OTS instance.

Parameters

- **ots** – OTS instance.
- **ots_init** – OTS initialization descriptor.

Returns
0 in case of success or negative value in case of error.

struct bt_ots ***bt_ots_free_instance_get**(void)

Get a free instance of OTS from the pool.

Returns
OTS instance in case of success or NULL in case of error.

int **bt_ots_client_register**(struct *bt_ots_client* *ots_inst)

Register an Object Transfer Service Instance.

Register an Object Transfer Service instance discovered on the peer. Call this function when an OTS instance is discovered (discovery is to be handled by the higher layer).

Parameters

- **ots_inst** – **[in]** Discovered OTS instance.

Returns
int 0 if success, ERRNO on failure.

int **bt_ots_client_unregister**(uint8_t index)

Unregister an Object Transfer Service Instance.

Unregister an Object Transfer Service instance when disconnect from the peer. Call this function when an ACL using OTS instance is disconnected.

Parameters

- **index** – **[in]** Index of OTS instance.

Returns

int 0 if success, ERRNO on failure.

uint8_t **bt_ots_client_indicate_handler**(struct bt_conn *conn, struct *bt_gatt_subscribe_params* *params, const void *data, uint16_t length)

OTS Indicate Handler function.

Set this function as callback for indicate handler when discovering OTS.

Parameters

- **conn** – Connection object. May be NULL, indicating that the peer is being unpaired.
- **params** – Subscription parameters.
- **data** – Attribute value data. If NULL then subscription was removed.
- **length** – Attribute value length.

int **bt_ots_client_read_feature**(struct *bt_ots_client* *otc_inst, struct bt_conn *conn)

Read the OTS feature characteristic.

Parameters

- **otc_inst** – Pointer to the OTC instance.
- **conn** – Pointer to the connection object.

Returns

int 0 if success, ERRNO on failure.

int **bt_ots_client_select_id**(struct *bt_ots_client* *otc_inst, struct bt_conn *conn, uint64_t obj_id)

Select an object by its Object ID.

Parameters

- **otc_inst** – Pointer to the OTC instance.
- **conn** – Pointer to the connection object.
- **obj_id** – Object's ID.

Returns

int 0 if success, ERRNO on failure.

int **bt_ots_client_select_first**(struct *bt_ots_client* *otc_inst, struct bt_conn *conn)

Select the first object.

Parameters

- **otc_inst** – Pointer to the OTC instance.
- **conn** – Pointer to the connection object.

Returns

int 0 if success, ERRNO on failure.

int **bt_ots_client_select_last**(struct *bt_ots_client* *otc_inst, struct bt_conn *conn)

Select the last object.

Parameters

- **otc_inst** – Pointer to the OTC instance.
- **conn** – Pointer to the connection object.

Returns

int 0 if success, ERRNO on failure.

int **bt_ots_client_select_next**(struct *bt_ots_client* *otc_inst, struct bt_conn *conn)

Select the next object.

Parameters

- **otc_inst** – Pointer to the OTC instance.
- **conn** – Pointer to the connection object.

Returns

int 0 if success, ERRNO on failure.

int **bt_ots_client_select_prev**(struct *bt_ots_client* *otc_inst, struct bt_conn *conn)

Select the previous object.

Parameters

- **otc_inst** – Pointer to the OTC instance.
- **conn** – Pointer to the connection object.

Returns

int 0 if success, ERRNO on failure.

int **bt_ots_client_read_object_metadata**(struct *bt_ots_client* *otc_inst, struct bt_conn *conn, uint8_t metadata)

Read the metadata of the current object.

The metadata are returned in the obj_metadata_read() callback.

Parameters

- **otc_inst** – Pointer to the OTC instance.
- **conn** – Pointer to the connection object.
- **metadata** – Bitfield (BT_OTS_METADATA_REQ_*) of the metadata to read.

Returns

int 0 if success, ERRNO on failure.

int **bt_ots_client_read_object_data**(struct *bt_ots_client* *otc_inst, struct bt_conn *conn)

Read the data of the current selected object.

This will trigger an OACP read operation for the current size of the object with a 0 offset and then expect receiving the content via the L2CAP CoC.

The data of the object are returned in the obj_data_read() callback.

Parameters

- **otc_inst** – Pointer to the OTC instance.
- **conn** – Pointer to the connection object.

Returns

int 0 if success, ERRNO on failure.

int **bt_ots_client_write_object_data**(struct *bt_ots_client* *otc_inst, struct bt_conn *conn, const void *buf, size_t len, off_t offset, enum *bt_ots_oacp_write_op_mode* mode)

Write the data of the current selected object.

This will trigger an OACP write operation for the current object with a specified offset and then expect transferring the content via the L2CAP CoC.

The length of the data written to object is returned in the obj_data_written() callback.

Parameters

- **otc_inst** – Pointer to the OTC instance.
- **conn** – Pointer to the connection object.
- **buf** – Pointer to the data buffer to be written.
- **len** – Size of data.
- **offset** – Offset to write, usually 0.
- **mode** – Mode Parameter for OACP Write Op Code. See *bt_ots_oacp_write_op_mode*.

Returns

int 0 if success, ERRNO on failure.

int **bt_ots_client_get_object_checksum**(struct *bt_ots_client* *otc_inst, struct bt_conn *conn, off_t offset, size_t len)

Get the checksum of the current selected object.

This will trigger an OACP calculate checksum operation for the current object with a specified offset and length.

The checksum goes to OACP IND and obj_checksum_calculated() callback.

Parameters

- **otc_inst** – Pointer to the OTC instance.
- **conn** – Pointer to the connection object.
- **offset** – Offset to calculate, usually 0.
- **len** – Len of data to calculate checksum for. May be less than the current object's size, but shall not be larger.

Returns

int 0 if success, ERRNO on failure.

int **bt_ots_client_decode_dirlisting**(uint8_t *data, uint16_t length, *bt_ots_client_dirlisting_cb* cb)

Decode Directory Listing object into object metadata.

If the Directory Listing object contains multiple objects, then the callback will be called for each of them.

Parameters

- **data** – The data received for the directory listing object.
- **length** – Length of the data.
- **cb** – The callback that will be called for each object.

static inline int **bt_ots_obj_id_to_str**(uint64_t obj_id, char *str, size_t len)

Converts binary OTS Object ID to string.

Parameters

- **obj_id** – Object ID.
- **str** – Address of user buffer with enough room to store formatted string containing binary Object ID.
- **len** – Length of data to be copied to user string buffer. Refer to BT_OTS_OBJ_ID_STR_LEN about recommended value.

Returns
Number of successfully formatted bytes from binary ID.

void **bt_ots_metadata_display**(struct *bt_ots_obj_metadata* *metadata, uint16_t count)

Displays one or more object metadata as text with LOG_INF.

Parameters

- **metadata** – Pointer to the first (or only) metadata in an array.
- **count** – Number of metadata objects to display information of.

struct **bt_ots_obj_type**

#include <*ots.h*> Type of an OTS object.

struct **bt_ots_obj_size**

#include <*ots.h*> Descriptor for OTS Object Size parameter.

Public Members

uint32_t **cur**

Current Size.

uint32_t **alloc**

Allocated Size.

struct **bt_ots_feat**

#include <*ots.h*> Features of the OTS.

struct **bt_ots_date_time**

#include <*ots.h*> Date and Time structure.

struct **bt_ots_obj_metadata**

#include <*ots.h*> Metadata of an OTS object.

Used by the server as a descriptor for OTS object initialization. Used by the client to present object metadata to the application.

Public Members

struct *bt_ots_obj_type* **type**

Object Type.

struct *bt_ots_obj_size* **size**

Object Size.

uint32_t **props**

Object Properties.

struct **bt_ots_obj_add_param**

#include <*ots.h*> Descriptor for OTS object addition.

Public Members

uint32_t **size**

Object size to allocate.

struct *bt_ots_obj_type* **type**

Object type.

struct **bt_ots_obj_created_desc**

#include <*ots.h*> Descriptor for OTS created object.

Descriptor for OTS object created by the application. This descriptor is returned by *bt_ots_cb::obj_created* callback which contains further documentation on distinguishing between server and client object creation.

Public Members

char ***name**

Object name.

The object name as a NULL terminated string.

When the server creates a new object the name shall be > 0 and <= BT_OTS_OBJ_MAX_NAME_LEN When the client creates a new object the name shall be an empty string

struct *bt_ots_obj_size* **size**

Object size.

bt_ots_obj_size::alloc shall be >= *bt_ots_obj_add_param::size*

When the server creates a new object *bt_ots_obj_size::cur* shall be <= *bt_ots_obj_add_param::size* When the client creates a new object *bt_ots_obj_size::cur* shall be 0

uint32_t **props**

Object properties.

struct **bt_ots_cb**

#include <*ots.h*> OTS callback structure.

Public Members

int (*obj_created)(struct bt_ots *ots, struct bt_conn *conn, uint64_t id, const struct *bt_ots_obj_add_param* *add_param, struct *bt_ots_obj_created_desc* *created_desc)

Object created callback.

This callback is called whenever a new object is created. Application can reject this request by returning an error when it does not have necessary resources to hold this new object. This callback is also triggered when the server creates a new object with *bt_ots_obj_add()* API.

Param ots
OTS instance.

Param conn
The connection that is requesting object creation or NULL if object is created by *bt_ots_obj_add()*.

Param id
Object ID.

Param add_param
Object creation requested parameters.

Param created_desc
Created object descriptor that shall be filled by the receiver of this callback.

Return
0 in case of success or negative value in case of error.

Return
-ENOTSUP if object type is not supported

Return
-ENOMEM if no available space for new object.

Return
-EINVAL if an invalid parameter is provided

Return
other negative values are treated as a generic operation failure

int (*obj_deleted)(struct bt_ots *ots, struct bt_conn *conn, uint64_t id)

Object deleted callback.

This callback is called whenever an object is deleted. It is also triggered when the server deletes an object with *bt_ots_obj_delete()* API.

Param ots
OTS instance.

Param conn
The connection that deleted the object or NULL if this request came from the server.

Param id
Object ID.

Retval When
an error is indicated by using a negative value, the object delete procedure is aborted and a corresponding failed status is returned to the client.

Return
0 in case of success.

Return
-EBUSY if the object is locked. This is generally not expected to be returned by the application as the OTS layer tracks object accesses. An object locked status is returned to the client.

Return
Other negative values in case of error. A generic operation failed status is returned to the client.

void (*obj_selected)(struct bt_ots *ots, struct bt_conn *conn, uint64_t id)

Object selected callback.

This callback is called on successful object selection.

Param ots
OTS instance.

Param conn
The connection that selected new object.

Param id
Object ID.

ssize_t (*obj_read)(struct bt_ots *ots, struct bt_conn *conn, uint64_t id, void **data, size_t len, off_t offset)

Object read callback.

This callback is called multiple times during the Object read operation. OTS module will keep requesting successive Object fragments from the application until the read operation is completed. The end of read operation is indicated by NULL data parameter.

Param ots
OTS instance.

Param conn
The connection that read object.

Param id
Object ID.

Param data
In: NULL once the read operations is completed. Out: Next chunk of data to be sent.

Param len
Remaining length requested by the client.

Param offset
Object data offset.

Return
Data length to be sent via data parameter. This value shall be smaller or equal to the len parameter.

Return
Negative value in case of an error.

ssize_t (*obj_write)(struct bt_ots *ots, struct bt_conn *conn, uint64_t id, const void *data, size_t len, off_t offset, size_t rem)

Object write callback.

This callback is called multiple times during the Object write operation. OTS module will keep providing successive Object fragments to the application until the write operation is completed. The offset and length of each write fragment is validated by the OTS module to be within the allocated size of the object. The remaining length indicates data length remaining to be written and will decrease each write iteration until it reaches 0 in the last write fragment.

Param ots
OTS instance.

Param conn
The connection that wrote object.

Param id
Object ID.

Param data
Next chunk of data to be written.

Param len
Length of the current chunk of data in the buffer.

Param offset
Object data offset.

Param rem
Remaining length in the write operation.
Return
Number of bytes written in case of success, if the number of bytes written does not match len, -EIO is returned to the L2CAP layer.
Return
A negative value in case of an error.
Return
-EINPROGRESS has a special meaning and is unsupported at the moment. It should not be returned.

void (*`obj_name_written`)(struct bt_ots *ots, struct bt_conn *conn, uint64_t id, const char *cur_name, const char *new_name)

Object name written callback.

This callback is called when the object name is written. This is a notification to the application that the object name will be updated by the OTS service implementation.
Param ots
OTS instance.
Param conn
The connection that wrote object name.
Param id
Object ID.
Param cur_name
Current object name.
Param new_name
New object name.

int (*`obj_cal_checksum`)(struct bt_ots *ots, struct bt_conn *conn, uint64_t id, off_t offset, size_t len, void **data)

Object Calculate checksum callback.

This callback is called when the OACP Calculate Checksum procedure is performed. Because object data is opaque to OTS, the application is the only one who knows where data is and should return pointer of actual object data.
Param ots
[in] OTS instance.
Param conn
[in] The connection that wrote object.
Param id
[in] Object ID.
Param offset
[in] The first octet of the object contents need to be calculated.
Param len
[in] The length number of octets object name.
Param data
[out] Pointer of actual object data.
Return
0 to accept, or any negative value to reject.

struct `bt_ots_init_param`

#include <*ots.h*> Descriptor for OTS initialization.

struct `bt_ots_client`

#include <*ots.h*> OTS client instance.

struct **bt_ots_client_cb**

#include <ots.h> OTS client callback structure.

Public Members

void (***obj_selected**)(struct *bt_ots_client* *ots_inst, struct bt_conn *conn, int err)

Callback function when a new object is selected.

Called when the a new object is selected and the current object has changed. The `cur_object` in `ots_inst` will have been reset, and metadata should be read again with *bt_ots_client_read_object_metadata()*.

Param ots_inst
Pointer to the OTC instance.

Param conn
The connection to the peer device.

Param err
Error code (bt_ots_olcp_res_code).

int (***obj_data_read**)(struct *bt_ots_client* *ots_inst, struct bt_conn *conn, uint32_t offset, uint32_t len, uint8_t *data_p, bool is_complete)

Callback function for the data of the selected object.

Called when the data of the selected object are read using *bt_ots_client_read_object_data()*.

Param ots_inst
Pointer to the OTC instance.

Param conn
The connection to the peer device.

Param offset
Offset of the received data.

Param len
Length of the received data.

Param data_p
Pointer to the received data.

Param is_complete
Indicate if the whole object has been received.

Return
int BT_OTS_STOP or BT_OTS_CONTINUE. BT_OTS_STOP can be used to stop reading.

void (***obj_metadata_read**)(struct *bt_ots_client* *ots_inst, struct bt_conn *conn, int err, uint8_t metadata_read)

Callback function for metadata of the selected object.

Called when metadata of the selected object are read using *bt_ots_client_read_object_metadata()*. Not all of the metadata may have been initialized.

Param ots_inst
Pointer to the OTC instance.

Param conn
The connection to the peer device.

Param err
Error value. 0 on success, GATT error or ERRNO on fail.

Param metadata_read
Bitfield of the metadata that was successfully read.

void (*obj_data_written)(struct *bt_ots_client* *ots_inst, struct bt_conn *conn, size_t len)

Callback function for the data of the write object.

Called when the data of the selected object is written using *bt_ots_client_write_object_data()*.

Param ots_inst
Pointer to the OTC instance.

Param conn
The connection to the peer device.

Param len
Length of the written data.

void (*obj_checksum_calculated)(struct *bt_ots_client* *ots_inst, struct bt_conn *conn, int err, uint32_t checksum)

Callback function when checksum indication is received.

Called when the oacp_ind_handler received response of OP BT_GATT_OTS_OACP_PROC_CHECKSUM_CALC.

Param ots_inst
Pointer to the OTC instance.

Param conn
The connection to the peer device.

Param err
Error code (bt_gatt_ots_oacp_res_code).

Param checksum
Checksum if error code is BT_GATT_OTS_OACP_RES_SUCCESS, otherwise 0.

Generic Access Profile (GAP)

API Reference

group bt_gap

Generic Access Profile (GAP)

Since
1.0

Version
1.0.0

Defines

BT_ID_DEFAULT

Convenience macro for specifying the default identity.

This helps make the code more readable, especially when only one identity is supported.

BT_DATA_SERIALIZED_SIZE(data_len)

Bluetooth data serialized size.

Get the size of a serialized *bt_data* given its data length.

Size of 'AD Structure'->'Length' field, equal to 1. Size of 'AD Structure'->'Data'->'AD Type' field, equal to 1. Size of 'AD Structure'->'Data'->'AD Data' field, equal to data_len.

See Core Specification Version 5.4 Vol. 3 Part C, 11, Figure 11.1.

`BT_DATA`(_type, _data, _data_len)

Helper to declare elements of *bt_data* arrays.

This macro is mainly for creating an array of struct *bt_data* elements which is then passed to e.g. *bt_le_adv_start()*.

Parameters

- **_type** – Type of advertising data field
- **_data** – Pointer to the data field payload
- **_data_len** – Number of bytes behind the _data pointer

`BT_DATA_BYTES`(_type, _bytes...)

Helper to declare elements of *bt_data* arrays.

This macro is mainly for creating an array of struct *bt_data* elements which is then passed to e.g. *bt_le_adv_start()*.

Parameters

- **_type** – Type of advertising data field
- **_bytes** – Variable number of single-byte parameters

`BT_LE_ADV_PARAM_INIT`(_options, _int_min, _int_max, _peer)

Initialize advertising parameters.

Parameters

- **_options** – Advertising Options
- **_int_min** – Minimum advertising interval
- **_int_max** – Maximum advertising interval
- **_peer** – Peer address, set to NULL for undirected advertising or address of peer for directed advertising.

`BT_LE_ADV_PARAM`(_options, _int_min, _int_max, _peer)

Helper to declare advertising parameters inline.

Parameters

- **_options** – Advertising Options
- **_int_min** – Minimum advertising interval
- **_int_max** – Maximum advertising interval
- **_peer** – Peer address, set to NULL for undirected advertising or address of peer for directed advertising.

`BT_LE_ADV_CONN_DIR`(_peer)

`BT_LE_ADV_CONN`

`BT_LE_ADV_CONN_ONE_TIME`

This is the recommended default for connectable advertisers.

`BT_LE_ADV_CONN_NAME`

Deprecated:
This macro will be removed in the near future, see https://github.com/zephyrproject-rtos/zephyr/issues/71686

`BT_LE_ADV_CONN_NAME_AD`

Deprecated:
This macro will be removed in the near future, see https://github.com/zephyrproject-rtos/zephyr/issues/71686

`BT_LE_ADV_CONN_DIR_LOW_DUTY(_peer)`

`BT_LE_ADV_NCONN`
Non-connectable advertising with private address.

`BT_LE_ADV_NCONN_NAME`

Deprecated:
This macro will be removed in the near future, see https://github.com/zephyrproject-rtos/zephyr/issues/71686

Non-connectable advertising with *BT_LE_ADV_OPT_USE_NAME*

`BT_LE_ADV_NCONN_IDENTITY`
Non-connectable advertising with *BT_LE_ADV_OPT_USE_IDENTITY*.

`BT_LE_EXT_ADV_CONN`
Connectable extended advertising.

`BT_LE_EXT_ADV_CONN_NAME`

Deprecated:
This macro will be removed in the near future, see https://github.com/zephyrproject-rtos/zephyr/issues/71686

Connectable extended advertising with *BT_LE_ADV_OPT_USE_NAME*

`BT_LE_EXT_ADV_SCAN`
Scannable extended advertising.

`BT_LE_EXT_ADV_SCAN_NAME`

Deprecated:
This macro will be removed in the near future, see https://github.com/zephyrproject-rtos/zephyr/issues/71686

Scannable extended advertising with *BT_LE_ADV_OPT_USE_NAME*

`BT_LE_EXT_ADV_NCONN`
Non-connectable extended advertising with private address.

`BT_LE_EXT_ADV_NCONN_NAME`

Deprecated:

This macro will be removed in the near future, see https://github.com/zephyrproject-rtos/zephyr/issues/71686

Non-connectable extended advertising with *BT_LE_ADV_OPT_USE_NAME*

`BT_LE_EXT_ADV_NCONN_IDENTITY`

Non-connectable extended advertising with *BT_LE_ADV_OPT_USE_IDENTITY*.

`BT_LE_EXT_ADV_CODED_NCONN`

Non-connectable extended advertising on coded PHY with private address.

`BT_LE_EXT_ADV_CODED_NCONN_NAME`

Deprecated:

This macro will be removed in the near future, see https://github.com/zephyrproject-rtos/zephyr/issues/71686

Non-connectable extended advertising on coded PHY with *BT_LE_ADV_OPT_USE_NAME*

`BT_LE_EXT_ADV_CODED_NCONN_IDENTITY`

Non-connectable extended advertising on coded PHY with *BT_LE_ADV_OPT_USE_IDENTITY*.

`BT_LE_EXT_ADV_START_PARAM_INIT`(_timeout, _n_evts)

Helper to initialize extended advertising start parameters inline.

Parameters

- **_timeout** – Advertiser timeout
- **_n_evts** – Number of advertising events

`BT_LE_EXT_ADV_START_PARAM`(_timeout, _n_evts)

Helper to declare extended advertising start parameters inline.

Parameters

- **_timeout** – Advertiser timeout
- **_n_evts** – Number of advertising events

`BT_LE_EXT_ADV_START_DEFAULT`

`BT_LE_PER_ADV_PARAM_INIT`(_int_min, _int_max, _options)

Helper to declare periodic advertising parameters inline.

Parameters

- **_int_min** – Minimum periodic advertising interval
- **_int_max** – Maximum periodic advertising interval
- **_options** – Periodic advertising properties bitfield.

`BT_LE_PER_ADV_PARAM`(_int_min, _int_max, _options)

Helper to declare periodic advertising parameters inline.

Parameters

- **_int_min** – Minimum periodic advertising interval
- **_int_max** – Maximum periodic advertising interval

- **_options** – Periodic advertising properties bitfield.

BT_LE_PER_ADV_DEFAULT

BT_LE_SCAN_OPT_FILTER_WHITELIST

BT_LE_SCAN_PARAM_INIT(_type, _options, _interval, _window)

Initialize scan parameters.

Parameters

- **_type** – Scan Type, BT_LE_SCAN_TYPE_ACTIVE or BT_LE_SCAN_TYPE_PASSIVE.
- **_options** – Scan options
- **_interval** – Scan Interval (N * 0.625 ms)
- **_window** – Scan Window (N * 0.625 ms)

BT_LE_SCAN_PARAM(_type, _options, _interval, _window)

Helper to declare scan parameters inline.

Parameters

- **_type** – Scan Type, BT_LE_SCAN_TYPE_ACTIVE or BT_LE_SCAN_TYPE_PASSIVE.
- **_options** – Scan options
- **_interval** – Scan Interval (N * 0.625 ms)
- **_window** – Scan Window (N * 0.625 ms)

BT_LE_SCAN_ACTIVE

Helper macro to enable active scanning to discover new devices.

BT_LE_SCAN_ACTIVE_CONTINUOUS

Helper macro to enable active scanning to discover new devices with window == interval.

Continuous scanning should be used to maximize the chances of receiving advertising packets.

BT_LE_SCAN_PASSIVE

Helper macro to enable passive scanning to discover new devices.

This macro should be used if information required for device identification (e.g., UUID) are known to be placed in Advertising Data.

BT_LE_SCAN_PASSIVE_CONTINUOUS

Helper macro to enable passive scanning to discover new devices with window==interval.

This macro should be used if information required for device identification (e.g., UUID) are known to be placed in Advertising Data.

BT_LE_SCAN_CODED_ACTIVE

Helper macro to enable active scanning to discover new devices.

Include scanning on Coded PHY in addition to 1M PHY.

BT_LE_SCAN_CODED_PASSIVE

Helper macro to enable passive scanning to discover new devices.

Include scanning on Coded PHY in addition to 1M PHY.

This macro should be used if information required for device identification (e.g., UUID) are known to be placed in Advertising Data.

Typedefs

typedef void (*bt_ready_cb_t)(int err)

Callback for notifying that Bluetooth has been enabled.

Param err
zero on success or (negative) error code otherwise.

typedef void bt_le_scan_cb_t(const *bt_addr_le_t* *addr, int8_t rssi, uint8_t adv_type, struct *net_buf_simple* *buf)

Callback type for reporting LE scan results.

A function of this type is given to the *bt_le_scan_start()* function and will be called for any discovered LE device.

Param addr
Advertiser LE address and type.

Param rssi
Strength of advertiser signal.

Param adv_type
Type of advertising response from advertiser. Uses the BT_GAP_ADV_TYPE_* values.

Param buf
Buffer containing advertiser data.

typedef void bt_br_discovery_cb_t(struct *bt_br_discovery_result* *results, size_t count)

Callback type for reporting BR/EDR discovery (inquiry) results.

A callback of this type is given to the *bt_br_discovery_start()* function and will be called at the end of the discovery with information about found devices populated in the results array.

Param results
Storage used for discovery results

Param count
Number of valid discovery results.

Enums

Advertising options.

Values:

enumerator BT_LE_ADV_OPT_NONE = 0

Convenience value when no options are specified.

enumerator **BT_LE_ADV_OPT_CONNECTABLE** = *BIT*(0)

Advertise as connectable.

Advertise as connectable. If not connectable then the type of advertising is determined by providing scan response data. The advertiser address is determined by the type of advertising and/or enabling privacy CONFIG_BT_PRIVACY .

enumerator **BT_LE_ADV_OPT_ONE_TIME** = *BIT*(1)

Advertise one time.

Don't try to resume connectable advertising after a connection. This option is only meaningful when used together with BT_LE_ADV_OPT_CONNECTABLE. If set the advertising will be stopped when *bt_le_adv_stop()* is called or when an incoming (peripheral) connection happens. If this option is not set the stack will take care of keeping advertising enabled even as connections occur. If Advertising directed or the advertiser was started with *bt_le_ext_adv_start* then this behavior is the default behavior and this flag has no effect.

enumerator **BT_LE_ADV_OPT_USE_IDENTITY** = *BIT*(2)

Advertise using identity address.

Advertise using the identity address as the advertiser address.

Note

The address used for advertising will not be the same as returned by *bt_le_oob_get_local*, instead *bt_id_get* should be used to get the LE address.

Warning

This will compromise the privacy of the device, so care must be taken when using this option.

enumerator **BT_LE_ADV_OPT_USE_NAME** = *BIT*(3)

Advertise using GAP device name.

Deprecated:

This option will be removed in the near future, see https://github.com/zephyrproject-rtos/zephyr/issues/71686

Include the GAP device name automatically when advertising. By default the GAP device name is put at the end of the scan response data. When advertising using *BT_LE_ADV_OPT_EXT_ADV* and not *BT_LE_ADV_OPT_SCANNABLE* then it will be put at the end of the advertising data. If the GAP device name does not fit into advertising data it will be converted to a shortened name if possible. *BT_LE_ADV_OPT_FORCE_NAME_IN_AD* can be used to force the device name to appear in the advertising data of an advert with scan response data.

The application can set the device name itself by including the following in the advertising data.

```
BT_DATA(BT_DATA_NAME_COMPLETE, name, sizeof(name) - 1)
```

enumerator BT_LE_ADV_OPT_DIR_MODE_LOW_DUTY = *BIT*(4)

Low duty cycle directed advertising.

Use low duty directed advertising mode, otherwise high duty mode will be used.

enumerator BT_LE_ADV_OPT_DIR_ADDR_RPA = *BIT*(5)

Directed advertising to privacy-enabled peer.

Enable use of Resolvable Private Address (RPA) as the target address in directed advertisements. This is required if the remote device is privacy-enabled and supports address resolution of the target address in directed advertisement. It is the responsibility of the application to check that the remote device supports address resolution of directed advertisements by reading its Central Address Resolution characteristic.

enumerator BT_LE_ADV_OPT_FILTER_SCAN_REQ = *BIT*(6)

Use filter accept list to filter devices that can request scan response data.

enumerator BT_LE_ADV_OPT_FILTER_CONN = *BIT*(7)

Use filter accept list to filter devices that can connect.

enumerator BT_LE_ADV_OPT_NOTIFY_SCAN_REQ = *BIT*(8)

Notify the application when a scan response data has been sent to an active scanner.

enumerator BT_LE_ADV_OPT_SCANNABLE = *BIT*(9)

Support scan response data.

When used together with *BT_LE_ADV_OPT_EXT_ADV* then this option cannot be used together with the *BT_LE_ADV_OPT_CONNECTABLE* option. When used together with *BT_LE_ADV_OPT_EXT_ADV* then scan response data must be set.

enumerator BT_LE_ADV_OPT_EXT_ADV = *BIT*(10)

Advertise with extended advertising.

This options enables extended advertising in the advertising set. In extended advertising the advertising set will send a small header packet on the three primary advertising channels. This small header points to the advertising data packet that will be sent on one of the 37 secondary advertising channels. The advertiser will send primary advertising on LE 1M PHY, and secondary advertising on LE 2M PHY. Connections will be established on LE 2M PHY.

Without this option the advertiser will send advertising data on the three primary advertising channels.

Note

Enabling this option requires extended advertising support in the peer devices scanning for advertisement packets.

Note

This cannot be used with *bt_le_adv_start()*.

enumerator BT_LE_ADV_OPT_NO_2M = *BIT*(11)

Disable use of LE 2M PHY on the secondary advertising channel.

Disabling the use of LE 2M PHY could be necessary if scanners don't support the LE 2M PHY. The advertiser will send primary advertising on LE 1M PHY, and secondary advertising on LE 1M PHY. Connections will be established on LE 1M PHY.

Note

Cannot be set if BT_LE_ADV_OPT_CODED is set.

Note

Requires *BT_LE_ADV_OPT_EXT_ADV*.

enumerator BT_LE_ADV_OPT_CODED = *BIT*(12)

Advertise on the LE Coded PHY (Long Range).

The advertiser will send both primary and secondary advertising on the LE Coded PHY. This gives the advertiser increased range with the trade-off of lower data rate and higher power consumption. Connections will be established on LE Coded PHY.

Note

Requires *BT_LE_ADV_OPT_EXT_ADV*

enumerator BT_LE_ADV_OPT_ANONYMOUS = *BIT*(13)

Advertise without a device address (identity or RPA).

Note

Requires *BT_LE_ADV_OPT_EXT_ADV*

enumerator BT_LE_ADV_OPT_USE_TX_POWER = *BIT*(14)

Advertise with transmit power.

Note

Requires *BT_LE_ADV_OPT_EXT_ADV*

enumerator BT_LE_ADV_OPT_DISABLE_CHAN_37 = *BIT*(15)

Disable advertising on channel index 37.

enumerator BT_LE_ADV_OPT_DISABLE_CHAN_38 = *BIT*(16)

Disable advertising on channel index 38.

enumerator **BT_LE_ADV_OPT_DISABLE_CHAN_39** = *BIT*(17)

Disable advertising on channel index 39.

enumerator **BT_LE_ADV_OPT_FORCE_NAME_IN_AD** = *BIT*(18)

Put GAP device name into advert data.

Deprecated:

This option will be removed in the near future, see https://github.com/zephyrproject-rtos/zephyr/issues/71686

Will place the GAP device name into the advertising data rather than the scan response data.

Note

Requires *BT_LE_ADV_OPT_USE_NAME*

enumerator **BT_LE_ADV_OPT_USE_NRPA** = *BIT*(19)

Advertise using a Non-Resolvable Private Address.

A new NRPA is set when updating the advertising parameters.

This is an advanced feature; most users will want to enable `CONFIG_BT_EXT_ADV` instead.

Note

Not implemented when `CONFIG_BT_PRIVACY` .

Note

Mutually exclusive with BT_LE_ADV_OPT_USE_IDENTITY.

Periodic Advertising options.

Values:

enumerator **BT_LE_PER_ADV_OPT_NONE** = 0

Convenience value when no options are specified.

enumerator **BT_LE_PER_ADV_OPT_USE_TX_POWER** = *BIT*(1)

Advertise with transmit power.

Note

Requires *BT_LE_ADV_OPT_EXT_ADV*

enumerator BT_LE_PER_ADV_OPT_INCLUDE_ADI = *BIT*(2)
Advertise with included AdvDataInfo (ADI).

> **Note**
> Requires *BT_LE_ADV_OPT_EXT_ADV*

Periodic advertising sync options.

Values:

enumerator BT_LE_PER_ADV_SYNC_OPT_NONE = 0
Convenience value when no options are specified.

enumerator BT_LE_PER_ADV_SYNC_OPT_USE_PER_ADV_LIST = *BIT*(0)
Use the periodic advertising list to sync with advertiser.
When this option is set, the address and SID of the parameters are ignored.

enumerator BT_LE_PER_ADV_SYNC_OPT_REPORTING_INITIALLY_DISABLED = *BIT*(1)
Disables periodic advertising reports.
No advertisement reports will be handled until enabled.

enumerator BT_LE_PER_ADV_SYNC_OPT_FILTER_DUPLICATE = *BIT*(2)
Filter duplicate Periodic Advertising reports.

enumerator BT_LE_PER_ADV_SYNC_OPT_DONT_SYNC_AOA = *BIT*(3)
Sync with Angle of Arrival (AoA) constant tone extension.

enumerator BT_LE_PER_ADV_SYNC_OPT_DONT_SYNC_AOD_1US = *BIT*(4)
Sync with Angle of Departure (AoD) 1 us constant tone extension.

enumerator BT_LE_PER_ADV_SYNC_OPT_DONT_SYNC_AOD_2US = *BIT*(5)
Sync with Angle of Departure (AoD) 2 us constant tone extension.

enumerator BT_LE_PER_ADV_SYNC_OPT_SYNC_ONLY_CONST_TONE_EXT = *BIT*(6)
Do not sync to packets without a constant tone extension.

Periodic Advertising Sync Transfer options.

Values:

enumerator BT_LE_PER_ADV_SYNC_TRANSFER_OPT_NONE = 0
Convenience value when no options are specified.

enumerator BT_LE_PER_ADV_SYNC_TRANSFER_OPT_SYNC_NO_AOA = *BIT*(0)
No Angle of Arrival (AoA)
Do not sync with Angle of Arrival (AoA) constant tone extension

enumerator BT_LE_PER_ADV_SYNC_TRANSFER_OPT_SYNC_NO_AOD_1US = *BIT*(1)

No Angle of Departure (AoD) 1 us.

Do not sync with Angle of Departure (AoD) 1 us constant tone extension

enumerator BT_LE_PER_ADV_SYNC_TRANSFER_OPT_SYNC_NO_AOD_2US = *BIT*(2)

No Angle of Departure (AoD) 2.

Do not sync with Angle of Departure (AoD) 2 us constant tone extension

enumerator BT_LE_PER_ADV_SYNC_TRANSFER_OPT_SYNC_ONLY_CTE = *BIT*(3)

Only sync to packets with constant tone extension.

enumerator BT_LE_PER_ADV_SYNC_TRANSFER_OPT_REPORTING_INITIALLY_DISABLED = *BIT*(4)

Sync to received PAST packets but don't generate sync reports.

This option must not be set at the same time as *BT_LE_PER_ADV_SYNC_TRANSFER_OPT_FILTER_DUPLICATES*.

enumerator BT_LE_PER_ADV_SYNC_TRANSFER_OPT_FILTER_DUPLICATES = *BIT*(5)

Sync to received PAST packets and generate sync reports with duplicate filtering.

This option must not be set at the same time as *BT_LE_PER_ADV_SYNC_TRANSFER_OPT_REPORTING_INITIALLY_DISABLED*.

Values:

enumerator BT_LE_SCAN_OPT_NONE = 0

Convenience value when no options are specified.

enumerator BT_LE_SCAN_OPT_FILTER_DUPLICATE = *BIT*(0)

Filter duplicates.

enumerator BT_LE_SCAN_OPT_FILTER_ACCEPT_LIST = *BIT*(1)

Filter using filter accept list.

enumerator BT_LE_SCAN_OPT_CODED = *BIT*(2)

Enable scan on coded PHY (Long Range).

enumerator BT_LE_SCAN_OPT_NO_1M = *BIT*(3)

Disable scan on 1M phy.

Note

Requires *BT_LE_SCAN_OPT_CODED*.

Values:

enumerator **BT_LE_SCAN_TYPE_PASSIVE** = 0x00

Scan without requesting additional information from advertisers.

enumerator **BT_LE_SCAN_TYPE_ACTIVE** = 0x01

Scan and request additional information from advertisers.

Using this scan type will automatically send scan requests to all devices. Scan responses are received in the same manner and using the same callbacks as advertising reports.

Functions

int **bt_enable**(*bt_ready_cb_t* cb)

Enable Bluetooth.

Enable Bluetooth. Must be the called before any calls that require communication with the local Bluetooth hardware.

When CONFIG_BT_SETTINGS is enabled, the application must load the Bluetooth settings after this API call successfully completes before Bluetooth APIs can be used. Loading the settings before calling this function is insufficient. Bluetooth settings can be loaded with *settings_load()* or *settings_load_subtree()* with argument "bt". The latter selectively loads only Bluetooth settings and is recommended if *settings_load()* has been called earlier.

Parameters

- **cb** – Callback to notify completion or NULL to perform the enabling synchronously.

Returns

Zero on success or (negative) error code otherwise.

int **bt_disable**(void)

Disable Bluetooth.

Disable Bluetooth. Can't be called before bt_enable has completed.

This API will clear all configured identities and keys that are not persistently stored with CONFIG_BT_SETTINGS . These can be restored with *settings_load()* before reenabling the stack.

This API does *not* clear previously registered callbacks like *bt_le_scan_cb_register* and *bt_conn_cb_register*. That is, the application shall not re-register them when the Bluetooth subsystem is re-enabled later.

Close and release HCI resources. Result is architecture dependent.

Returns

Zero on success or (negative) error code otherwise.

bool **bt_is_ready**(void)

Check if Bluetooth is ready.

Returns

true when Bluetooth is ready, false otherwise

int **bt_set_name**(const char *name)

Set Bluetooth Device Name.

Set Bluetooth GAP Device Name.

When advertising with device name in the advertising data the name should be updated by calling *bt_le_adv_update_data* or *bt_le_ext_adv_set_data*.

See also

CONFIG_BT_DEVICE_NAME_MAX .

Note

Requires CONFIG_BT_DEVICE_NAME_DYNAMIC .

Parameters

- **name** – New name

Returns

Zero on success or (negative) error code otherwise.

const char ***bt_get_name**(void)

Get Bluetooth Device Name.

Get Bluetooth GAP Device Name.

Returns

Bluetooth Device Name

uint16_t **bt_get_appearance**(void)

Get local Bluetooth appearance.

Bluetooth Appearance is a description of the external appearance of a device in terms of an Appearance Value.

See also

https://specificationrefs.bluetooth.com/assigned-values/Appearance%20Values.pdf

Returns

Appearance Value of local Bluetooth host.

int **bt_set_appearance**(uint16_t new_appearance)

Set local Bluetooth appearance.

Automatically preserves the new appearance across reboots if CONFIG_BT_SETTINGS is enabled.

This symbol is linkable if CONFIG_BT_DEVICE_APPEARANCE_DYNAMIC is enabled.

Parameters

- **new_appearance** – Appearance Value

Return values

- **0** – Success.
- **other** – Persistent storage failed. Appearance was not updated.

void **bt_id_get**(*bt_addr_le_t* *addrs, size_t *count)

Get the currently configured identities.

Returns an array of the currently configured identity addresses. To make sure all available identities can be retrieved, the number of elements in the *addrs* array should be CONFIG_BT_ID_MAX. The identity identifier that some APIs expect (such as advertising parameters) is simply the index of the identity in the *addrs* array.

If *addrs* is passed as NULL, then returned *count* contains the count of all available identities that can be retrieved with a subsequent call to this function with non-NULL *addrs* parameter.

Note

Deleted identities may show up as *BT_ADDR_LE_ANY* in the returned array.

Parameters

- **addrs** – Array where to store the configured identities.
- **count** – Should be initialized to the array size. Once the function returns it will contain the number of returned identities.

int **bt_id_create**(*bt_addr_le_t* *addr, uint8_t *irk)

Create a new identity.

Create a new identity using the given address and IRK. This function can be called before calling *bt_enable()*. However, the new identity will only be stored persistently in flash when this API is used after *bt_enable()*. The reason is that the persistent settings are loaded after *bt_enable()* and would therefore cause potential conflicts with the stack blindly overwriting what's stored in flash. The identity will also not be written to flash in case a pre-defined address is provided, since in such a situation the app clearly has some place it got the address from and will be able to repeat the procedure on every power cycle, i.e. it would be redundant to also store the information in flash.

Generating random static address or random IRK is not supported when calling this function before *bt_enable()*.

If the application wants to have the stack randomly generate identities and store them in flash for later recovery, the way to do it would be to first initialize the stack (using bt_enable), then call *settings_load()*, and after that check with *bt_id_get()* how many identities were recovered. If an insufficient amount of identities were recovered the app may then call *bt_id_create()* to create new ones.

If supported by the HCI driver (indicated by setting CONFIG_BT_HCI_SET_PUBLIC_ADDR), the first call to this function can be used to set the controller's public identity address. This call must happen before calling *bt_enable()*. Subsequent calls always add/generate random static addresses.

Parameters

- **addr** – Address to use for the new identity. If NULL or initialized to BT_ADDR_LE_ANY the stack will generate a new random static address for the identity and copy it to the given parameter upon return from this function (in case the parameter was non-NULL).
- **irk** – Identity Resolving Key (16 bytes) to be used with this identity. If set to all zeroes or NULL, the stack will generate a random IRK for the identity and copy it back to the parameter upon return from this function (in case the parameter was non-NULL). If privacy CONFIG_BT_PRIVACY is not enabled this parameter must be NULL.

Returns
Identity identifier (>= 0) in case of success, or a negative error code on failure.

int **bt_id_reset**(uint8_t id, *bt_addr_le_t* *addr, uint8_t *irk)

Reset/reclaim an identity for reuse.

The semantics of the *addr* and *irk* parameters of this function are the same as with *bt_id_create()*. The difference is the first *id* parameter that needs to be an existing identity (if it doesn't exist this function will return an error). When given an existing identity this function will disconnect any connections created using it, remove any pairing keys or other data associated with it, and then create a new identity in the same slot, based on the *addr* and *irk* parameters.

Note

the default identity (BT_ID_DEFAULT) cannot be reset, i.e. this API will return an error if asked to do that.

Parameters

- **id** – Existing identity identifier.
- **addr** – Address to use for the new identity. If NULL or initialized to BT_ADDR_LE_ANY the stack will generate a new static random address for the identity and copy it to the given parameter upon return from this function (in case the parameter was non-NULL).
- **irk** – Identity Resolving Key (16 bytes) to be used with this identity. If set to all zeroes or NULL, the stack will generate a random IRK for the identity and copy it back to the parameter upon return from this function (in case the parameter was non-NULL). If privacy CONFIG_BT_PRIVACY is not enabled this parameter must be NULL.

Returns
Identity identifier (>= 0) in case of success, or a negative error code on failure.

int **bt_id_delete**(uint8_t id)

Delete an identity.

When given a valid identity this function will disconnect any connections created using it, remove any pairing keys or other data associated with it, and then flag is as deleted, so that it can not be used for any operations. To take back into use the slot the identity was occupying the *bt_id_reset()* API needs to be used.

Note

the default identity (BT_ID_DEFAULT) cannot be deleted, i.e. this API will return an error if asked to do that.

Parameters

- **id** – Existing identity identifier.

Returns
0 in case of success, or a negative error code on failure.

size_t **bt_data_get_len**(const struct *bt_data* data[], size_t data_count)

Get the total size (in bytes) of a given set of *bt_data* structures.

Parameters

- **data** – **[in]** Array of *bt_data* structures.
- **data_count** – **[in]** Number of *bt_data* structures in `data`.

Returns

Size of the concatenated data, built from the *bt_data* structure set.

size_t **bt_data_serialize**(const struct *bt_data* *input, uint8_t *output)

Serialize a *bt_data* struct into an advertising structure (a flat byte array).

The data are formatted according to the Bluetooth Core Specification v. 5.4, vol. 3, part C, 11.

Parameters

- **input** – **[in]** Single *bt_data* structure to read from.
- **output** – **[out]** Buffer large enough to store the advertising structure in `input`. The size of it must be at least the size of the `input->data_len + 2` (for the type and the length).

Returns

Number of bytes written in `output`.

int **bt_le_adv_start**(const struct *bt_le_adv_param* *param, const struct *bt_data* *ad, size_t ad_len, const struct *bt_data* *sd, size_t sd_len)

Start advertising.

Set advertisement data, scan response data, advertisement parameters and start advertising.

When the advertisement parameter peer address has been set the advertising will be directed to the peer. In this case advertisement data and scan response data parameters are ignored. If the mode is high duty cycle the timeout will be *BT_GAP_ADV_HIGH_DUTY_CYCLE_MAX_TIMEOUT*.

This function cannot be used with *BT_LE_ADV_OPT_EXT_ADV* in the `param.options`. For extended advertising, the bt_le_ext_adv_* functions must be used.

Parameters

- **param** – Advertising parameters.
- **ad** – Data to be used in advertisement packets.
- **ad_len** – Number of elements in ad
- **sd** – Data to be used in scan response packets.
- **sd_len** – Number of elements in sd

Returns

Zero on success or (negative) error code otherwise.

Returns

-ENOMEM No free connection objects available for connectable advertiser.

Returns

-ECONNREFUSED When connectable advertising is requested and there is already maximum number of connections established in the controller. This error code is only guaranteed when using Zephyr controller, for other controllers code returned in this case may be -EIO.

int **bt_le_adv_update_data**(const struct *bt_data* *ad, size_t ad_len, const struct *bt_data* *sd, size_t sd_len)

Update advertising.

Update advertisement and scan response data.

Parameters

- **ad** – Data to be used in advertisement packets.
- **ad_len** – Number of elements in ad
- **sd** – Data to be used in scan response packets.
- **sd_len** – Number of elements in sd

Returns
Zero on success or (negative) error code otherwise.

int **bt_le_adv_stop**(void)

Stop advertising.

Stops ongoing advertising.

Returns
Zero on success or (negative) error code otherwise.

int **bt_le_ext_adv_create**(const struct *bt_le_adv_param* *param, const struct *bt_le_ext_adv_cb* *cb, struct bt_le_ext_adv **adv)

Create advertising set.

Create a new advertising set and set advertising parameters. Advertising parameters can be updated with *bt_le_ext_adv_update_param*.

Parameters

- **param** – **[in]** Advertising parameters.
- **cb** – **[in]** Callback struct to notify about advertiser activity. Can be NULL. Must point to valid memory during the lifetime of the advertising set.
- **adv** – **[out]** Valid advertising set object on success.

Returns
Zero on success or (negative) error code otherwise.

int **bt_le_ext_adv_start**(struct bt_le_ext_adv *adv, const struct *bt_le_ext_adv_start_param* *param)

Start advertising with the given advertising set.

If the advertiser is limited by either the timeout or number of advertising events the application will be notified by the advertiser sent callback once the limit is reached. If the advertiser is limited by both the timeout and the number of advertising events then the limit that is reached first will stop the advertiser.

Parameters

- **adv** – Advertising set object.
- **param** – Advertise start parameters.

int **bt_le_ext_adv_stop**(struct bt_le_ext_adv *adv)

Stop advertising with the given advertising set.

Stop advertising with a specific advertising set. When using this function the advertising sent callback will not be called.

Parameters

- **adv** – Advertising set object.

Returns
Zero on success or (negative) error code otherwise.

int **bt_le_ext_adv_set_data**(struct bt_le_ext_adv *adv, const struct *bt_data* *ad, size_t ad_len, const struct *bt_data* *sd, size_t sd_len)

Set an advertising set's advertising or scan response data.

Set advertisement data or scan response data. If the advertising set is currently advertising then the advertising data will be updated in subsequent advertising events.

When both *BT_LE_ADV_OPT_EXT_ADV* and *BT_LE_ADV_OPT_SCANNABLE* are enabled then advertising data is ignored. When *BT_LE_ADV_OPT_SCANNABLE* is not enabled then scan response data is ignored.

If the advertising set has been configured to send advertising data on the primary advertising channels then the maximum data length is *BT_GAP_ADV_MAX_ADV_DATA_LEN* bytes. If the advertising set has been configured for extended advertising, then the maximum data length is defined by the controller with the maximum possible of *BT_GAP_ADV_MAX_EXT_ADV_DATA_LEN* bytes.

Note

Not all scanners support extended data length advertising data.

Note

When updating the advertising data while advertising the advertising data and scan response data length must be smaller or equal to what can be fit in a single advertising packet. Otherwise the advertiser must be stopped.

Parameters

- **adv** – Advertising set object.
- **ad** – Data to be used in advertisement packets.
- **ad_len** – Number of elements in ad
- **sd** – Data to be used in scan response packets.
- **sd_len** – Number of elements in sd

Returns
Zero on success or (negative) error code otherwise.

int **bt_le_ext_adv_update_param**(struct bt_le_ext_adv *adv, const struct *bt_le_adv_param* *param)

Update advertising parameters.

Update the advertising parameters. The function will return an error if the advertiser set is currently advertising. Stop the advertising set before calling this function.

Note

When changing the option *BT_LE_ADV_OPT_USE_NAME* then *bt_le_ext_adv_set_data* needs to be called in order to update the advertising data and scan response data.

Parameters

- `adv` – Advertising set object.
- `param` – Advertising parameters.

Returns
Zero on success or (negative) error code otherwise.

int `bt_le_ext_adv_delete`(struct bt_le_ext_adv *adv)

Delete advertising set.

Delete advertising set. This will free up the advertising set and make it possible to create a new advertising set.

Returns
Zero on success or (negative) error code otherwise.

uint8_t `bt_le_ext_adv_get_index`(struct bt_le_ext_adv *adv)

Get array index of an advertising set.

This function is used to map bt_adv to index of an array of advertising sets. The array has CONFIG_BT_EXT_ADV_MAX_ADV_SET elements.

Parameters

- `adv` – Advertising set.

Returns
Index of the advertising set object. The range of the returned value is 0..CONFIG_BT_EXT_ADV_MAX_ADV_SET-1

int `bt_le_ext_adv_get_info`(const struct bt_le_ext_adv *adv, struct *bt_le_ext_adv_info* *info)

Get advertising set info.

Parameters

- `adv` – Advertising set object
- `info` – Advertising set info object

Returns
Zero on success or (negative) error code on failure.

int `bt_le_per_adv_set_param`(struct bt_le_ext_adv *adv, const struct *bt_le_per_adv_param* *param)

Set or update the periodic advertising parameters.

The periodic advertising parameters can only be set or updated on an extended advertisement set which is neither scannable, connectable nor anonymous.

Parameters

- `adv` – Advertising set object.
- `param` – Advertising parameters.

Returns
Zero on success or (negative) error code otherwise.

int `bt_le_per_adv_set_data`(const struct bt_le_ext_adv *adv, const struct *bt_data* *ad, size_t ad_len)

Set or update the periodic advertising data.

The periodic advertisement data can only be set or updated on an extended advertisement set which is neither scannable, connectable nor anonymous.

Parameters

- **adv** – Advertising set object.
- **ad** – Advertising data.
- **ad_len** – Advertising data length.

Returns
Zero on success or (negative) error code otherwise.

int **bt_le_per_adv_set_subevent_data**(const struct bt_le_ext_adv *adv, uint8_t num_subevents, const struct *bt_le_per_adv_subevent_data_params* *params)

Set the periodic advertising with response subevent data.

Set the data for one or more subevents of a Periodic Advertising with Responses Advertiser in reply data request.

Parameters

- **adv** – The extended advertiser the PAwR train belongs to.
- **num_subevents** – The number of subevents to set data for.
- **params** – Subevent parameters.

Pre
There are `num_subevents` elements in `params`.

Pre
The controller has requested data for the subevents in `params`.

Returns
Zero on success or (negative) error code otherwise.

int **bt_le_per_adv_start**(struct bt_le_ext_adv *adv)

Starts periodic advertising.

Enabling the periodic advertising can be done independently of extended advertising, but both periodic advertising and extended advertising shall be enabled before any periodic advertising data is sent. The periodic advertising and extended advertising can be enabled in any order.

Once periodic advertising has been enabled, it will continue advertising until *bt_le_per_adv_stop()* has been called, or if the advertising set is deleted by *bt_le_ext_adv_delete()*. Calling *bt_le_ext_adv_stop()* will not stop the periodic advertising.

Parameters

- **adv** – Advertising set object.

Returns
Zero on success or (negative) error code otherwise.

int **bt_le_per_adv_stop**(struct bt_le_ext_adv *adv)

Stops periodic advertising.

Disabling the periodic advertising can be done independently of extended advertising. Disabling periodic advertising will not disable extended advertising.

Parameters

- **adv** – Advertising set object.

Returns
Zero on success or (negative) error code otherwise.

uint8_t **bt_le_per_adv_sync_get_index**(struct bt_le_per_adv_sync *per_adv_sync)

Get array index of an periodic advertising sync object.

This function is get the index of an array of periodic advertising sync objects. The array has CONFIG_BT_PER_ADV_SYNC_MAX elements.

Parameters

- **per_adv_sync** – The periodic advertising sync object.

Returns

Index of the periodic advertising sync object. The range of the returned value is 0..CONFIG_BT_PER_ADV_SYNC_MAX-1

struct bt_le_per_adv_sync ***bt_le_per_adv_sync_lookup_index**(uint8_t index)

Get a periodic advertising sync object from the array index.

This function is to get the periodic advertising sync object from the array index. The array has CONFIG_BT_PER_ADV_SYNC_MAX elements.

Parameters

- **index** – The index of the periodic advertising sync object. The range of the index value is 0..CONFIG_BT_PER_ADV_SYNC_MAX-1

Returns

The periodic advertising sync object of the array index or NULL if invalid index.

int **bt_le_per_adv_sync_get_info**(struct bt_le_per_adv_sync *per_adv_sync, struct *bt_le_per_adv_sync_info* *info)

Get periodic adv sync information.

Parameters

- **per_adv_sync** – Periodic advertising sync object.
- **info** – Periodic advertising sync info object

Returns

Zero on success or (negative) error code on failure.

struct bt_le_per_adv_sync ***bt_le_per_adv_sync_lookup_addr**(const *bt_addr_le_t* *adv_addr, uint8_t sid)

Look up an existing periodic advertising sync object by advertiser address.

Parameters

- **adv_addr** – Advertiser address.
- **sid** – The advertising set ID.

Returns

Periodic advertising sync object or NULL if not found.

int **bt_le_per_adv_sync_create**(const struct *bt_le_per_adv_sync_param* *param, struct bt_le_per_adv_sync **out_sync)

Create a periodic advertising sync object.

Create a periodic advertising sync object that can try to synchronize to periodic advertising reports from an advertiser. Scan shall either be disabled or extended scan shall be enabled.

This function does not timeout, and will continue to look for an advertiser until it either finds it or *bt_le_per_adv_sync_delete()* is called. It is thus suggested to implement a timeout when using this, if it is expected to find the advertiser within a reasonable timeframe.

Parameters

- `param` – **[in]** Periodic advertising sync parameters.
- `out_sync` – **[out]** Periodic advertising sync object on.

Returns
Zero on success or (negative) error code otherwise.

int `bt_le_per_adv_sync_delete`(struct bt_le_per_adv_sync *per_adv_sync)

Delete periodic advertising sync.

Delete the periodic advertising sync object. Can be called regardless of the state of the sync. If the syncing is currently syncing, the syncing is cancelled. If the sync has been established, it is terminated. The periodic advertising sync object will be invalidated afterwards.

If the state of the sync object is syncing, then a new periodic advertising sync object may not be created until the controller has finished canceling this object.

Parameters

- `per_adv_sync` – The periodic advertising sync object.

Returns
Zero on success or (negative) error code otherwise.

int `bt_le_per_adv_sync_cb_register`(struct *bt_le_per_adv_sync_cb* *cb)

Register periodic advertising sync callbacks.

Adds the callback structure to the list of callback structures for periodic advertising syncs.

This callback will be called for all periodic advertising sync activity, such as synced, terminated and when data is received.

Parameters

- `cb` – Callback struct. Must point to memory that remains valid.

Return values

- `0` – Success.
- `-EEXIST` – if cb was already registered.

int `bt_le_per_adv_sync_recv_enable`(struct bt_le_per_adv_sync *per_adv_sync)

Enables receiving periodic advertising reports for a sync.

If the sync is already receiving the reports, -EALREADY is returned.

Parameters

- `per_adv_sync` – The periodic advertising sync object.

Returns
Zero on success or (negative) error code otherwise.

int `bt_le_per_adv_sync_recv_disable`(struct bt_le_per_adv_sync *per_adv_sync)

Disables receiving periodic advertising reports for a sync.

If the sync report receiving is already disabled, -EALREADY is returned.

Parameters

- `per_adv_sync` – The periodic advertising sync object.

Returns
Zero on success or (negative) error code otherwise.

int **bt_le_per_adv_sync_transfer**(const struct bt_le_per_adv_sync *per_adv_sync, const struct bt_conn *conn, uint16_t service_data)

Transfer the periodic advertising sync information to a peer device.

This will allow another device to quickly synchronize to the same periodic advertising train that this device is currently synced to.

Parameters

- **per_adv_sync** – The periodic advertising sync to transfer.
- **conn** – The peer device that will receive the sync information.
- **service_data** – Application service data provided to the remote host.

Returns
Zero on success or (negative) error code otherwise.

int **bt_le_per_adv_set_info_transfer**(const struct bt_le_ext_adv *adv, const struct bt_conn *conn, uint16_t service_data)

Transfer the information about a periodic advertising set.

This will allow another device to quickly synchronize to periodic advertising set from this device.

Parameters

- **adv** – The periodic advertising set to transfer info of.
- **conn** – The peer device that will receive the information.
- **service_data** – Application service data provided to the remote host.

Returns
Zero on success or (negative) error code otherwise.

int **bt_le_per_adv_sync_transfer_subscribe**(const struct bt_conn *conn, const struct *bt_le_per_adv_sync_transfer_param* *param)

Subscribe to periodic advertising sync transfers (PASTs).

Sets the parameters and allow other devices to transfer periodic advertising syncs.

Parameters

- **conn** – The connection to set the parameters for. If NULL default parameters for all connections will be set. Parameters set for specific connection will always have precedence.
- **param** – The periodic advertising sync transfer parameters.

Returns
Zero on success or (negative) error code otherwise.

int **bt_le_per_adv_sync_transfer_unsubscribe**(const struct bt_conn *conn)

Unsubscribe from periodic advertising sync transfers (PASTs).

Remove the parameters that allow other devices to transfer periodic advertising syncs.

Parameters

- **conn** – The connection to remove the parameters for. If NULL default parameters for all connections will be removed. Unsubscribing for a specific device, will still allow other devices to transfer periodic advertising syncs.

Returns
Zero on success or (negative) error code otherwise.

int **bt_le_per_adv_list_add**(const *bt_addr_le_t* *addr, uint8_t sid)

Add a device to the periodic advertising list.

Add peer device LE address to the periodic advertising list. This will make it possibly to automatically create a periodic advertising sync to this device.

Parameters

- **addr** – Bluetooth LE identity address.
- **sid** – The advertising set ID. This value is obtained from the *bt_le_scan_recv_info* in the scan callback.

Returns

Zero on success or (negative) error code otherwise.

int **bt_le_per_adv_list_remove**(const *bt_addr_le_t* *addr, uint8_t sid)

Remove a device from the periodic advertising list.

Removes peer device LE address from the periodic advertising list.

Parameters

- **addr** – Bluetooth LE identity address.
- **sid** – The advertising set ID. This value is obtained from the *bt_le_scan_recv_info* in the scan callback.

Returns

Zero on success or (negative) error code otherwise.

int **bt_le_per_adv_list_clear**(void)

Clear the periodic advertising list.

Clears the entire periodic advertising list.

Returns

Zero on success or (negative) error code otherwise.

int **bt_le_scan_start**(const struct *bt_le_scan_param* *param, *bt_le_scan_cb_t* cb)

Start (LE) scanning.

Start LE scanning with given parameters and provide results through the specified callback.

Note

The LE scanner by default does not use the Identity Address of the local device when CONFIG_BT_PRIVACY is disabled. This is to prevent the active scanner from disclosing the identity information when requesting additional information from advertisers. In order to enable directed advertiser reports then CONFIG_BT_SCAN_WITH_IDENTITY must be enabled.

Note

Setting the param.timeout parameter is not supported when CONFIG_BT_PRIVACY is enabled, when the param.type is *BT_LE_SCAN_TYPE_ACTIVE*. Supplying a non-zero timeout will result in an -EINVAL error code.

Parameters

- **param** – Scan parameters.

- **cb** – Callback to notify scan results. May be NULL if callback registration through *bt_le_scan_cb_register* is preferred.

Returns
Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

int **bt_le_scan_stop**(void)

Stop (LE) scanning.

Stops ongoing LE scanning.

Returns
Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

int **bt_le_scan_cb_register**(struct *bt_le_scan_cb* *cb)

Register scanner packet callbacks.

Adds the callback structure to the list of callback structures that monitors scanner activity.

This callback will be called for all scanner activity, regardless of what API was used to start the scanner.

Parameters

- **cb** – Callback struct. Must point to memory that remains valid.

Return values

- **0** – Success.
- **-EEXIST** – if cb was already registered.

void **bt_le_scan_cb_unregister**(struct *bt_le_scan_cb* *cb)

Unregister scanner packet callbacks.

Remove the callback structure from the list of scanner callbacks.

Parameters

- **cb** – Callback struct. Must point to memory that remains valid.

int **bt_le_filter_accept_list_add**(const *bt_addr_le_t* *addr)

Add device (LE) to filter accept list.

Add peer device LE address to the filter accept list.

> **Note**
>
> The filter accept list cannot be modified when an LE role is using the filter accept list, i.e advertiser or scanner using a filter accept list or automatic connecting to devices using filter accept list.

Parameters

- **addr** – Bluetooth LE identity address.

Returns
Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

int **bt_le_filter_accept_list_remove**(const *bt_addr_le_t* *addr)

Remove device (LE) from filter accept list.

Remove peer device LE address from the filter accept list.

> **Note**
>
> The filter accept list cannot be modified when an LE role is using the filter accept list, i.e advertiser or scanner using a filter accept list or automatic connecting to devices using filter accept list.

Parameters

- **addr** – Bluetooth LE identity address.

Returns

Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

int **bt_le_filter_accept_list_clear**(void)

Clear filter accept list.

Clear all devices from the filter accept list.

> **Note**
>
> The filter accept list cannot be modified when an LE role is using the filter accept list, i.e advertiser or scanner using a filter accept list or automatic connecting to devices using filter accept list.

Returns

Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

int **bt_le_set_chan_map**(uint8_t chan_map[5])

Set (LE) channel map.

Parameters

- **chan_map** – Channel map.

Returns

Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

int **bt_le_set_rpa_timeout**(uint16_t new_rpa_timeout)

Set the Resolvable Private Address timeout in runtime.

The new RPA timeout value will be used for the next RPA rotation and all subsequent rotations until another override is scheduled with this API.

Initially, the if `CONFIG_BT_RPA_TIMEOUT` is used as the RPA timeout.

This symbol is linkable if `CONFIG_BT_RPA_TIMEOUT_DYNAMIC` is enabled.

Parameters

- **new_rpa_timeout** – Resolvable Private Address timeout in seconds

Return values

- **0** – Success.

- `-EINVAL` – RPA timeout value is invalid. Valid range is 1s - 3600s.

void **bt_data_parse**(struct *net_buf_simple* *ad, bool (*func)(struct *bt_data* *data, void *user_data), void *user_data)

Helper for parsing advertising (or EIR or OOB) data.

A helper for parsing the basic data types used for Extended Inquiry Response (EIR), Advertising Data (AD), and OOB data blocks. The most common scenario is to call this helper on the advertising data received in the callback that was given to *bt_le_scan_start()*.

Warning

This helper function will consume `ad` when parsing. The user should make a copy if the original data is to be used afterwards

Parameters

- `ad` – Advertising data as given to the bt_le_scan_cb_t callback.
- `func` – Callback function which will be called for each element that's found in the data. The callback should return true to continue parsing, or false to stop parsing.
- `user_data` – User data to be passed to the callback.

int **bt_le_oob_get_local**(uint8_t id, struct *bt_le_oob* *oob)

Get local LE Out of Band (OOB) information.

This function allows to get local information that are useful for Out of Band pairing or connection creation.

If privacy `CONFIG_BT_PRIVACY` is enabled this will result in generating new Resolvable Private Address (RPA) that is valid for `CONFIG_BT_RPA_TIMEOUT` seconds. This address will be used for advertising started by *bt_le_adv_start*, active scanning and connection creation.

Note

If privacy is enabled the RPA cannot be refreshed in the following cases:

- Creating a connection in progress, wait for the connected callback. In addition when extended advertising `CONFIG_BT_EXT_ADV` is not enabled or not supported by the controller:
- Advertiser is enabled using a Random Static Identity Address for a different local identity.
- The local identity conflicts with the local identity used by other roles.

Parameters

- `id` – **[in]** Local identity, in most cases BT_ID_DEFAULT.
- `oob` – **[out]** LE OOB information

Returns

Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

int **bt_le_ext_adv_oob_get_local**(struct bt_le_ext_adv *adv, struct *bt_le_oob* *oob)

Get local LE Out of Band (OOB) information.

This function allows to get local information that are useful for Out of Band pairing or connection creation.

If privacy `CONFIG_BT_PRIVACY` is enabled this will result in generating new Resolvable Private Address (RPA) that is valid for `CONFIG_BT_RPA_TIMEOUT` seconds. This address will be used by the advertising set.

Note

When generating OOB information for multiple advertising set all OOB information needs to be generated at the same time.

Note

If privacy is enabled the RPA cannot be refreshed in the following cases:

- Creating a connection in progress, wait for the connected callback.

Parameters

- `adv` – **[in]** The advertising set object
- `oob` – **[out]** LE OOB information

Returns

Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

int **bt_br_discovery_start**(const struct *bt_br_discovery_param* *param, struct *bt_br_discovery_result* *results, size_t count, *bt_br_discovery_cb_t* cb)

Start BR/EDR discovery.

Start BR/EDR discovery (inquiry) and provide results through the specified callback. When bt_br_discovery_cb_t is called it indicates that discovery has completed. If more inquiry results were received during session than fits in provided result storage, only ones with highest RSSI will be reported.

Parameters

- `param` – Discovery parameters.
- `results` – Storage for discovery results.
- `count` – Number of results in storage. Valid range: 1-255.
- `cb` – Callback to notify discovery results.

Returns

Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error

int **bt_br_discovery_stop**(void)

Stop BR/EDR discovery.

Stops ongoing BR/EDR discovery. If discovery was stopped by this call results won't be reported

Returns
Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

int **bt_br_oob_get_local**(struct *bt_br_oob* *oob)

Get BR/EDR local Out Of Band information.

This function allows to get local controller information that are useful for Out Of Band pairing or connection creation process.

Parameters

- **oob** – Out Of Band information

int **bt_br_set_discoverable**(bool enable)

Enable/disable set controller in discoverable state.

Allows make local controller to listen on INQUIRY SCAN channel and responds to devices making general inquiry. To enable this state it's mandatory to first be in connectable state.

Parameters

- **enable** – Value allowing/disallowing controller to become discoverable.

Returns
Negative if fail set to requested state or requested state has been already set. Zero if done successfully.

int **bt_br_set_connectable**(bool enable)

Enable/disable set controller in connectable state.

Allows make local controller to be connectable. It means the controller start listen to devices requests on PAGE SCAN channel. If disabled also resets discoverability if was set.

Parameters

- **enable** – Value allowing/disallowing controller to be connectable.

Returns
Negative if fail set to requested state or requested state has been already set. Zero if done successfully.

int **bt_unpair**(uint8_t id, const *bt_addr_le_t* *addr)

Clear pairing information.

Parameters

- **id** – Local identity (mostly just BT_ID_DEFAULT).
- **addr** – Remote address, NULL or BT_ADDR_LE_ANY to clear all remote devices.

Returns
0 on success or negative error value on failure.

void **bt_foreach_bond**(uint8_t id, void (*func)(const struct *bt_bond_info* *info, void *user_data), void *user_data)

Iterate through all existing bonds.

Parameters

- **id** – Local identity (mostly just BT_ID_DEFAULT).
- **func** – Function to call for each bond.
- **user_data** – Data to pass to the callback function.

int **bt_configure_data_path**(uint8_t dir, uint8_t id, uint8_t vs_config_len, const uint8_t *vs_config)

Configure vendor data path.

Request the Controller to configure the data transport path in a given direction between the Controller and the Host.

Parameters

- **dir** – Direction to be configured, BT_HCI_DATAPATH_DIR_HOST_TO_CTLR or BT_HCI_DATAPATH_DIR_CTLR_TO_HOST
- **id** – Vendor specific logical transport channel ID, range [BT_HCI_DATAPATH_ID_VS..BT_HCI_DATAPATH_ID_VS_END]
- **vs_config_len** – Length of additional vendor specific configuration data
- **vs_config** – Pointer to additional vendor specific configuration data

Returns

0 in case of success or negative value in case of error.

int **bt_le_per_adv_sync_subevent**(struct bt_le_per_adv_sync *per_adv_sync, struct *bt_le_per_adv_sync_subevent_params* *params)

Synchronize with a subset of subevents.

Until this command is issued, the subevent(s) the controller is synchronized to is unspecified.

Parameters

- **per_adv_sync** – The periodic advertising sync object.
- **params** – Parameters.

Returns

0 in case of success or negative value in case of error.

int **bt_le_per_adv_set_response_data**(struct bt_le_per_adv_sync *per_adv_sync, const struct *bt_le_per_adv_response_params* *params, const struct *net_buf_simple* *data)

Set the data for a response slot in a specific subevent of the PAwR.

This function is called by the application to set the response data. The data for a response slot shall be transmitted only once.

Parameters

- **per_adv_sync** – The periodic advertising sync object.
- **params** – Parameters.
- **data** – The response data to send.

Returns

Zero on success or (negative) error code otherwise.

struct **bt_le_ext_adv_sent_info**

#include <bluetooth.h>

Public Members

uint8_t **num_sent**

The number of advertising events completed.

struct **bt_le_ext_adv_connected_info**

#include <bluetooth.h>

Public Members

struct bt_conn ***conn**

Connection object of the new connection.

struct **bt_le_ext_adv_scanned_info**

#include <bluetooth.h>

Public Members

bt_addr_le_t ***addr**

Active scanner LE address and type.

struct **bt_le_per_adv_data_request**

#include <bluetooth.h>

Public Members

uint8_t **start**

The first subevent data can be set for.

uint8_t **count**

The number of subevents data can be set for.

struct **bt_le_per_adv_response_info**

#include <bluetooth.h>

Public Members

uint8_t **subevent**

The subevent the response was received in.

uint8_t **tx_status**

Status of the subevent indication.

0 if subevent indication was transmitted. 1 if subevent indication was not transmitted. All other values RFU.

int8_t **tx_power**

The TX power of the response in dBm.

int8_t **rssi**

The RSSI of the response in dBm.

uint8_t **cte_type**

The Constant Tone Extension (CTE) of the advertisement (bt_df_cte_type)

uint8_t **response_slot**

The slot the response was received in.

struct **bt_le_ext_adv_cb**

#include <bluetooth.h>

Public Members

void (***sent**)(struct bt_le_ext_adv *adv, struct *bt_le_ext_adv_sent_info* *info)

The advertising set has finished sending adv data.

This callback notifies the application that the advertising set has finished sending advertising data. The advertising set can either have been stopped by a timeout or because the specified number of advertising events has been reached.

Param adv
The advertising set object.

Param info
Information about the sent event.

void (***connected**)(struct bt_le_ext_adv *adv, struct *bt_le_ext_adv_connected_info* *info)

The advertising set has accepted a new connection.

This callback notifies the application that the advertising set has accepted a new connection.

Param adv
The advertising set object.

Param info
Information about the connected event.

void (***scanned**)(struct bt_le_ext_adv *adv, struct *bt_le_ext_adv_scanned_info* *info)

The advertising set has sent scan response data.

This callback notifies the application that the advertising set has has received a Scan Request packet, and has sent a Scan Response packet.

Param adv
The advertising set object.

Param addr
Information about the scanned event.

struct **bt_data**

#include <bluetooth.h> Bluetooth data.

Description of different data types that can be encoded into advertising data. Used to form arrays that are passed to the *bt_le_adv_start()* function.

struct **bt_le_adv_param**

#include <bluetooth.h> LE Advertising Parameters.

Public Members

uint8_t **id**

Local identity.

> **Note**
>
> When extended advertising CONFIG_BT_EXT_ADV is not enabled or not supported by the controller it is not possible to scan and advertise simultaneously using two different random addresses.

uint8_t **sid**

Advertising Set Identifier, valid range 0x00 - 0x0f.

> **Note**
>
> Requires *BT_LE_ADV_OPT_EXT_ADV*

uint8_t **secondary_max_skip**

Secondary channel maximum skip count.

Maximum advertising events the advertiser can skip before it must send advertising data on the secondary advertising channel.

> **Note**
>
> Requires *BT_LE_ADV_OPT_EXT_ADV*

uint32_t **options**

Bit-field of advertising options.

uint32_t **interval_min**

Minimum Advertising Interval (N * 0.625 milliseconds) Minimum Advertising Interval shall be less than or equal to the Maximum Advertising Interval.

The Minimum Advertising Interval and Maximum Advertising Interval should not be the same value (as stated in Bluetooth Core Spec 5.2, section 7.8.5) Range: 0x0020 to 0x4000

uint32_t **interval_max**

Maximum Advertising Interval (N * 0.625 milliseconds) Minimum Advertising Interval shall be less than or equal to the Maximum Advertising Interval.

The Minimum Advertising Interval and Maximum Advertising Interval should not be the same value (as stated in Bluetooth Core Spec 5.2, section 7.8.5) Range: 0x0020 to 0x4000

const *bt_addr_le_t* ***peer**

Directed advertising to peer.

When this parameter is set the advertiser will send directed advertising to the remote device.

The advertising type will either be high duty cycle, or low duty cycle if the BT_LE_ADV_OPT_DIR_MODE_LOW_DUTY option is enabled. When using *BT_LE_ADV_OPT_EXT_ADV* then only low duty cycle is allowed.

In case of connectable high duty cycle if the connection could not be established within the timeout the connected() callback will be called with the status set to BT_HCI_ERR_ADV_TIMEOUT.

struct **bt_le_per_adv_param**

#include <*bluetooth.h*>

Public Members

uint16_t **interval_min**

Minimum Periodic Advertising Interval (N * 1.25 ms)

Shall be greater or equal to BT_GAP_PER_ADV_MIN_INTERVAL and less or equal to interval_max.

uint16_t **interval_max**

Maximum Periodic Advertising Interval (N * 1.25 ms)

Shall be less or equal to BT_GAP_PER_ADV_MAX_INTERVAL and greater or equal to interval_min.

uint32_t **options**

Bit-field of periodic advertising options.

struct **bt_le_ext_adv_start_param**

#include <*bluetooth.h*>

Public Members

uint16_t **timeout**

Advertiser timeout (N * 10 ms).

Application will be notified by the advertiser sent callback. Set to zero for no timeout.

When using high duty cycle directed connectable advertising then this parameters must be set to a non-zero value less than or equal to the maximum of *BT_GAP_ADV_HIGH_DUTY_CYCLE_MAX_TIMEOUT*.

If privacy `CONFIG_BT_PRIVACY` is enabled then the timeout must be less than `CONFIG_BT_RPA_TIMEOUT`.

uint8_t **num_events**

Number of advertising events.

Application will be notified by the advertiser sent callback. Set to zero for no limit.

struct **bt_le_ext_adv_info**

#include <*bluetooth.h*> Advertising set info structure.

Public Members

int8_t **tx_power**
Currently selected Transmit Power (dBM).

const *bt_addr_le_t* ***addr**
Current local advertising address used.

struct **bt_le_per_adv_subevent_data_params**
#include <bluetooth.h>

Public Members

uint8_t **subevent**
The subevent to set data for.

uint8_t **response_slot_start**
The first response slot to listen to.

uint8_t **response_slot_count**
The number of response slots to listen to.

const struct *net_buf_simple* ***data**
The data to send.

struct **bt_le_per_adv_sync_synced_info**
#include <bluetooth.h>

Public Members

const *bt_addr_le_t* ***addr**
Advertiser LE address and type.

uint8_t **sid**
Advertiser SID.

uint16_t **interval**
Periodic advertising interval (N * 1.25 ms)

uint8_t **phy**
Advertiser PHY.

bool **recv_enabled**
True if receiving periodic advertisements, false otherwise.

uint16_t **service_data**

Service Data provided by the peer when sync is transferred.

Will always be 0 when the sync is locally created.

struct bt_conn ***conn**

Peer that transferred the periodic advertising sync.

Will always be 0 when the sync is locally created.

struct **bt_le_per_adv_sync_term_info**

#include <bluetooth.h>

Public Members

const *bt_addr_le_t* ***addr**

Advertiser LE address and type.

uint8_t **sid**

Advertiser SID.

uint8_t **reason**

Cause of periodic advertising termination.

struct **bt_le_per_adv_sync_recv_info**

#include <bluetooth.h>

Public Members

const *bt_addr_le_t* ***addr**

Advertiser LE address and type.

uint8_t **sid**

Advertiser SID.

int8_t **tx_power**

The TX power of the advertisement.

int8_t **rssi**

The RSSI of the advertisement excluding any CTE.

uint8_t **cte_type**

The Constant Tone Extension (CTE) of the advertisement (bt_df_cte_type)

struct **bt_le_per_adv_sync_state_info**

#include <bluetooth.h>

Public Members

bool **recv_enabled**

True if receiving periodic advertisements, false otherwise.

struct **bt_le_per_adv_sync_cb**

#include <bluetooth.h>

Public Members

void (***synced**)(struct bt_le_per_adv_sync *sync, struct *bt_le_per_adv_sync_synced_info* *info)

The periodic advertising has been successfully synced.

This callback notifies the application that the periodic advertising set has been successfully synced, and will now start to receive periodic advertising reports.

Param sync
The periodic advertising sync object.

Param info
Information about the sync event.

void (***term**)(struct bt_le_per_adv_sync *sync, const struct *bt_le_per_adv_sync_term_info* *info)

The periodic advertising sync has been terminated.

This callback notifies the application that the periodic advertising sync has been terminated, either by local request, remote request or because due to missing data, e.g. by being out of range or sync.

Param sync
The periodic advertising sync object.

void (***recv**)(struct bt_le_per_adv_sync *sync, const struct *bt_le_per_adv_sync_recv_info* *info, struct *net_buf_simple* *buf)

Periodic advertising data received.

This callback notifies the application of an periodic advertising report.

Param sync
The advertising set object.

Param info
Information about the periodic advertising event.

Param buf
Buffer containing the periodic advertising data. NULL if the controller failed to receive a subevent indication. Only happens if `CONFIG_BT_PER_ADV_SYNC_RSP` is enabled.

void (***state_changed**)(struct bt_le_per_adv_sync *sync, const struct *bt_le_per_adv_sync_state_info* *info)

The periodic advertising sync state has changed.

This callback notifies the application about changes to the sync state. Initialize sync and termination is handled by their individual callbacks, and won't be notified here.

Param sync
The periodic advertising sync object.

Param info
Information about the state change.

void (*biginfo)(struct bt_le_per_adv_sync *sync, const struct bt_iso_biginfo *biginfo)

BIGInfo advertising report received.

This callback notifies the application of a BIGInfo advertising report. This is received if the advertiser is broadcasting isochronous streams in a BIG. See iso.h for more information.

Param sync
The advertising set object.

Param biginfo
The BIGInfo report.

void (*cte_report_cb)(struct bt_le_per_adv_sync *sync, struct bt_df_per_adv_sync_iq_samples_report const *info)

Callback for IQ samples report collected when sampling CTE received with periodic advertising PDU.

Param sync
The periodic advertising sync object.

Param info
Information about the sync event.

struct bt_le_per_adv_sync_param

#include <bluetooth.h>

Public Members

bt_addr_le_t addr

Periodic Advertiser Address.

Only valid if not using the periodic advertising list (BT_LE_PER_ADV_SYNC_OPT_USE_PER_ADV_LIST)

uint8_t sid

Advertiser SID.

Only valid if not using the periodic advertising list (BT_LE_PER_ADV_SYNC_OPT_USE_PER_ADV_LIST)

uint32_t options

Bit-field of periodic advertising sync options.

uint16_t skip

Maximum event skip.

Maximum number of periodic advertising events that can be skipped after a successful receive. Range: 0x0000 to 0x01F3

uint16_t timeout

Synchronization timeout (N * 10 ms)

Synchronization timeout for the periodic advertising sync. Range 0x000A to 0x4000 (100 ms to 163840 ms)

struct **bt_le_per_adv_sync_info**

#include <bluetooth.h> Advertising set info structure.

Public Members

bt_addr_le_t **addr**

Periodic Advertiser Address.

uint8_t **sid**

Advertiser SID.

uint16_t **interval**

Periodic advertising interval (N * 1.25 ms)

uint8_t **phy**

Advertiser PHY.

struct **bt_le_per_adv_sync_transfer_param**

#include <bluetooth.h>

Public Members

uint16_t **skip**

Maximum event skip.

The number of periodic advertising packets that can be skipped after a successful receive.

uint16_t **timeout**

Synchronization timeout (N * 10 ms)

Synchronization timeout for the periodic advertising sync. Range 0x000A to 0x4000 (100 ms to 163840 ms)

uint32_t **options**

Periodic Advertising Sync Transfer options.

struct **bt_le_scan_param**

#include <bluetooth.h> LE scan parameters.

Public Members

uint8_t **type**

Scan type (BT_LE_SCAN_TYPE_ACTIVE or BT_LE_SCAN_TYPE_PASSIVE)

uint32_t **options**

Bit-field of scanning options.

uint16_t **interval**
Scan interval (N * 0.625 ms)

uint16_t **window**
Scan window (N * 0.625 ms)

uint16_t **timeout**
Scan timeout (N * 10 ms)

Application will be notified by the scan timeout callback. Set zero to disable timeout.

uint16_t **interval_coded**
Scan interval LE Coded PHY (N * 0.625 MS)

Set zero to use same as LE 1M PHY scan interval.

uint16_t **window_coded**
Scan window LE Coded PHY (N * 0.625 MS)

Set zero to use same as LE 1M PHY scan window.

struct **bt_le_scan_recv_info**
#include <bluetooth.h> LE advertisement and scan response packet information.

Public Members

const *bt_addr_le_t* ***addr**
Advertiser LE address and type.

If advertiser is anonymous then this address will be *BT_ADDR_LE_ANY*.

uint8_t **sid**
Advertising Set Identifier.

int8_t **rssi**
Strength of advertiser signal.

int8_t **tx_power**
Transmit power of the advertiser.

uint8_t **adv_type**
Advertising packet type.

Uses the BT_GAP_ADV_TYPE_* value.

May indicate that this is a scan response if the type is *BT_GAP_ADV_TYPE_SCAN_RSP*.

uint16_t **adv_props**
Advertising packet properties bitfield.

Uses the BT_GAP_ADV_PROP_* values. May indicate that this is a scan response if the value contains the *BT_GAP_ADV_PROP_SCAN_RESPONSE* bit.

uint16_t **interval**

Periodic advertising interval (N * 1.25 ms).

If 0 there is no periodic advertising.

uint8_t **primary_phy**

Primary advertising channel PHY.

uint8_t **secondary_phy**

Secondary advertising channel PHY.

struct **bt_le_scan_cb**

#include <bluetooth.h> Listener context for (LE) scanning.

Public Members

void (***recv**)(const struct *bt_le_scan_recv_info* *info, struct *net_buf_simple* *buf)

Advertisement packet and scan response received callback.

Param info
Advertiser packet and scan response information.

Param buf
Buffer containing advertiser data.

void (***timeout**)(void)

The scanner has stopped scanning after scan timeout.

struct **bt_le_oob_sc_data**

#include <bluetooth.h> LE Secure Connections pairing Out of Band data.

Public Members

uint8_t **r**[16]

Random Number.

uint8_t **c**[16]

Confirm Value.

struct **bt_le_oob**

#include <bluetooth.h> LE Out of Band information.

Public Members

bt_addr_le_t **addr**

LE address.

If privacy is enabled this is a Resolvable Private Address.

struct *bt_le_oob_sc_data* **le_sc_data**

LE Secure Connections pairing Out of Band data.

struct **bt_br_discovery_result**

#include <bluetooth.h> BR/EDR discovery result structure.

Public Members

bt_addr_t **addr**

Remote device address.

int8_t **rssi**

RSSI from inquiry.

uint8_t **cod**[3]

Class of Device.

uint8_t **eir**[240]

Extended Inquiry Response.

struct **bt_br_discovery_param**

#include <bluetooth.h> BR/EDR discovery parameters.

Public Members

uint8_t **length**

Maximum length of the discovery in units of 1.28 seconds.

Valid range is 0x01 - 0x30.

bool **limited**

True if limited discovery procedure is to be used.

struct **bt_br_oob**

#include <bluetooth.h>

Public Members

bt_addr_t **addr**

BR/EDR address.

struct **bt_bond_info**

#include <bluetooth.h> Information about a bond with a remote device.

Public Members

bt_addr_le_t **addr**
Address of the remote device.

struct **bt_le_per_adv_sync_subevent_params**
#include <bluetooth.h>

Public Members

uint16_t **properties**
Periodic Advertising Properties.
Bit 6 is include TxPower, all others RFU.

uint8_t **num_subevents**
Number of subevents to sync to.

uint8_t ***subevents**
The subevent(s) to synchronize with.
The array must have *num_subevents* elements.

struct **bt_le_per_adv_response_params**
#include <bluetooth.h>

Public Members

uint16_t **request_event**
The periodic event counter of the request the response is sent to.
bt_le_per_adv_sync_recv_info

> **Note**
> The response can be sent up to one periodic interval after the request was received.

uint8_t **request_subevent**
The subevent counter of the request the response is sent to.
bt_le_per_adv_sync_recv_info

uint8_t **response_subevent**
The subevent the response shall be sent in.

uint8_t **response_slot**
The response slot the response shall be sent in.

group bt_addr

Bluetooth device address definitions and utilities.

Defines

BT_ADDR_LE_PUBLIC

BT_ADDR_LE_RANDOM

BT_ADDR_LE_PUBLIC_ID

BT_ADDR_LE_RANDOM_ID

BT_ADDR_LE_UNRESOLVED

BT_ADDR_LE_ANONYMOUS

BT_ADDR_SIZE

Length in bytes of a standard Bluetooth address.

BT_ADDR_LE_SIZE

Length in bytes of an LE Bluetooth address.

Not packed, so no sizeof()

BT_ADDR_ANY

Bluetooth device “any” address, not a valid address.

BT_ADDR_NONE

Bluetooth device “none” address, not a valid address.

BT_ADDR_LE_ANY

Bluetooth LE device “any” address, not a valid address.

BT_ADDR_LE_NONE

Bluetooth LE device “none” address, not a valid address.

BT_ADDR_IS_RPA(a)

Check if a Bluetooth LE random address is resolvable private address.

BT_ADDR_IS_NRPA(a)

Check if a Bluetooth LE random address is a non-resolvable private address.

BT_ADDR_IS_STATIC(a)

Check if a Bluetooth LE random address is a static address.

BT_ADDR_SET_RPA(a)

Set a Bluetooth LE random address as a resolvable private address.

BT_ADDR_SET_NRPA(a)

Set a Bluetooth LE random address as a non-resolvable private address.

BT_ADDR_SET_STATIC(a)

Set a Bluetooth LE random address as a static address.

BT_ADDR_STR_LEN

Recommended length of user string buffer for Bluetooth address.

The recommended length guarantee the output of address conversion will not lose valuable information about address being processed.

BT_ADDR_LE_STR_LEN

Recommended length of user string buffer for Bluetooth LE address.

The recommended length guarantee the output of address conversion will not lose valuable information about address being processed.

Functions

static inline int **bt_addr_cmp**(const *bt_addr_t* *a, const *bt_addr_t* *b)

Compare Bluetooth device addresses.

Parameters

- **a** – First Bluetooth device address to compare
- **b** – Second Bluetooth device address to compare

Returns

negative value if *a* < *b*, 0 if *a* == *b*, else positive

static inline bool **bt_addr_eq**(const *bt_addr_t* *a, const *bt_addr_t* *b)

Determine equality of two Bluetooth device addresses.

Return values

- **true** – if the two addresses are equal
- **false** – otherwise

static inline int **bt_addr_le_cmp**(const *bt_addr_le_t* *a, const *bt_addr_le_t* *b)

Compare Bluetooth LE device addresses.

See also

bt_addr_le_eq

Parameters

- **a** – First Bluetooth LE device address to compare
- **b** – Second Bluetooth LE device address to compare

Returns

negative value if *a* < *b*, 0 if *a* == *b*, else positive

static inline bool **bt_addr_le_eq**(const *bt_addr_le_t* *a, const *bt_addr_le_t* *b)

Determine equality of two Bluetooth LE device addresses.

The Bluetooth LE addresses are equal if and only if both the types and the 48-bit addresses are numerically equal.

Return values

- `true` – if the two addresses are equal
- `false` – otherwise

static inline void **bt_addr_copy**(*bt_addr_t* *dst, const *bt_addr_t* *src)

Copy Bluetooth device address.

Parameters

- `dst` – Bluetooth device address destination buffer.
- `src` – Bluetooth device address source buffer.

static inline void **bt_addr_le_copy**(*bt_addr_le_t* *dst, const *bt_addr_le_t* *src)

Copy Bluetooth LE device address.

Parameters

- `dst` – Bluetooth LE device address destination buffer.
- `src` – Bluetooth LE device address source buffer.

int **bt_addr_le_create_nrpa**(*bt_addr_le_t* *addr)

Create a Bluetooth LE random non-resolvable private address.

int **bt_addr_le_create_static**(*bt_addr_le_t* *addr)

Create a Bluetooth LE random static address.

static inline bool **bt_addr_le_is_rpa**(const *bt_addr_le_t* *addr)

Check if a Bluetooth LE address is a random private resolvable address.

Parameters

- `addr` – Bluetooth LE device address.

Returns

true if address is a random private resolvable address.

static inline bool **bt_addr_le_is_identity**(const *bt_addr_le_t* *addr)

Check if a Bluetooth LE address is valid identity address.

Valid Bluetooth LE identity addresses are either public address or random static address.

Parameters

- `addr` – Bluetooth LE device address.

Returns

true if address is a valid identity address.

static inline int **bt_addr_to_str**(const *bt_addr_t* *addr, char *str, size_t len)

Converts binary Bluetooth address to string.

Parameters

- `addr` – Address of buffer containing binary Bluetooth address.
- `str` – Address of user buffer with enough room to store formatted string containing binary address.
- `len` – Length of data to be copied to user string buffer. Refer to BT_ADDR_STR_LEN about recommended value.

Returns

Number of successfully formatted bytes from binary address.

static inline int **bt_addr_le_to_str**(const *bt_addr_le_t* *addr, char *str, size_t len)

Converts binary LE Bluetooth address to string.

Parameters

- **addr** – Address of buffer containing binary LE Bluetooth address.
- **str** – Address of user buffer with enough room to store formatted string containing binary LE address.
- **len** – Length of data to be copied to user string buffer. Refer to BT_ADDR_LE_STR_LEN about recommended value.

Returns

Number of successfully formatted bytes from binary address.

int **bt_addr_from_str**(const char *str, *bt_addr_t* *addr)

Convert Bluetooth address from string to binary.

Parameters

- **str** – **[in]** The string representation of a Bluetooth address.
- **addr** – **[out]** Address of buffer to store the Bluetooth address

Return values

0 – Success. The parsed address is stored in addr.

Returns

-EINVAL Invalid address string. str is not a well-formed Bluetooth address.

int **bt_addr_le_from_str**(const char *str, const char *type, *bt_addr_le_t* *addr)

Convert LE Bluetooth address from string to binary.

Parameters

- **str** – **[in]** The string representation of an LE Bluetooth address.
- **type** – **[in]** The string representation of the LE Bluetooth address type.
- **addr** – **[out]** Address of buffer to store the LE Bluetooth address

Returns

Zero on success or (negative) error code otherwise.

Variables

const *bt_addr_t* **bt_addr_any**

const *bt_addr_t* **bt_addr_none**

const *bt_addr_le_t* **bt_addr_le_any**

const *bt_addr_le_t* **bt_addr_le_none**

struct **bt_addr_t**

#include <*addr.h*> Bluetooth Device Address.

struct **bt_addr_le_t**

#include <*addr.h*> Bluetooth LE Device Address.

group bt_gap_defines

Bluetooth Generic Access Profile defines and Assigned Numbers.

Company Identifiers (see Bluetooth Assigned Numbers)

BT_COMP_ID_LF

The Linux Foundation.

EIR/AD data type definitions

BT_DATA_FLAGS

AD flags.

BT_DATA_UUID16_SOME

16-bit UUID, more available

BT_DATA_UUID16_ALL

16-bit UUID, all listed

BT_DATA_UUID32_SOME

32-bit UUID, more available

BT_DATA_UUID32_ALL

32-bit UUID, all listed

BT_DATA_UUID128_SOME

128-bit UUID, more available

BT_DATA_UUID128_ALL

128-bit UUID, all listed

BT_DATA_NAME_SHORTENED

Shortened name.

BT_DATA_NAME_COMPLETE

Complete name.

BT_DATA_TX_POWER

Tx Power.

BT_DATA_SM_TK_VALUE

Security Manager TK Value.

BT_DATA_SM_OOB_FLAGS

Security Manager OOB Flags.

BT_DATA_PERIPHERAL_INT_RANGE
: Peripheral Connection Interval Range.

BT_DATA_SOLICIT16
: Solicit UUIDs, 16-bit.

BT_DATA_SOLICIT128
: Solicit UUIDs, 128-bit.

BT_DATA_SVC_DATA16
: Service data, 16-bit UUID.

BT_DATA_PUB_TARGET_ADDR
: Public Target Address.

BT_DATA_RAND_TARGET_ADDR
: Random Target Address.

BT_DATA_GAP_APPEARANCE
: GAP appearance.

BT_DATA_ADV_INT
: Advertising Interval.

BT_DATA_LE_BT_DEVICE_ADDRESS
: LE Bluetooth Device Address.

BT_DATA_LE_ROLE
: LE Role.

BT_DATA_SIMPLE_PAIRING_HASH
: Simple Pairing Hash C256.

BT_DATA_SIMPLE_PAIRING_RAND
: Simple Pairing Randomizer R256.

BT_DATA_SOLICIT32
: Solicit UUIDs, 32-bit.

BT_DATA_SVC_DATA32
: Service data, 32-bit UUID.

BT_DATA_SVC_DATA128
: Service data, 128-bit UUID.

BT_DATA_LE_SC_CONFIRM_VALUE
: LE SC Confirmation Value.

BT_DATA_LE_SC_RANDOM_VALUE
: LE SC Random Value.

BT_DATA_URI
: URI.

BT_DATA_INDOOR_POS
: Indoor Positioning.

BT_DATA_TRANS_DISCOVER_DATA
: Transport Discovery Data.

BT_DATA_LE_SUPPORTED_FEATURES
: LE Supported Features.

BT_DATA_CHANNEL_MAP_UPDATE_IND
: Channel Map Update Indication.

BT_DATA_MESH_PROV
: Mesh Provisioning PDU.

BT_DATA_MESH_MESSAGE
: Mesh Networking PDU.

BT_DATA_MESH_BEACON
: Mesh Beacon.

BT_DATA_BIG_INFO
: BIGInfo.

BT_DATA_BROADCAST_CODE
: Broadcast Code.

BT_DATA_CSIS_RSI
: CSIS Random Set ID type.

BT_DATA_ADV_INT_LONG
: Advertising Interval long.

BT_DATA_BROADCAST_NAME
: Broadcast Name.

BT_DATA_ENCRYPTED_AD_DATA
: Encrypted Advertising Data.

BT_DATA_3D_INFO
: 3D Information Data

BT_DATA_MANUFACTURER_DATA
: Manufacturer Specific Data.

BT_LE_AD_LIMITED
: Limited Discoverable.

BT_LE_AD_GENERAL
: General Discoverable.

BT_LE_AD_NO_BREDR
: BR/EDR not supported.

Appearance Values

Last Modified on 2023-01-05

BT_APPEARANCE_UNKNOWN
: Generic Unknown.

BT_APPEARANCE_GENERIC_PHONE
: Generic Phone.

BT_APPEARANCE_GENERIC_COMPUTER
: Generic Computer.

BT_APPEARANCE_COMPUTER_DESKTOP_WORKSTATION
: Desktop Workstation.

BT_APPEARANCE_COMPUTER_SERVER_CLASS
: Server-class Computer.

BT_APPEARANCE_COMPUTER_LAPTOP
: Laptop.

BT_APPEARANCE_COMPUTER_HANDHELD_PCPDA
: Handheld PC/PDA (clamshell)

BT_APPEARANCE_COMPUTER_PALMSIZE_PCPDA
: Palmsize PC/PDA.

BT_APPEARANCE_COMPUTER_WEARABLE_COMPUTER
: Wearable computer (watch size)

BT_APPEARANCE_COMPUTER_TABLET
: Tablet.

BT_APPEARANCE_COMPUTER_DOCKING_STATION
: Docking Station.

BT_APPEARANCE_COMPUTER_ALL_IN_ONE
: All in One.

BT_APPEARANCE_COMPUTER_BLADE_SERVER
: Blade Server.

BT_APPEARANCE_COMPUTER_CONVERTIBLE
: Convertible.

BT_APPEARANCE_COMPUTER_DETACHABLE
: Detachable.

BT_APPEARANCE_COMPUTER_IOT_GATEWAY
: IoT Gateway.

BT_APPEARANCE_COMPUTER_MINI_PC
: Mini PC.

BT_APPEARANCE_COMPUTER_STICK_PC
: Stick PC.

BT_APPEARANCE_GENERIC_WATCH
: Generic Watch.

BT_APPEARANCE_SPORTS_WATCH
: Sports Watch.

BT_APPEARANCE_SMARTWATCH
: Smartwatch.

BT_APPEARANCE_GENERIC_CLOCK
: Generic Clock.

BT_APPEARANCE_GENERIC_DISPLAY
: Generic Display.

BT_APPEARANCE_GENERIC_REMOTE
: Generic Remote Control.

BT_APPEARANCE_GENERIC_EYEGLASSES
: Generic Eye-glasses.

BT_APPEARANCE_GENERIC_TAG
: Generic Tag.

BT_APPEARANCE_GENERIC_KEYRING
: Generic Keyring.

BT_APPEARANCE_GENERIC_MEDIA_PLAYER
: Generic Media Player.

BT_APPEARANCE_GENERIC_BARCODE_SCANNER
: Generic Barcode Scanner.

BT_APPEARANCE_GENERIC_THERMOMETER
: Generic Thermometer.

BT_APPEARANCE_THERMOMETER_EAR
: Ear Thermometer.

BT_APPEARANCE_GENERIC_HEART_RATE
: Generic Heart Rate Sensor.

BT_APPEARANCE_HEART_RATE_BELT
: Heart Rate Belt.

BT_APPEARANCE_GENERIC_BLOOD_PRESSURE
: Generic Blood Pressure.

BT_APPEARANCE_BLOOD_PRESSURE_ARM
: Arm Blood Pressure.

BT_APPEARANCE_BLOOD_PRESSURE_WRIST
: Wrist Blood Pressure.

BT_APPEARANCE_GENERIC_HID
: Generic Human Interface Device.

BT_APPEARANCE_HID_KEYBOARD
: Keyboard.

BT_APPEARANCE_HID_MOUSE
: Mouse.

BT_APPEARANCE_HID_JOYSTICK
: Joystick.

BT_APPEARANCE_HID_GAMEPAD
: Gamepad.

BT_APPEARANCE_HID_DIGITIZER_TABLET
: Digitizer Tablet.

BT_APPEARANCE_HID_CARD_READER
: Card Reader.

`BT_APPEARANCE_HID_DIGITAL_PEN`
: Digital Pen.

`BT_APPEARANCE_HID_BARCODE_SCANNER`
: Barcode Scanner.

`BT_APPEARANCE_HID_TOUCHPAD`
: Touchpad.

`BT_APPEARANCE_HID_PRESENTATION_REMOTE`
: Presentation Remote.

`BT_APPEARANCE_GENERIC_GLUCOSE`
: Generic Glucose Meter.

`BT_APPEARANCE_GENERIC_WALKING`
: Generic Running Walking Sensor.

`BT_APPEARANCE_WALKING_IN_SHOE`
: In-Shoe Running Walking Sensor.

`BT_APPEARANCE_WALKING_ON_SHOE`
: On-Shoe Running Walking Sensor.

`BT_APPEARANCE_WALKING_ON_HIP`
: On-Hip Running Walking Sensor.

`BT_APPEARANCE_GENERIC_CYCLING`
: Generic Cycling.

`BT_APPEARANCE_CYCLING_COMPUTER`
: Cycling Computer.

`BT_APPEARANCE_CYCLING_SPEED`
: Speed Sensor.

`BT_APPEARANCE_CYCLING_CADENCE`
: Cadence Sensor.

`BT_APPEARANCE_CYCLING_POWER`
: Power Sensor.

`BT_APPEARANCE_CYCLING_SPEED_CADENCE`
: Speed and Cadence Sensor.

`BT_APPEARANCE_GENERIC_CONTROL_DEVICE`
: Generic Control Device.

BT_APPEARANCE_CONTROL_SWITCH
: Switch.

BT_APPEARANCE_CONTROL_MULTI_SWITCH
: Multi-switch.

BT_APPEARANCE_CONTROL_BUTTON
: Button.

BT_APPEARANCE_CONTROL_SLIDER
: Slider.

BT_APPEARANCE_CONTROL_ROTARY_SWITCH
: Rotary Switch.

BT_APPEARANCE_CONTROL_TOUCH_PANEL
: Touch Panel.

BT_APPEARANCE_CONTROL_SINGLE_SWITCH
: Single Switch.

BT_APPEARANCE_CONTROL_DOUBLE_SWITCH
: Double Switch.

BT_APPEARANCE_CONTROL_TRIPLE_SWITCH
: Triple Switch.

BT_APPEARANCE_CONTROL_BATTERY_SWITCH
: Battery Switch.

BT_APPEARANCE_CONTROL_ENERGY_HARVESTING_SWITCH
: Energy Harvesting Switch.

BT_APPEARANCE_CONTROL_PUSH_BUTTON
: Push Button.

BT_APPEARANCE_GENERIC_NETWORK_DEVICE
: Generic Network Device.

BT_APPEARANCE_NETWORK_ACCESS_POINT
: Access Point.

BT_APPEARANCE_NETWORK_MESH_DEVICE
: Mesh Device.

BT_APPEARANCE_NETWORK_MESH_PROXY
: Mesh Network Proxy.

BT_APPEARANCE_GENERIC_SENSOR
: Generic Sensor.

BT_APPEARANCE_SENSOR_MOTION
: Motion Sensor.

BT_APPEARANCE_SENSOR_AIR_QUALITY
: Air quality Sensor.

BT_APPEARANCE_SENSOR_TEMPERATURE
: Temperature Sensor.

BT_APPEARANCE_SENSOR_HUMIDITY
: Humidity Sensor.

BT_APPEARANCE_SENSOR_LEAK
: Leak Sensor.

BT_APPEARANCE_SENSOR_SMOKE
: Smoke Sensor.

BT_APPEARANCE_SENSOR_OCCUPANCY
: Occupancy Sensor.

BT_APPEARANCE_SENSOR_CONTACT
: Contact Sensor.

BT_APPEARANCE_SENSOR_CARBON_MONOXIDE
: Carbon Monoxide Sensor.

BT_APPEARANCE_SENSOR_CARBON_DIOXIDE
: Carbon Dioxide Sensor.

BT_APPEARANCE_SENSOR_AMBIENT_LIGHT
: Ambient Light Sensor.

BT_APPEARANCE_SENSOR_ENERGY
: Energy Sensor.

BT_APPEARANCE_SENSOR_COLOR_LIGHT
: Color Light Sensor.

BT_APPEARANCE_SENSOR_RAIN
: Rain Sensor.

BT_APPEARANCE_SENSOR_FIRE
: Fire Sensor.

BT_APPEARANCE_SENSOR_WIND
: Wind Sensor.

BT_APPEARANCE_SENSOR_PROXIMITY
: Proximity Sensor.

BT_APPEARANCE_SENSOR_MULTI
: Multi-Sensor.

BT_APPEARANCE_SENSOR_FLUSH_MOUNTED
: Flush Mounted Sensor.

BT_APPEARANCE_SENSOR_CEILING_MOUNTED
: Ceiling Mounted Sensor.

BT_APPEARANCE_SENSOR_WALL_MOUNTED
: Wall Mounted Sensor.

BT_APPEARANCE_MULTISENSOR
: Multisensor.

BT_APPEARANCE_SENSOR_ENERGY_METER
: Energy Meter.

BT_APPEARANCE_SENSOR_FLAME_DETECTOR
: Flame Detector.

BT_APPEARANCE_SENSOR_VEHICLE_TIRE_PRESSURE
: Vehicle Tire Pressure Sensor.

BT_APPEARANCE_GENERIC_LIGHT_FIXTURES
: Generic Light Fixtures.

BT_APPEARANCE_LIGHT_FIXTURES_WALL
: Wall Light.

BT_APPEARANCE_LIGHT_FIXTURES_CEILING
: Ceiling Light.

BT_APPEARANCE_LIGHT_FIXTURES_FLOOR
: Floor Light.

BT_APPEARANCE_LIGHT_FIXTURES_CABINET
: Cabinet Light.

BT_APPEARANCE_LIGHT_FIXTURES_DESK
: Desk Light.

`BT_APPEARANCE_LIGHT_FIXTURES_TROFFER`
: Troffer Light.

`BT_APPEARANCE_LIGHT_FIXTURES_PENDANT`
: Pendant Light.

`BT_APPEARANCE_LIGHT_FIXTURES_IN_GROUND`
: In-ground Light.

`BT_APPEARANCE_LIGHT_FIXTURES_FLOOD`
: Flood Light.

`BT_APPEARANCE_LIGHT_FIXTURES_UNDERWATER`
: Underwater Light.

`BT_APPEARANCE_LIGHT_FIXTURES_BOLLARD_WITH`
: Bollard with Light.

`BT_APPEARANCE_LIGHT_FIXTURES_PATHWAY`
: Pathway Light.

`BT_APPEARANCE_LIGHT_FIXTURES_GARDEN`
: Garden Light.

`BT_APPEARANCE_LIGHT_FIXTURES_POLE_TOP`
: Pole-top Light.

`BT_APPEARANCE_SPOT_LIGHT`
: Spotlight.

`BT_APPEARANCE_LIGHT_FIXTURES_LINEAR`
: Linear Light.

`BT_APPEARANCE_LIGHT_FIXTURES_STREET`
: Street Light.

`BT_APPEARANCE_LIGHT_FIXTURES_SHELVES`
: Shelves Light.

`BT_APPEARANCE_LIGHT_FIXTURES_BAY`
: Bay Light.

`BT_APPEARANCE_LIGHT_FIXTURES_EMERGENCY_EXIT`
: Emergency Exit Light.

`BT_APPEARANCE_LIGHT_FIXTURES_CONTROLLER`
: Light Controller.

BT_APPEARANCE_LIGHT_FIXTURES_DRIVER
: Light Driver.

BT_APPEARANCE_LIGHT_FIXTURES_BULB
: Bulb.

BT_APPEARANCE_LIGHT_FIXTURES_LOW_BAY
: Low-bay Light.

BT_APPEARANCE_LIGHT_FIXTURES_HIGH_BAY
: High-bay Light.

BT_APPEARANCE_GENERIC_FAN
: Generic Fan.

BT_APPEARANCE_FAN_CEILING
: Ceiling Fan.

BT_APPEARANCE_FAN_AXIAL
: Axial Fan.

BT_APPEARANCE_FAN_EXHAUST
: Exhaust Fan.

BT_APPEARANCE_FAN_PEDESTAL
: Pedestal Fan.

BT_APPEARANCE_FAN_DESK
: Desk Fan.

BT_APPEARANCE_FAN_WALL
: Wall Fan.

BT_APPEARANCE_GENERIC_HVAC
: Generic HVAC.

BT_APPEARANCE_HVAC_THERMOSTAT
: Thermostat.

BT_APPEARANCE_HVAC_HUMIDIFIER
: Humidifier.

BT_APPEARANCE_HVAC_DEHUMIDIFIER
: De-humidifier.

BT_APPEARANCE_HVAC_HEATER
: Heater.

BT_APPEARANCE_HVAC_RADIATOR
: Radiator.

BT_APPEARANCE_HVAC_BOILER
: Boiler.

BT_APPEARANCE_HVAC_HEAT_PUMP
: Heat Pump.

BT_APPEARANCE_HVAC_INFRARED_HEATER
: Infrared Heater.

BT_APPEARANCE_HVAC_RADIANT_PANEL_HEATER
: Radiant Panel Heater.

BT_APPEARANCE_HVAC_FAN_HEATER
: Fan Heater.

BT_APPEARANCE_HVAC_AIR_CURTAIN
: Air Curtain.

BT_APPEARANCE_GENERIC_AIR_CONDITIONING
: Generic Air Conditioning.

BT_APPEARANCE_GENERIC_HUMIDIFIER
: Generic Humidifier.

BT_APPEARANCE_GENERIC_HEATING
: Generic Heating.

BT_APPEARANCE_HEATING_RADIATOR
: Radiator.

BT_APPEARANCE_HEATING_BOILER
: Boiler.

BT_APPEARANCE_HEATING_HEAT_PUMP
: Heat Pump.

BT_APPEARANCE_HEATING_INFRARED_HEATER
: Infrared Heater.

BT_APPEARANCE_HEATING_RADIANT_PANEL_HEATER
: Radiant Panel Heater.

BT_APPEARANCE_HEATING_FAN_HEATER
: Fan Heater.

BT_APPEARANCE_HEATING_AIR_CURTAIN
: Air Curtain.

BT_APPEARANCE_GENERIC_ACCESS_CONTROL
: Generic Access Control.

BT_APPEARANCE_CONTROL_ACCESS_DOOR
: Access Door.

BT_APPEARANCE_CONTROL_GARAGE_DOOR
: Garage Door.

BT_APPEARANCE_CONTROL_EMERGENCY_EXIT_DOOR
: Emergency Exit Door.

BT_APPEARANCE_CONTROL_ACCESS_LOCK
: Access Lock.

BT_APPEARANCE_CONTROL_ELEVATOR
: Elevator.

BT_APPEARANCE_CONTROL_WINDOW
: Window.

BT_APPEARANCE_CONTROL_ENTRANCE_GATE
: Entrance Gate.

BT_APPEARANCE_CONTROL_DOOR_LOCK
: Door Lock.

BT_APPEARANCE_CONTROL_LOCKER
: Locker.

BT_APPEARANCE_GENERIC_MOTORIZED_DEVICE
: Generic Motorized Device.

BT_APPEARANCE_MOTORIZED_GATE
: Motorized Gate.

BT_APPEARANCE_MOTORIZED_AWNING
: Awning.

BT_APPEARANCE_MOTORIZED_BLINDS_OR_SHADES
: Blinds or Shades.

BT_APPEARANCE_MOTORIZED_CURTAINS
: Curtains.

BT_APPEARANCE_MOTORIZED_SCREEN
: Screen.

BT_APPEARANCE_GENERIC_POWER_DEVICE
: Generic Power Device.

BT_APPEARANCE_POWER_OUTLET
: Power Outlet.

BT_APPEARANCE_POWER_STRIP
: Power Strip.

BT_APPEARANCE_POWER_PLUG
: Plug.

BT_APPEARANCE_POWER_SUPPLY
: Power Supply.

BT_APPEARANCE_POWER_LED_DRIVER
: LED Driver.

BT_APPEARANCE_POWER_FLUORESCENT_LAMP_GEAR
: Fluorescent Lamp Gear.

BT_APPEARANCE_POWER_HID_LAMP_GEAR
: HID Lamp Gear.

BT_APPEARANCE_POWER_CHARGE_CASE
: Charge Case.

BT_APPEARANCE_POWER_POWER_BANK
: Power Bank.

BT_APPEARANCE_GENERIC_LIGHT_SOURCE
: Generic Light Source.

BT_APPEARANCE_LIGHT_SOURCE_INCANDESCENT_BULB
: Incandescent Light Bulb.

BT_APPEARANCE_LIGHT_SOURCE_LED_LAMP
: LED Lamp.

BT_APPEARANCE_LIGHT_SOURCE_HID_LAMP
: HID Lamp.

BT_APPEARANCE_LIGHT_SOURCE_FLUORESCENT_LAMP
: Fluorescent Lamp.

BT_APPEARANCE_LIGHT_SOURCE_LED_ARRAY
: LED Array.

BT_APPEARANCE_LIGHT_SOURCE_MULTICOLOR_LED_ARRAY
: Multi-Color LED Array.

BT_APPEARANCE_LIGHT_SOURCE_LOW_VOLTAGE_HALOGEN
: Low voltage halogen.

BT_APPEARANCE_LIGHT_SOURCE_OLED
: Organic light emitting diode.

BT_APPEARANCE_GENERIC_WINDOW_COVERING
: Generic Window Covering.

BT_APPEARANCE_WINDOW_SHADES
: Window Shades.

BT_APPEARANCE_WINDOW_BLINDS
: Window Blinds.

BT_APPEARANCE_WINDOW_AWNING
: Window Awning.

BT_APPEARANCE_WINDOW_CURTAIN
: Window Curtain.

BT_APPEARANCE_WINDOW_EXTERIOR_SHUTTER
: Exterior Shutter.

BT_APPEARANCE_WINDOW_EXTERIOR_SCREEN
: Exterior Screen.

BT_APPEARANCE_GENERIC_AUDIO_SINK
: Generic Audio Sink.

BT_APPEARANCE_AUDIO_SINK_STANDALONE_SPEAKER
: Standalone Speaker.

BT_APPEARANCE_AUDIO_SINK_SOUNDBAR
: Soundbar.

BT_APPEARANCE_AUDIO_SINK_BOOKSHELF_SPEAKER
: Bookshelf Speaker.

BT_APPEARANCE_AUDIO_SINK_STANDMOUNTED_SPEAKER
: Standmounted Speaker.

BT_APPEARANCE_AUDIO_SINK_SPEAKERPHONE
: Speakerphone.

BT_APPEARANCE_GENERIC_AUDIO_SOURCE
: Generic Audio Source.

BT_APPEARANCE_AUDIO_SOURCE_MICROPHONE
: Microphone.

BT_APPEARANCE_AUDIO_SOURCE_ALARM
: Alarm.

BT_APPEARANCE_AUDIO_SOURCE_BELL
: Bell.

BT_APPEARANCE_AUDIO_SOURCE_HORN
: Horn.

BT_APPEARANCE_AUDIO_SOURCE_BROADCASTING_DEVICE
: Broadcasting Device.

BT_APPEARANCE_AUDIO_SOURCE_SERVICE_DESK
: Service Desk.

BT_APPEARANCE_AUDIO_SOURCE_KIOSK
: Kiosk.

BT_APPEARANCE_AUDIO_SOURCE_BROADCASTING_ROOM
: Broadcasting Room.

BT_APPEARANCE_AUDIO_SOURCE_AUDITORIUM
: Auditorium.

BT_APPEARANCE_GENERIC_MOTORIZED_VEHICLE
: Generic Motorized Vehicle.

BT_APPEARANCE_VEHICLE_CAR
: Car.

BT_APPEARANCE_VEHICLE_LARGE_GOODS
: Large Goods Vehicle.

BT_APPEARANCE_VEHICLE_TWO_WHEELED
: 2-Wheeled Vehicle

BT_APPEARANCE_VEHICLE_MOTORBIKE
: Motorbike.

`BT_APPEARANCE_VEHICLE_SCOOTER`
: Scooter.

`BT_APPEARANCE_VEHICLE_MOPED`
: Moped.

`BT_APPEARANCE_VEHICLE_THREE_WHEELED`
: 3-Wheeled Vehicle

`BT_APPEARANCE_VEHICLE_LIGHT`
: Light Vehicle.

`BT_APPEARANCE_VEHICLE_QUAD_BIKE`
: Quad Bike.

`BT_APPEARANCE_VEHICLE_MINIBUS`
: Minibus.

`BT_APPEARANCE_VEHICLE_BUS`
: Bus.

`BT_APPEARANCE_VEHICLE_TROLLEY`
: Trolley.

`BT_APPEARANCE_VEHICLE_AGRICULTURAL`
: Agricultural Vehicle.

`BT_APPEARANCE_VEHICLE_CAMPER_OR_CARAVAN`
: Camper/Caravan.

`BT_APPEARANCE_VEHICLE_RECREATIONAL`
: Recreational Vehicle/Motor Home.

`BT_APPEARANCE_GENERIC_DOMESTIC_APPLIANCE`
: Generic Domestic Appliance.

`BT_APPEARANCE_APPLIANCE_REFRIGERATOR`
: Refrigerator.

`BT_APPEARANCE_APPLIANCE_FREEZER`
: Freezer.

`BT_APPEARANCE_APPLIANCE_OVEN`
: Oven.

`BT_APPEARANCE_APPLIANCE_MICROWAVE`
: Microwave.

BT_APPEARANCE_APPLIANCE_TOASTER
: Toaster.

BT_APPEARANCE_APPLIANCE_WASHING_MACHINE
: Washing Machine.

BT_APPEARANCE_APPLIANCE_DRYER
: Dryer.

BT_APPEARANCE_APPLIANCE_COFFEE_MAKER
: Coffee maker.

BT_APPEARANCE_APPLIANCE_CLOTHES_IRON
: Clothes iron.

BT_APPEARANCE_APPLIANCE_CURLING_IRON
: Curling iron.

BT_APPEARANCE_APPLIANCE_HAIR_DRYER
: Hair dryer.

BT_APPEARANCE_APPLIANCE_VACUUM_CLEANER
: Vacuum cleaner.

BT_APPEARANCE_APPLIANCE_ROBOTIC_VACUUM_CLEANER
: Robotic vacuum cleaner.

BT_APPEARANCE_APPLIANCE_RICE_COOKER
: Rice cooker.

BT_APPEARANCE_APPLIANCE_CLOTHES_STEAMER
: Clothes steamer.

BT_APPEARANCE_GENERIC_WEARABLE_AUDIO_DEVICE
: Generic Wearable Audio Device.

BT_APPEARANCE_WEARABLE_AUDIO_DEVICE_EARBUD
: Earbud.

BT_APPEARANCE_WEARABLE_AUDIO_DEVICE_HEADSET
: Headset.

BT_APPEARANCE_WEARABLE_AUDIO_DEVICE_HEADPHONES
: Headphones.

BT_APPEARANCE_WEARABLE_AUDIO_DEVICE_NECK_BAND
: Neck Band.

BT_APPEARANCE_GENERIC_AIRCRAFT
: Generic Aircraft.

BT_APPEARANCE_AIRCRAFT_LIGHT
: Light Aircraft.

BT_APPEARANCE_AIRCRAFT_MICROLIGHT
: Microlight.

BT_APPEARANCE_AIRCRAFT_PARAGLIDER
: Paraglider.

BT_APPEARANCE_AIRCRAFT_LARGE_PASSENGER
: Large Passenger Aircraft.

BT_APPEARANCE_GENERIC_AV_EQUIPMENT
: Generic AV Equipment.

BT_APPEARANCE_AV_EQUIPMENT_AMPLIFIER
: Amplifier.

BT_APPEARANCE_AV_EQUIPMENT_RECEIVER
: Receiver.

BT_APPEARANCE_AV_EQUIPMENT_RADIO
: Radio.

BT_APPEARANCE_AV_EQUIPMENT_TUNER
: Tuner.

BT_APPEARANCE_AV_EQUIPMENT_TURNTABLE
: Turntable.

BT_APPEARANCE_AV_EQUIPMENT_CD_PLAYER
: CD Player.

BT_APPEARANCE_AV_EQUIPMENT_DVD_PLAYER
: DVD Player.

BT_APPEARANCE_AV_EQUIPMENT_BLURAY_PLAYER
: Bluray Player.

BT_APPEARANCE_AV_EQUIPMENT_OPTICAL_DISC_PLAYER
: Optical Disc Player.

BT_APPEARANCE_AV_EQUIPMENT_SET_TOP_BOX
: Set-Top Box.

BT_APPEARANCE_GENERIC_DISPLAY_EQUIPMENT
: Generic Display Equipment.

BT_APPEARANCE_DISPLAY_EQUIPMENT_TELEVISION
: Television.

BT_APPEARANCE_DISPLAY_EQUIPMENT_MONITOR
: Monitor.

BT_APPEARANCE_DISPLAY_EQUIPMENT_PROJECTOR
: Projector.

BT_APPEARANCE_GENERIC_HEARING_AID
: Generic Hearing aid.

BT_APPEARANCE_HEARING_AID_IN_EAR
: In-ear hearing aid.

BT_APPEARANCE_HEARING_AID_BEHIND_EAR
: Behind-ear hearing aid.

BT_APPEARANCE_HEARING_AID_COCHLEAR_IMPLANT
: Cochlear Implant.

BT_APPEARANCE_GENERIC_GAMING
: Generic Gaming.

BT_APPEARANCE_HOME_VIDEO_GAME_CONSOLE
: Home Video Game Console.

BT_APPEARANCE_PORTABLE_HANDHELD_CONSOLE
: Portable handheld console.

BT_APPEARANCE_GENERIC_SIGNAGE
: Generic Signage.

BT_APPEARANCE_SIGNAGE_DIGITAL
: Digital Signage.

BT_APPEARANCE_SIGNAGE_ELECTRONIC_LABEL
: Electronic Label.

BT_APPEARANCE_GENERIC_PULSE_OXIMETER
: Generic Pulse Oximeter.

BT_APPEARANCE_PULSE_OXIMETER_FINGERTIP
: Fingertip Pulse Oximeter.

BT_APPEARANCE_PULSE_OXIMETER_WRIST
: Wrist Worn Pulse Oximeter.

BT_APPEARANCE_GENERIC_WEIGHT_SCALE
: Generic Weight Scale.

BT_APPEARANCE_GENERIC_PERSONAL_MOBILITY_DEVICE
: Generic Personal Mobility Device.

BT_APPEARANCE_MOBILITY_POWERED_WHEELCHAIR
: Powered Wheelchair.

BT_APPEARANCE_MOBILITY_SCOOTER
: Mobility Scooter.

BT_APPEARANCE_CONTINUOUS_GLUCOSE_MONITOR
: Continuous Glucose Monitor.

BT_APPEARANCE_GENERIC_INSULIN_PUMP
: Generic Insulin Pump.

BT_APPEARANCE_INSULIN_PUMP_DURABLE
: Insulin Pump, durable pump.

BT_APPEARANCE_INSULIN_PUMP_PATCH
: Insulin Pump, patch pump.

BT_APPEARANCE_INSULIN_PEN
: Insulin Pen.

BT_APPEARANCE_GENERIC_MEDICATION_DELIVERY
: Generic Medication Delivery.

BT_APPEARANCE_GENERIC_SPIROMETER
: Generic Spirometer.

BT_APPEARANCE_SPIROMETER_HANDHELD
: Handheld Spirometer.

BT_APPEARANCE_GENERIC_OUTDOOR_SPORTS
: Generic Outdoor Sports Activity.

BT_APPEARANCE_OUTDOOR_SPORTS_LOCATION
: Location Display.

BT_APPEARANCE_OUTDOOR_SPORTS_LOCATION_AND_NAV
: Location and Navigation Display.

BT_APPEARANCE_OUTDOOR_SPORTS_LOCATION_POD
: Location Pod.

BT_APPEARANCE_OUTDOOR_SPORTS_LOCATION_POD_AND_NAV
: Location and Navigation Pod.

Defined GAP timers

BT_GAP_SCAN_FAST_INTERVAL_MIN

BT_GAP_SCAN_FAST_INTERVAL

BT_GAP_SCAN_FAST_WINDOW

BT_GAP_SCAN_SLOW_INTERVAL_1

BT_GAP_SCAN_SLOW_WINDOW_1

BT_GAP_SCAN_SLOW_INTERVAL_2

BT_GAP_SCAN_SLOW_WINDOW_2

BT_GAP_ADV_FAST_INT_MIN_1

BT_GAP_ADV_FAST_INT_MAX_1

BT_GAP_ADV_FAST_INT_MIN_2

BT_GAP_ADV_FAST_INT_MAX_2

BT_GAP_ADV_SLOW_INT_MIN

BT_GAP_ADV_SLOW_INT_MAX

BT_GAP_PER_ADV_FAST_INT_MIN_1

BT_GAP_PER_ADV_FAST_INT_MAX_1

BT_GAP_PER_ADV_FAST_INT_MIN_2

BT_GAP_PER_ADV_FAST_INT_MAX_2

BT_GAP_PER_ADV_SLOW_INT_MIN

BT_GAP_PER_ADV_SLOW_INT_MAX

BT_GAP_INIT_CONN_INT_MIN

BT_GAP_INIT_CONN_INT_MAX

Defines

BT_GAP_ADV_MAX_ADV_DATA_LEN
: Maximum advertising data length.

BT_GAP_ADV_MAX_EXT_ADV_DATA_LEN
: Maximum extended advertising data length.

> **Note**
>
> The maximum advertising data length that can be sent by an extended advertiser is defined by the controller.

BT_GAP_TX_POWER_INVALID

BT_GAP_RSSI_INVALID

BT_GAP_SID_INVALID

BT_GAP_NO_TIMEOUT

BT_GAP_ADV_HIGH_DUTY_CYCLE_MAX_TIMEOUT

BT_GAP_DATA_LEN_DEFAULT
: Default data length.

BT_GAP_DATA_LEN_MAX
: Maximum data length.

BT_GAP_DATA_TIME_DEFAULT
: Default data time.

BT_GAP_DATA_TIME_MAX
: Maximum data time.

BT_GAP_SID_MAX
: Maximum advertising set number.

BT_GAP_PER_ADV_MAX_SKIP
: Maximum number of consecutive periodic advertisement events that can be skipped after a successful receive.

BT_GAP_PER_ADV_MIN_TIMEOUT

Minimum Periodic Advertising Timeout (N * 10 ms)

BT_GAP_PER_ADV_MAX_TIMEOUT

Maximum Periodic Advertising Timeout (N * 10 ms)

BT_GAP_PER_ADV_MIN_INTERVAL

Minimum Periodic Advertising Interval (N * 1.25 ms)

BT_GAP_PER_ADV_MAX_INTERVAL

Maximum Periodic Advertising Interval (N * 1.25 ms)

BT_GAP_PER_ADV_INTERVAL_TO_MS(interval)

Convert periodic advertising interval (N * 1.25 ms) to milliseconds.

5 / 4 represents 1.25 ms unit.

BT_LE_SUPP_FEAT_40_ENCODE(w64)

Encode 40 least significant bits of 64-bit LE Supported Features into array values in little-endian format.

Helper macro to encode 40 least significant bits of 64-bit LE Supported Features value into advertising data. The number of bits that are encoded is a number of LE Supported Features defined by BT 5.3 Core specification.

Example of how to encode the 0x000000DFF00DF00D into advertising data.

```
BT_DATA_BYTES(BT_DATA_LE_SUPPORTED_FEATURES, BT_LE_SUPP_FEAT_40_
↪ENCODE(0x000000DFF00DF00D))
```

Parameters

- **w64** – LE Supported Features value (64-bits)

Returns

The comma separated values for LE Supported Features value that may be used directly as an argument for *BT_DATA_BYTES*.

BT_LE_SUPP_FEAT_32_ENCODE(w64)

Encode 4 least significant bytes of 64-bit LE Supported Features into 4 bytes long array of values in little-endian format.

Helper macro to encode 64-bit LE Supported Features value into advertising data. The macro encodes 4 least significant bytes into advertising data. Other 4 bytes are not encoded.

Example of how to encode the 0x000000DFF00DF00D into advertising data.

```
BT_DATA_BYTES(BT_DATA_LE_SUPPORTED_FEATURES, BT_LE_SUPP_FEAT_32_
↪ENCODE(0x000000DFF00DF00D))
```

Parameters

- **w64** – LE Supported Features value (64-bits)

Returns

The comma separated values for LE Supported Features value that may be used directly as an argument for *BT_DATA_BYTES*.

BT_LE_SUPP_FEAT_24_ENCODE(w64)

Encode 3 least significant bytes of 64-bit LE Supported Features into 3 bytes long array of values in little-endian format.

Helper macro to encode 64-bit LE Supported Features value into advertising data. The macro encodes 3 least significant bytes into advertising data. Other 5 bytes are not encoded.

Example of how to encode the 0x000000DFF00DF00D into advertising data.

```
BT_DATA_BYTES(BT_DATA_LE_SUPPORTED_FEATURES, BT_LE_SUPP_FEAT_24_
↪ENCODE(0x000000DFF00DF00D))
```

Parameters

- **w64** – LE Supported Features value (64-bits)

Returns

The comma separated values for LE Supported Features value that may be used directly as an argument for *BT_DATA_BYTES*.

BT_LE_SUPP_FEAT_16_ENCODE(w64)

Encode 2 least significant bytes of 64-bit LE Supported Features into 2 bytes long array of values in little-endian format.

Helper macro to encode 64-bit LE Supported Features value into advertising data. The macro encodes 3 least significant bytes into advertising data. Other 6 bytes are not encoded.

Example of how to encode the 0x000000DFF00DF00D into advertising data.

```
BT_DATA_BYTES(BT_DATA_LE_SUPPORTED_FEATURES, BT_LE_SUPP_FEAT_16_
↪ENCODE(0x000000DFF00DF00D))
```

Parameters

- **w64** – LE Supported Features value (64-bits)

Returns

The comma separated values for LE Supported Features value that may be used directly as an argument for *BT_DATA_BYTES*.

BT_LE_SUPP_FEAT_8_ENCODE(w64)

Encode the least significant byte of 64-bit LE Supported Features into single byte long array.

Helper macro to encode 64-bit LE Supported Features value into advertising data. The macro encodes the least significant byte into advertising data. Other 7 bytes are not encoded.

Example of how to encode the 0x000000DFF00DF00D into advertising data.

```
BT_DATA_BYTES(BT_DATA_LE_SUPPORTED_FEATURES, BT_LE_SUPP_FEAT_8_
↪ENCODE(0x000000DFF00DF00D))
```

Parameters

- **w64** – LE Supported Features value (64-bits)

Returns

The value of least significant byte of LE Supported Features value that may be used directly as an argument for *BT_DATA_BYTES*.

BT_LE_SUPP_FEAT_VALIDATE(w64)

Validate whether LE Supported Features value does not use bits that are reserved for future use.

Helper macro to check if w64 has zeros as bits 40-63. The macro is compliant with BT 5.3 Core Specification where bits 0-40 has assigned values. In case of invalid value, build time error is reported.

Enums

LE PHY types.

Values:

enumerator BT_GAP_LE_PHY_NONE = 0

Convenience macro for when no PHY is set.

enumerator BT_GAP_LE_PHY_1M = *BIT*(0)

LE 1M PHY.

enumerator BT_GAP_LE_PHY_2M = *BIT*(1)

LE 2M PHY.

enumerator BT_GAP_LE_PHY_CODED = *BIT*(2)

LE Coded PHY.

Advertising PDU types.

Values:

enumerator BT_GAP_ADV_TYPE_ADV_IND = 0x00

Scannable and connectable advertising.

enumerator BT_GAP_ADV_TYPE_ADV_DIRECT_IND = 0x01

Directed connectable advertising.

enumerator BT_GAP_ADV_TYPE_ADV_SCAN_IND = 0x02

Non-connectable and scannable advertising.

enumerator BT_GAP_ADV_TYPE_ADV_NONCONN_IND = 0x03

Non-connectable and non-scannable advertising.

enumerator BT_GAP_ADV_TYPE_SCAN_RSP = 0x04

Additional advertising data requested by an active scanner.

enumerator BT_GAP_ADV_TYPE_EXT_ADV = 0x05

Extended advertising, see advertising properties.

Advertising PDU properties.

Values:

enumerator BT_GAP_ADV_PROP_CONNECTABLE = *BIT*(0)
Connectable advertising.

enumerator BT_GAP_ADV_PROP_SCANNABLE = *BIT*(1)
Scannable advertising.

enumerator BT_GAP_ADV_PROP_DIRECTED = *BIT*(2)
Directed advertising.

enumerator BT_GAP_ADV_PROP_SCAN_RESPONSE = *BIT*(3)
Additional advertising data requested by an active scanner.

enumerator BT_GAP_ADV_PROP_EXT_ADV = *BIT*(4)
Extended advertising.

Constant Tone Extension (CTE) types.

Values:

enumerator BT_GAP_CTE_AOA = 0x00
Angle of Arrival.

enumerator BT_GAP_CTE_AOD_1US = 0x01
Angle of Departure with 1 us slots.

enumerator BT_GAP_CTE_AOD_2US = 0x02
Angle of Departure with 2 us slots.

enumerator BT_GAP_CTE_NONE = 0xFF
No extensions.

Peripheral sleep clock accuracy (SCA) in ppm (parts per million)

Values:

enumerator BT_GAP_SCA_UNKNOWN = 0
Unknown.

enumerator BT_GAP_SCA_251_500 = 0
251 ppm to 500 ppm

enumerator BT_GAP_SCA_151_250 = 1
151 ppm to 250 ppm

enumerator **BT_GAP_SCA_101_150** = 2
101 ppm to 150 ppm

enumerator **BT_GAP_SCA_76_100** = 3
76 ppm to 100 ppm

enumerator **BT_GAP_SCA_51_75** = 4
51 ppm to 75 ppm

enumerator **BT_GAP_SCA_31_50** = 5
31 ppm to 50 ppm

enumerator **BT_GAP_SCA_21_30** = 6
21 ppm to 30 ppm

enumerator **BT_GAP_SCA_0_20** = 7
0 ppm to 20 ppm

Bluetooth: Isochronous Channels

Commands

```
iso --help
iso - Bluetooth ISO shell commands
Subcommands:
   cig_create  :[dir=tx,rx,txrx] [interval] [packing] [framing] [latency] [sdu]
               [phy] [rtn]
   cig_term    :Terminate the CIG
   connect     :Connect ISO Channel
   listen      :<dir=tx,rx,txrx> [security level]
   send        :Send to ISO Channel [count]
   disconnect  :Disconnect ISO Channel
   create-big  :Create a BIG as a broadcaster [enc <broadcast code>]
   broadcast   :Broadcast on ISO channels
   sync-big    :Synchronize to a BIG as a receiver <BIS bitfield> [mse] [timeout]
               [enc <broadcast code>]
   term-big    :Terminate a BIG
```

1. [Central] Create CIG:

Requires to be connected:

```
uart:~$ iso cig_create
CIG created
```

2. [Peripheral] Listen to ISO connections

```
uart:~$ iso listen txrx
```

3. [Central] Connect ISO channel:

```
uart:~$ iso connect
ISO Connect pending...
ISO Channel 0x20000f88 connected
```

4. Send data:

```
uart:~$ iso send
send: 40 bytes of data
ISO sending...
```

5. Disconnect ISO channel:

```
uart:~$ iso disconnect
ISO Disconnect pending...
ISO Channel 0x20000f88 disconnected with reason 0x16
```

Generic Attribute Profile (GATT) GATT layer manages the service database providing APIs for service registration and attribute declaration.

Services can be registered using *bt_gatt_service_register()* API which takes the *bt_gatt_service* struct that provides the list of attributes the service contains. The helper macro *BT_GATT_SERVICE()* can be used to declare a service.

Attributes can be declared using the *bt_gatt_attr* struct or using one of the helper macros:

BT_GATT_PRIMARY_SERVICE
: Declares a Primary Service.

BT_GATT_SECONDARY_SERVICE
: Declares a Secondary Service.

BT_GATT_INCLUDE_SERVICE
: Declares a Include Service.

BT_GATT_CHARACTERISTIC
: Declares a Characteristic.

BT_GATT_DESCRIPTOR
: Declares a Descriptor.

BT_GATT_ATTRIBUTE
: Declares an Attribute.

BT_GATT_CCC
: Declares a Client Characteristic Configuration.

BT_GATT_CEP
: Declares a Characteristic Extended Properties.

BT_GATT_CUD
: Declares a Characteristic User Format.

Each attribute contain a `uuid`, which describes their type, a `read` callback, a `write` callback and a set of permission. Both read and write callbacks can be set to NULL if the attribute permission don't allow their respective operations.

Note

32-bit UUIDs are not supported in GATT. All 32-bit UUIDs shall be converted to 128-bit UUIDs when the UUID is contained in an ATT PDU.

Note

Attribute `read` and `write` callbacks are called directly from RX Thread thus it is not recommended to block for long periods of time in them.

Attribute value changes can be notified using *bt_gatt_notify()* API, alternatively there is *bt_gatt_notify_cb()* where it is possible to pass a callback to be called when it is necessary to know the exact instant when the data has been transmitted over the air. Indications are supported by *bt_gatt_indicate()* API.

Client procedures can be enabled with the configuration option: CONFIG_BT_GATT_CLIENT

Discover procedures can be initiated with the use of *bt_gatt_discover()* API which takes the *bt_gatt_discover_params* struct which describes the type of discovery. The parameters also serves as a filter when setting the uuid field only attributes which matches will be discovered, in contrast setting it to NULL allows all attributes to be discovered.

Note

Caching discovered attributes is not supported.

Read procedures are supported by *bt_gatt_read()* API which takes the *bt_gatt_read_params* struct as parameters. In the parameters one or more attributes can be set, though setting multiple handles requires the option: CONFIG_BT_GATT_READ_MULTIPLE

Write procedures are supported by *bt_gatt_write()* API and takes *bt_gatt_write_params* struct as parameters. In case the write operation don't require a response *bt_gatt_write_without_response()* or *bt_gatt_write_without_response_cb()* APIs can be used, with the later working similarly to *bt_gatt_notify_cb()*.

Subscriptions to notification and indication can be initiated with use of *bt_gatt_subscribe()* API which takes *bt_gatt_subscribe_params* as parameters. Multiple subscriptions to the same attribute are supported so there could be multiple notify callback being triggered for the same attribute. Subscriptions can be removed with use of *bt_gatt_unsubscribe()* API.

Note

When subscriptions are removed notify callback is called with the data set to NULL.

Related code samples

BLE logging backend
Send log messages over BLE using the BLE logging backend.

API Reference

group **bt_gatt**

Generic Attribute Profile (GATT)

Defines

BT_GATT_ERR(_att_err)

Construct error return value for attribute read and write callbacks.

Parameters

- **_att_err** – ATT error code

Returns
Appropriate error code for the attribute callbacks.

`BT_GATT_CHRC_BROADCAST`

Characteristic Properties Bit field values.

Characteristic broadcast property.

If set, permits broadcasts of the Characteristic Value using Server Characteristic Configuration Descriptor.

`BT_GATT_CHRC_READ`

Characteristic read property.

If set, permits reads of the Characteristic Value.

`BT_GATT_CHRC_WRITE_WITHOUT_RESP`

Characteristic write without response property.

If set, permits write of the Characteristic Value without response.

`BT_GATT_CHRC_WRITE`

Characteristic write with response property.

If set, permits write of the Characteristic Value with response.

`BT_GATT_CHRC_NOTIFY`

Characteristic notify property.

If set, permits notifications of a Characteristic Value without acknowledgment.

`BT_GATT_CHRC_INDICATE`

Characteristic indicate property.

If set, permits indications of a Characteristic Value with acknowledgment.

`BT_GATT_CHRC_AUTH`

Characteristic Authenticated Signed Writes property.

If set, permits signed writes to the Characteristic Value.

`BT_GATT_CHRC_EXT_PROP`

Characteristic Extended Properties property.

If set, additional characteristic properties are defined in the Characteristic Extended Properties Descriptor.

`BT_GATT_CEP_RELIABLE_WRITE`

Characteristic Extended Properties Bit field values.

`BT_GATT_CEP_WRITABLE_AUX`

`BT_GATT_CCC_NOTIFY`

Client Characteristic Configuration Values.

Client Characteristic Configuration Notification.

If set, changes to Characteristic Value shall be notified.

BT_GATT_CCC_INDICATE

Client Characteristic Configuration Indication.

If set, changes to Characteristic Value shall be indicated.

BT_GATT_SCC_BROADCAST

Server Characteristic Configuration Values.

Server Characteristic Configuration Broadcast

If set, the characteristic value shall be broadcast in the advertising data when the server is advertising.

Typedefs

typedef ssize_t (***bt_gatt_attr_read_func_t**)(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, void *buf, uint16_t len, uint16_t offset)

Attribute read callback.

The callback can also be used locally to read the contents of the attribute in which case no connection will be set.

Param conn
The connection that is requesting to read

Param attr
The attribute that's being read

Param buf
Buffer to place the read result in

Param len
Length of data to read

Param offset
Offset to start reading from

Return
Number of bytes read, or in case of an error *BT_GATT_ERR()* with a specific BT_ATT_ERR_* error code.

typedef ssize_t (***bt_gatt_attr_write_func_t**)(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, const void *buf, uint16_t len, uint16_t offset, uint8_t flags)

Attribute write callback.

Param conn
The connection that is requesting to write

Param attr
The attribute that's being written

Param buf
Buffer with the data to write

Param len
Number of bytes in the buffer

Param offset
Offset to start writing from

Param flags
Flags (BT_GATT_WRITE_FLAG_*)

Return
Number of bytes written, or in case of an error *BT_GATT_ERR()* with a specific BT_ATT_ERR_* error code.

Enums

enum **bt_gatt_perm**

GATT attribute permission bit field values.

Values:

enumerator **BT_GATT_PERM_NONE** = 0

No operations supported, e.g.

for notify-only

enumerator **BT_GATT_PERM_READ** = *BIT*(0)

Attribute read permission.

enumerator **BT_GATT_PERM_WRITE** = *BIT*(1)

Attribute write permission.

enumerator **BT_GATT_PERM_READ_ENCRYPT** = *BIT*(2)

Attribute read permission with encryption.

```
If set, requires encryption for read access.
```

enumerator **BT_GATT_PERM_WRITE_ENCRYPT** = *BIT*(3)

Attribute write permission with encryption.

```
If set, requires encryption for write access.
```

enumerator **BT_GATT_PERM_READ_AUTHEN** = *BIT*(4)

Attribute read permission with authentication.

```
If set, requires encryption using authenticated link-key for read
access.
```

enumerator **BT_GATT_PERM_WRITE_AUTHEN** = *BIT*(5)

Attribute write permission with authentication.

```
If set, requires encryption using authenticated link-key for write
access.
```

enumerator **BT_GATT_PERM_PREPARE_WRITE** = *BIT*(6)

Attribute prepare write permission.

```
If set, allows prepare writes with use of ``BT_GATT_WRITE_FLAG_PREPARE``
passed to write callback.
```

enumerator **BT_GATT_PERM_READ_LESC** = *BIT*(7)

Attribute read permission with LE Secure Connection encryption.

```
If set, requires that LE Secure Connections is used for read access.
```

enumerator **BT_GATT_PERM_WRITE_LESC** = *BIT*(8)

Attribute write permission with LE Secure Connection encryption.

```
If set, requires that LE Secure Connections is used for write access.
```

GATT attribute write flags.

Values:

enumerator **BT_GATT_WRITE_FLAG_PREPARE** = *BIT*(0)

Attribute prepare write flag.

```
If set, write callback should only check if the device is
authorized but no data shall be written.
```

enumerator **BT_GATT_WRITE_FLAG_CMD** = *BIT*(1)

Attribute write command flag.

```
If set, indicates that write operation is a command (Write without
response) which doesn't generate any response.
```

enumerator **BT_GATT_WRITE_FLAG_EXECUTE** = *BIT*(2)

Attribute write execute flag.

```
If set, indicates that write operation is a execute, which indicates
the end of a long write, and will come after 1 or more
@ref BT_GATT_WRITE_FLAG_PREPARE.
```

struct **bt_gatt_attr**

#include <gatt.h> GATT Attribute structure.

Public Members

const struct *bt_uuid* ***uuid**

Attribute UUID.

bt_gatt_attr_read_func_t **read**

Attribute read callback.

bt_gatt_attr_write_func_t **write**

Attribute write callback.

void ***user_data**

Attribute user data.

uint16_t **handle**
Attribute handle.

uint16_t **perm**
Attribute permissions.
Will be 0 if returned from *bt_gatt_discover()*.

struct **bt_gatt_service_static**
#include <*gatt.h*> GATT Service structure.

Public Members

const struct *bt_gatt_attr* ***attrs**
Service Attributes.

size_t **attr_count**
Service Attribute count.

struct **bt_gatt_service**
#include <*gatt.h*> GATT Service structure.

Public Members

struct *bt_gatt_attr* ***attrs**
Service Attributes.

size_t **attr_count**
Service Attribute count.

struct **bt_gatt_service_val**
#include <*gatt.h*> Service Attribute Value.

Public Members

const struct *bt_uuid* ***uuid**
Service UUID.

uint16_t **end_handle**
Service end handle.

struct **bt_gatt_include**
#include <*gatt.h*> Include Attribute Value.

Public Members

const struct *bt_uuid* ***uuid**

Service UUID.

uint16_t **start_handle**

Service start handle.

uint16_t **end_handle**

Service end handle.

struct **bt_gatt_cb**

#include <*gatt.h*> GATT callback structure.

Public Members

void (***att_mtu_updated**)(struct bt_conn *conn, uint16_t tx, uint16_t rx)

The maximum ATT MTU on a connection has changed.

This callback notifies the application that the maximum TX or RX ATT MTU has increased.

Param conn
Connection object.

Param tx
Updated TX ATT MTU.

Param rx
Updated RX ATT MTU.

struct **bt_gatt_authorization_cb**

#include <*gatt.h*> GATT authorization callback structure.

Public Members

bool (***read_authorize**)(struct bt_conn *conn, const struct *bt_gatt_attr* *attr)

Authorize the GATT read operation.

This callback allows the application to authorize the GATT read operation for the attribute that is being read.

Param conn
Connection object.

Param attr
The attribute that is being read.

Retval true
Authorize the operation and allow it to execute.

Retval false
Reject the operation and prevent it from executing.

bool (***write_authorize**)(struct bt_conn *conn, const struct *bt_gatt_attr* *attr)

Authorize the GATT write operation.

This callback allows the application to authorize the GATT write operation for the attribute that is being written.

Param conn
Connection object.
Param attr
The attribute that is being written.
Retval true
Authorize the operation and allow it to execute.
Retval false
Reject the operation and prevent it from executing.

struct `bt_gatt_chrc`

#include <*gatt.h*> Characteristic Attribute Value.

Public Members

const struct *bt_uuid* *`uuid`

Characteristic UUID.

uint16_t `value_handle`

Characteristic Value handle.

uint8_t `properties`

Characteristic properties.

struct `bt_gatt_cep`

#include <*gatt.h*> Characteristic Extended Properties Attribute Value.

Public Members

uint16_t `properties`

Characteristic Extended properties.

struct `bt_gatt_ccc`

#include <*gatt.h*> Client Characteristic Configuration Attribute Value.

Public Members

uint16_t `flags`

Client Characteristic Configuration flags.

struct `bt_gatt_scc`

#include <*gatt.h*> Server Characteristic Configuration Attribute Value.

Public Members

uint16_t `flags`

Server Characteristic Configuration flags.

struct **bt_gatt_cpf**

#include <*gatt.h*> GATT Characteristic Presentation Format Attribute Value.

Public Members

uint8_t **format**

Format of the value of the characteristic.

int8_t **exponent**

Exponent field to determine how the value of this characteristic is further formatted.

uint16_t **unit**

Unit of the characteristic.

uint8_t **name_space**

Name space of the description.

uint16_t **description**

Description of the characteristic as defined in a higher layer profile.

GATT Server

group **bt_gatt_server**

Defines

BT_GATT_SERVICE_DEFINE(_name, ...)

Statically define and register a service.

Helper macro to statically define and register a service.

Parameters

- **_name** – Service name.

BT_GATT_SERVICE_INSTANCE_DEFINE(_name, _instances, _instance_num, _attrs_def)

Statically define service structure array.

Helper macro to statically define service structure array. Each element of the array is linked to the service attribute array which is also defined in this scope using `_attrs_def` macro.

Parameters

- **_name** – Name of service structure array.
- **_instances** – Array of instances to pass as user context to the attribute callbacks.
- **_instance_num** – Number of elements in instance array.
- **_attrs_def** – Macro provided by the user that defines attribute array for the service. This macro should accept single parameter which is the instance context.

BT_GATT_SERVICE(_attrs)

Service Structure Declaration Macro.

Helper macro to declare a service structure.

Parameters

- **_attrs** – Service attributes.

BT_GATT_PRIMARY_SERVICE(_service)

Primary Service Declaration Macro.

Helper macro to declare a primary service attribute.

Parameters

- **_service** – Service attribute value.

BT_GATT_SECONDARY_SERVICE(_service)

Secondary Service Declaration Macro.

Helper macro to declare a secondary service attribute.

> **Note**
>
> A secondary service is only intended to be included from a primary service or another secondary service or other higher layer specification.

Parameters

- **_service** – Service attribute value.

BT_GATT_INCLUDE_SERVICE(_service_incl)

Include Service Declaration Macro.

Helper macro to declare database internal include service attribute.

Parameters

- **_service_incl** – the first service attribute of service to include

BT_GATT_CHRC_INIT(_uuid, _handle, _props)

BT_GATT_CHARACTERISTIC(_uuid, _props, _perm, _read, _write, _user_data)

Characteristic and Value Declaration Macro.

Helper macro to declare a characteristic attribute along with its attribute value.

Parameters

- **_uuid** – Characteristic attribute uuid.
- **_props** – Characteristic attribute properties, a bitmap of BT_GATT_CHRC_* macros.
- **_perm** – Characteristic Attribute access permissions, a bitmap of *bt_gatt_perm* values.
- **_read** – Characteristic Attribute read callback (*bt_gatt_attr_read_func_t*).
- **_write** – Characteristic Attribute write callback (*bt_gatt_attr_write_func_t*).
- **_user_data** – Characteristic Attribute user data.

BT_GATT_CCC_MAX

BT_GATT_CCC_INITIALIZER(_changed, _write, _match)

Initialize Client Characteristic Configuration Declaration Macro.

Helper macro to initialize a Managed CCC attribute value.

Parameters

- **_changed** – Configuration changed callback.
- **_write** – Configuration write callback.
- **_match** – Configuration match callback.

BT_GATT_CCC_MANAGED(_ccc, _perm)

Managed Client Characteristic Configuration Declaration Macro.

Helper macro to declare a Managed CCC attribute.

Parameters

- **_ccc** – CCC attribute user data, shall point to a _bt_gatt_ccc.
- **_perm** – CCC access permissions, a bitmap of *bt_gatt_perm* values.

BT_GATT_CCC(_changed, _perm)

Client Characteristic Configuration Declaration Macro.

Helper macro to declare a CCC attribute.

Parameters

- **_changed** – Configuration changed callback.
- **_perm** – CCC access permissions, a bitmap of *bt_gatt_perm* values.

BT_GATT_CEP(_value)

Characteristic Extended Properties Declaration Macro.

Helper macro to declare a CEP attribute.

Parameters

- **_value** – Pointer to a struct *bt_gatt_cep*.

BT_GATT_CUD(_value, _perm)

Characteristic User Format Descriptor Declaration Macro.

Helper macro to declare a CUD attribute.

Parameters

- **_value** – User description NULL-terminated C string.
- **_perm** – Descriptor attribute access permissions, a bitmap of *bt_gatt_perm* values.

BT_GATT_CPF(_value)

Characteristic Presentation Format Descriptor Declaration Macro.

Helper macro to declare a CPF attribute.

Parameters

- **_value** – Pointer to a struct *bt_gatt_cpf*.

BT_GATT_DESCRIPTOR(_uuid, _perm, _read, _write, _user_data)

Descriptor Declaration Macro.

Helper macro to declare a descriptor attribute.

Parameters

- **_uuid** – Descriptor attribute uuid.
- **_perm** – Descriptor attribute access permissions, a bitmap of *bt_gatt_perm* values.
- **_read** – Descriptor attribute read callback (*bt_gatt_attr_read_func_t*).
- **_write** – Descriptor attribute write callback (*bt_gatt_attr_write_func_t*).
- **_user_data** – Descriptor attribute user data.

BT_GATT_ATTRIBUTE(_uuid, _perm, _read, _write, _user_data)

Attribute Declaration Macro.

Helper macro to declare an attribute.

Parameters

- **_uuid** – Attribute uuid.
- **_perm** – Attribute access permissions, a bitmap of *bt_gatt_perm* values.
- **_read** – Attribute read callback (*bt_gatt_attr_read_func_t*).
- **_write** – Attribute write callback (*bt_gatt_attr_write_func_t*).
- **_user_data** – Attribute user data.

Typedefs

typedef uint8_t (***bt_gatt_attr_func_t**)(const struct *bt_gatt_attr* *attr, uint16_t handle, void *user_data)

Attribute iterator callback.

Param attr
Attribute found.

Param handle
Attribute handle found.

Param user_data
Data given.

Return
BT_GATT_ITER_CONTINUE if should continue to the next attribute.

Return
BT_GATT_ITER_STOP to stop.

typedef void (***bt_gatt_complete_func_t**)(struct bt_conn *conn, void *user_data)

Notification complete result callback.

Param conn
Connection object.

Param user_data
Data passed in by the user.

typedef void (***bt_gatt_indicate_func_t**)(struct bt_conn *conn, struct *bt_gatt_indicate_params* *params, uint8_t err)

Indication complete result callback.

Param conn
Connection object.

Param params
Indication params object.

Param err
ATT error code

typedef void (*bt_gatt_indicate_params_destroy_t)(struct *bt_gatt_indicate_params* *params)

Enums

Values:

enumerator BT_GATT_ITER_STOP = 0

enumerator BT_GATT_ITER_CONTINUE

Functions

static inline const char *bt_gatt_err_to_str(int gatt_err)
Converts a GATT error to string.

The GATT errors are created with *BT_GATT_ERR*.

The error codes are described in the Bluetooth Core specification, Vol 3, Part F, Section 3.4.1.1.

The ATT and GATT documentation found in Vol 4, Part F and Part G describe when the different error codes are used.

See also the defined BT_ATT_ERR_* macros.

Returns
The string representation of the GATT error code. If CONFIG_BT_ATT_ERR_TO_STR is not enabled, this just returns the empty string.

void bt_gatt_cb_register(struct *bt_gatt_cb* *cb)
Register GATT callbacks.

Register callbacks to monitor the state of GATT. The callback struct must remain valid for the remainder of the program.

Parameters

- **cb** – Callback struct.

int bt_gatt_authorization_cb_register(const struct *bt_gatt_authorization_cb* *cb)
Register GATT authorization callbacks.

Register callbacks to perform application-specific authorization of GATT operations on all registered GATT attributes. The callback structure must remain valid throughout the entire duration of the Bluetooth subsys activity.

The CONFIG_BT_GATT_AUTHORIZATION_CUSTOM Kconfig must be enabled to make this API functional.

This API allows the user to register only one callback structure concurrently. Passing NULL unregisters the previous set of callbacks and makes it possible to register a new one.

Parameters

- **cb** – Callback struct.

Returns
Zero on success or negative error code otherwise

int **bt_gatt_service_register**(struct *bt_gatt_service* *svc)

Register GATT service.

Register GATT service. Applications can make use of macros such as BT_GATT_PRIMARY_SERVICE, BT_GATT_CHARACTERISTIC, BT_GATT_DESCRIPTOR, etc.

When using CONFIG_BT_SETTINGS then all services that should have bond configuration loaded, i.e. CCC values, must be registered before calling *settings_load*.

When using CONFIG_BT_GATT_CACHING and CONFIG_BT_SETTINGS then all services that should be included in the GATT Database Hash calculation should be added before calling *settings_load*. All services registered after settings_load will trigger a new database hash calculation and a new hash stored.

There are two situations where this function can be called: either before bt_init() has been called, or after *settings_load()* has been called. Registering a service in the middle is not supported and will return an error.

Parameters

- **svc** – Service containing the available attributes

Returns
0 in case of success or negative value in case of error.

Returns
-EAGAIN if bt_init() has been called but *settings_load()* hasn't yet.

int **bt_gatt_service_unregister**(struct *bt_gatt_service* *svc)

Unregister GATT service.

Parameters

- **svc** – Service to be unregistered.

Returns
0 in case of success or negative value in case of error.

bool **bt_gatt_service_is_registered**(const struct *bt_gatt_service* *svc)

Check if GATT service is registered.

Parameters

- **svc** – Service to be checked.

Returns
true if registered or false if not register.

void **bt_gatt_foreach_attr_type**(uint16_t start_handle, uint16_t end_handle, const struct *bt_uuid* *uuid, const void *attr_data, uint16_t num_matches, *bt_gatt_attr_func_t* func, void *user_data)

Attribute iterator by type.

Iterate attributes in the given range matching given UUID and/or data.

Parameters

- **start_handle** – Start handle.
- **end_handle** – End handle.
- **uuid** – UUID to match, passing NULL skips UUID matching.

- **attr_data** – Attribute data to match, passing NULL skips data matching.
- **num_matches** – Number matches, passing 0 makes it unlimited.
- **func** – Callback function.
- **user_data** – Data to pass to the callback.

static inline void **bt_gatt_foreach_attr**(uint16_t start_handle, uint16_t end_handle, *bt_gatt_attr_func_t* func, void *user_data)

Attribute iterator.

Iterate attributes in the given range.

Parameters

- **start_handle** – Start handle.
- **end_handle** – End handle.
- **func** – Callback function.
- **user_data** – Data to pass to the callback.

struct *bt_gatt_attr* ***bt_gatt_attr_next**(const struct *bt_gatt_attr* *attr)

Iterate to the next attribute.

Iterate to the next attribute following a given attribute.

Parameters

- **attr** – Current Attribute.

Returns

The next attribute or NULL if it cannot be found.

struct *bt_gatt_attr* ***bt_gatt_find_by_uuid**(const struct *bt_gatt_attr* *attr, uint16_t attr_count, const struct *bt_uuid* *uuid)

Find Attribute by UUID.

Find the attribute with the matching UUID. To limit the search to a service set the attr to the service attributes and the `attr_count` to the service attribute count .

Parameters

- **attr** – Pointer to an attribute that serves as the starting point for the search of a match for the UUID. Passing NULL will search the entire range.
- **attr_count** – The number of attributes from the starting point to search for a match for the UUID. Set to 0 to search until the end.
- **uuid** – UUID to match.

uint16_t **bt_gatt_attr_get_handle**(const struct *bt_gatt_attr* *attr)

Get Attribute handle.

Parameters

- **attr** – Attribute object.

Returns

Handle of the corresponding attribute or zero if the attribute could not be found.

uint16_t **bt_gatt_attr_value_handle**(const struct *bt_gatt_attr* *attr)

Get the handle of the characteristic value descriptor.

Note

The `user_data` of the attribute must of type *bt_gatt_chrc*.

Parameters

- **attr** – A Characteristic Attribute.

Returns
the handle of the corresponding Characteristic Value. The value will be zero (the invalid handle) if `attr` was not a characteristic attribute.

ssize_t **bt_gatt_attr_read**(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, void *buf, uint16_t buf_len, uint16_t offset, const void *value, uint16_t value_len)

Generic Read Attribute value helper.

Read attribute value from local database storing the result into buffer.

Parameters

- **conn** – Connection object.
- **attr** – Attribute to read.
- **buf** – Buffer to store the value.
- **buf_len** – Buffer length.
- **offset** – Start offset.
- **value** – Attribute value.
- **value_len** – Length of the attribute value.

Returns
number of bytes read in case of success or negative values in case of error.

ssize_t **bt_gatt_attr_read_service**(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, void *buf, uint16_t len, uint16_t offset)

Read Service Attribute helper.

Read service attribute value from local database storing the result into buffer after encoding it.

Note

Only use this with attributes which `user_data` is a *bt_uuid*.

Parameters

- **conn** – Connection object.
- **attr** – Attribute to read.
- **buf** – Buffer to store the value read.
- **len** – Buffer length.
- **offset** – Start offset.

Returns
number of bytes read in case of success or negative values in case of error.

ssize_t **bt_gatt_attr_read_included**(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, void *buf, uint16_t len, uint16_t offset)

Read Include Attribute helper.

Read include service attribute value from local database storing the result into buffer after encoding it.

> **Note**
>
> Only use this with attributes which user_data is a *bt_gatt_include*.

Parameters

- **conn** – Connection object.
- **attr** – Attribute to read.
- **buf** – Buffer to store the value read.
- **len** – Buffer length.
- **offset** – Start offset.

Returns

number of bytes read in case of success or negative values in case of error.

ssize_t **bt_gatt_attr_read_chrc**(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, void *buf, uint16_t len, uint16_t offset)

Read Characteristic Attribute helper.

Read characteristic attribute value from local database storing the result into buffer after encoding it.

> **Note**
>
> Only use this with attributes which `user_data` is a *bt_gatt_chrc*.

Parameters

- **conn** – Connection object.
- **attr** – Attribute to read.
- **buf** – Buffer to store the value read.
- **len** – Buffer length.
- **offset** – Start offset.

Returns

number of bytes read in case of success or negative values in case of error.

ssize_t **bt_gatt_attr_read_ccc**(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, void *buf, uint16_t len, uint16_t offset)

Read Client Characteristic Configuration Attribute helper.

Read CCC attribute value from local database storing the result into buffer after encoding it.

> **Note**
>
> Only use this with attributes which user_data is a _bt_gatt_ccc.

Parameters

- `conn` – Connection object.
- `attr` – Attribute to read.
- `buf` – Buffer to store the value read.
- `len` – Buffer length.
- `offset` – Start offset.

Returns
number of bytes read in case of success or negative values in case of error.

ssize_t **bt_gatt_attr_write_ccc**(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, const void *buf, uint16_t len, uint16_t offset, uint8_t flags)

Write Client Characteristic Configuration Attribute helper.

Write value in the buffer into CCC attribute.

> **Note**
>
> Only use this with attributes which user_data is a _bt_gatt_ccc.

Parameters

- `conn` – Connection object.
- `attr` – Attribute to read.
- `buf` – Buffer to store the value read.
- `len` – Buffer length.
- `offset` – Start offset.
- `flags` – Write flags.

Returns
number of bytes written in case of success or negative values in case of error.

ssize_t **bt_gatt_attr_read_cep**(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, void *buf, uint16_t len, uint16_t offset)

Read Characteristic Extended Properties Attribute helper.

Read CEP attribute value from local database storing the result into buffer after encoding it.

> **Note**
>
> Only use this with attributes which user_data is a *bt_gatt_cep*.

Parameters

- `conn` – Connection object

- **attr** – Attribute to read
- **buf** – Buffer to store the value read
- **len** – Buffer length
- **offset** – Start offset

Returns
number of bytes read in case of success or negative values in case of error.

ssize_t **bt_gatt_attr_read_cud**(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, void *buf, uint16_t len, uint16_t offset)

Read Characteristic User Description Descriptor Attribute helper.

Read CUD attribute value from local database storing the result into buffer after encoding it.

Note

Only use this with attributes which user_data is a NULL-terminated C string.

Parameters

- **conn** – Connection object
- **attr** – Attribute to read
- **buf** – Buffer to store the value read
- **len** – Buffer length
- **offset** – Start offset

Returns
number of bytes read in case of success or negative values in case of error.

ssize_t **bt_gatt_attr_read_cpf**(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, void *buf, uint16_t len, uint16_t offset)

Read Characteristic Presentation format Descriptor Attribute helper.

Read CPF attribute value from local database storing the result into buffer after encoding it.

Note

Only use this with attributes which user_data is a bt_gatt_pf.

Parameters

- **conn** – Connection object
- **attr** – Attribute to read
- **buf** – Buffer to store the value read
- **len** – Buffer length
- **offset** – Start offset

Returns
number of bytes read in case of success or negative values in case of error.

int **bt_gatt_notify_cb**(struct bt_conn *conn, struct *bt_gatt_notify_params* *params)

Notify attribute value change.

This function works in the same way as *bt_gatt_notify*. With the addition that after sending the notification the callback function will be called.

The callback is run from System Workqueue context. When called from the System Workqueue context this API will not wait for resources for the callback but instead return an error. The number of pending callbacks can be increased with the `CONFIG_BT_CONN_TX_MAX` option.

Alternatively it is possible to notify by UUID by setting it on the parameters, when using this method the attribute if provided is used as the start range when looking up for possible matches.

Parameters

- **conn** – Connection object.
- **params** – Notification parameters.

Returns

0 in case of success or negative value in case of error.

int **bt_gatt_notify_multiple**(struct bt_conn *conn, uint16_t num_params, struct *bt_gatt_notify_params* params[])

Send multiple notifications in a single PDU.

The GATT Server will send a single ATT_MULTIPLE_HANDLE_VALUE_NTF PDU containing all the notifications passed to this API.

All `params` must have the same `func` and `user_data` (due to implementation limitation). But `func(user_data)` will be invoked for each parameter.

As this API may block to wait for Bluetooth Host resources, it is not recommended to call it from a cooperative thread or a Bluetooth callback.

The peer's GATT Client must write to this device's Client Supported Features attribute and set the bit for Multiple Handle Value Notifications before this API can be used.

Only use this API to force the use of the ATT_MULTIPLE_HANDLE_VALUE_NTF PDU. For standard applications, `bt_gatt_notify_cb` is preferred, as it will use this PDU if supported and automatically fallback to ATT_HANDLE_VALUE_NTF when not supported by the peer.

This API has an additional limitation: it only accepts valid attribute references and not UUIDs like `bt_gatt_notify` and `bt_gatt_notify_cb`.

Parameters

- **conn** – Target client. Notifying all connected clients by passing `NULL` is not yet supported, please use `bt_gatt_notify` instead.
- **num_params** – Element count of `params` array. Has to be greater than 1.
- **params** – Array of notification parameters. It is okay to free this after calling this function.

Return values

- **0** – Success. The PDU is queued for sending.
- **-EINVAL** –
  - One of the attribute handles is invalid.
  - Only one parameter was passed. This API expects 2 or more.
  - Not all `func` were equal or not all `user_data` were equal.

  - One of the characteristics is not notifiable.
  - An UUID was passed in one of the parameters.
- `-ERANGE` –
  - The notifications cannot all fit in a single ATT_MULTIPLE_HANDLE_VALUE_NTF.
  - They exceed the MTU of all open ATT bearers.
- `-EPERM` – The connection has a lower security level than required by one of the attributes.
- `-EOPNOTSUPP` – The peer hasn't yet communicated that it supports this PDU type.

static inline int **bt_gatt_notify**(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, const void *data, uint16_t len)

Notify attribute value change.

Send notification of attribute value change, if connection is NULL notify all peer that have notification enabled via CCC otherwise do a direct notification only the given connection.

The attribute object on the parameters can be the so called Characteristic Declaration, which is usually declared with BT_GATT_CHARACTERISTIC followed by BT_GATT_CCC, or the Characteristic Value Declaration which is automatically created after the Characteristic Declaration when using BT_GATT_CHARACTERISTIC.

Parameters

- `conn` – Connection object.
- `attr` – Characteristic or Characteristic Value attribute.
- `data` – Pointer to Attribute data.
- `len` – Attribute value length.

Returns

0 in case of success or negative value in case of error.

static inline int **bt_gatt_notify_uuid**(struct bt_conn *conn, const struct *bt_uuid* *uuid, const struct *bt_gatt_attr* *attr, const void *data, uint16_t len)

Notify attribute value change by UUID.

Send notification of attribute value change, if connection is NULL notify all peer that have notification enabled via CCC otherwise do a direct notification only on the given connection.

The attribute object is the starting point for the search of the UUID.

Parameters

- `conn` – Connection object.
- `uuid` – The UUID. If the server contains multiple services with the same UUID, then the first occurrence, starting from the attr given, is used.
- `attr` – Pointer to an attribute that serves as the starting point for the search of a match for the UUID.
- `data` – Pointer to Attribute data.
- `len` – Attribute value length.

Returns

0 in case of success or negative value in case of error.

int **bt_gatt_indicate**(struct bt_conn *conn, struct *bt_gatt_indicate_params* *params)

Indicate attribute value change.

Send an indication of attribute value change. if connection is NULL indicate all peer that have notification enabled via CCC otherwise do a direct indication only the given connection.

The attribute object on the parameters can be the so called Characteristic Declaration, which is usually declared with BT_GATT_CHARACTERISTIC followed by BT_GATT_CCC, or the Characteristic Value Declaration which is automatically created after the Characteristic Declaration when using BT_GATT_CHARACTERISTIC.

Alternatively it is possible to indicate by UUID by setting it on the parameters, when using this method the attribute if provided is used as the start range when looking up for possible matches.

Note

This procedure is asynchronous therefore the parameters need to remains valid while it is active. The procedure is active until the destroy callback is run.

Parameters

- **conn** – Connection object.
- **params** – Indicate parameters.

Returns

0 in case of success or negative value in case of error.

bool **bt_gatt_is_subscribed**(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, uint16_t ccc_type)

Check if connection have subscribed to attribute.

Check if connection has subscribed to attribute value change.

The attribute object can be the so called Characteristic Declaration, which is usually declared with BT_GATT_CHARACTERISTIC followed by BT_GATT_CCC, or the Characteristic Value Declaration which is automatically created after the Characteristic Declaration when using BT_GATT_CHARACTERISTIC, or the Client Characteristic Configuration Descriptor (CCCD) which is created by BT_GATT_CCC.

Parameters

- **conn** – Connection object.
- **attr** – Attribute object.
- **ccc_type** – The subscription type, *BT_GATT_CCC_NOTIFY* and/or *BT_GATT_CCC_INDICATE*.

Returns

true if the attribute object has been subscribed.

uint16_t **bt_gatt_get_mtu**(struct bt_conn *conn)

Get ATT MTU for a connection.

Get negotiated ATT connection MTU, note that this does not equal the largest amount of attribute data that can be transferred within a single packet.

Parameters

- **conn** – Connection object.

Returns
MTU in bytes

struct **bt_gatt_ccc_cfg**

#include <*gatt.h*> GATT CCC configuration entry.

Public Members

uint8_t **id**

Local identity, BT_ID_DEFAULT in most cases.

bt_addr_le_t **peer**

Remote peer address.

uint16_t **value**

Configuration value.

struct **bt_gatt_notify_params**

#include <*gatt.h*>

Public Members

const struct *bt_uuid* ***uuid**

Notification Attribute UUID type.

Optional, use to search for an attribute with matching UUID when the attribute object pointer is not known.

const struct *bt_gatt_attr* ***attr**

Notification Attribute object.

Optional if uuid is provided, in this case it will be used as start range to search for the attribute with the given UUID.

const void ***data**

Notification Value data.

uint16_t **len**

Notification Value length.

bt_gatt_complete_func_t **func**

Notification Value callback.

void ***user_data**

Notification Value callback user data.

struct **bt_gatt_indicate_params**

#include <*gatt.h*> GATT Indicate Value parameters.

Public Members

const struct *bt_uuid* ***uuid**

Indicate Attribute UUID type.

Optional, use to search for an attribute with matching UUID when the attribute object pointer is not known.

const struct *bt_gatt_attr* ***attr**

Indicate Attribute object.

Optional if uuid is provided, in this case it will be used as start range to search for the attribute with the given UUID.

bt_gatt_indicate_func_t **func**

Indicate Value callback.

bt_gatt_indicate_params_destroy_t **destroy**

Indicate operation complete callback.

const void ***data**

Indicate Value data.

uint16_t **len**

Indicate Value length.

GATT Client

group **bt_gatt_client**

Typedefs

typedef uint8_t (***bt_gatt_discover_func_t**)(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, struct *bt_gatt_discover_params* *params)

Discover attribute callback function.

If discovery procedure has completed this callback will be called with attr set to NULL. This will not happen if procedure was stopped by returning BT_GATT_ITER_STOP.

The attribute object as well as its UUID and value objects are temporary and must be copied to in order to cache its information. Only the following fields of the attribute contains valid information:

- uuid UUID representing the type of attribute.
- handle Handle in the remote database.
- user_data The value of the attribute, if the discovery type maps to an ATT operation that provides this information. NULL otherwise. See below.

The effective type of `attr->user_data` is determined by `params`. Note that the fields `params->type` and `params->uuid` are left unchanged by the discovery procedure.

params->type	params->uuid	Type of attr->user_data
BT_GATT_DISCOVER_PRIMARY	any	*bt_gatt_service_val*
BT_GATT_DISCOVER_SECONDARY	any	*bt_gatt_service_val*
BT_GATT_DISCOVER_INCLUDE	any	*bt_gatt_include*
BT_GATT_DISCOVER_CHARACTERIST	any	*bt_gatt_chrc*
BT_GATT_DISCOVER_STD_CHAR_DES	*BT_UUID_GATT_CEP*	*bt_gatt_cep*
BT_GATT_DISCOVER_STD_CHAR_DES	*BT_UUID_GATT_CCC*	*bt_gatt_ccc*
BT_GATT_DISCOVER_STD_CHAR_DES	*BT_UUID_GATT_SCC*	*bt_gatt_scc*
BT_GATT_DISCOVER_STD_CHAR_DES	*BT_UUID_GATT_CPF*	*bt_gatt_cpf*
BT_GATT_DISCOVER_DESCRIPTOR	any	NULL
BT_GATT_DISCOVER_ATTRIBUTE	any	NULL

Also consider if using read-by-type instead of discovery is more convenient. See *bt_gatt_read* with *bt_gatt_read_params::handle_count* set to `0`.

Param conn
Connection object.

Param attr
Attribute found, or NULL if not found.

Param params
Discovery parameters given.

Return
BT_GATT_ITER_CONTINUE to continue discovery procedure.

Return
BT_GATT_ITER_STOP to stop discovery procedure.

typedef uint8_t (***bt_gatt_read_func_t***)(struct bt_conn *conn, uint8_t err, struct *bt_gatt_read_params* *params, const void *data, uint16_t length)

Read callback function.

When reading using by_uuid, `params->start_handle` is the attribute handle for this `data` item.

Param conn
Connection object.

Param err
ATT error code.

Param params
Read parameters used.

Param data
Attribute value data. NULL means read has completed.

Param length
Attribute value length.

Return
BT_GATT_ITER_CONTINUE if should continue to the next attribute.

Return
BT_GATT_ITER_STOP to stop.

typedef void (***bt_gatt_write_func_t***)(struct bt_conn *conn, uint8_t err, struct *bt_gatt_write_params* *params)

Write callback function.

Param conn
Connection object.

Param err
ATT error code.

Param params
Write parameters used.

typedef uint8_t (***bt_gatt_notify_func_t**)(struct bt_conn *conn, struct *bt_gatt_subscribe_params* *params, const void *data, uint16_t length)

Notification callback function.

In the case of an empty notification, the `data` pointer will be non-NULL while the `length` will be 0, which is due to the special case where a `data` NULL pointer means unsubscribed.

Param conn
Connection object. May be NULL, indicating that the peer is being unpaired

Param params
Subscription parameters.

Param data
Attribute value data. If NULL then subscription was removed.

Param length
Attribute value length.

Return
BT_GATT_ITER_CONTINUE to continue receiving value notifications. BT_GATT_ITER_STOP to unsubscribe from value notifications.

typedef void (***bt_gatt_subscribe_func_t**)(struct bt_conn *conn, uint8_t err, struct *bt_gatt_subscribe_params* *params)

Subscription callback function.

Param conn
Connection object.

Param err
ATT error code.

Param params
Subscription parameters used.

Enums

GATT Discover types.

Values:

enumerator `BT_GATT_DISCOVER_PRIMARY`

Discover Primary Services.

enumerator `BT_GATT_DISCOVER_SECONDARY`

Discover Secondary Services.

enumerator BT_GATT_DISCOVER_INCLUDE

Discover Included Services.

enumerator BT_GATT_DISCOVER_CHARACTERISTIC

Discover Characteristic Values.

```
Discover Characteristic Value and its properties.
```

enumerator BT_GATT_DISCOVER_DESCRIPTOR

Discover Descriptors.

```
Discover Attributes which are not services or characteristics.

@note The use of this type of discover is not recommended for
      discovering in ranges across multiple services/characteristics
      as it may incur in extra round trips.
```

enumerator BT_GATT_DISCOVER_ATTRIBUTE

Discover Attributes.

```
Discover Attributes of any type.

@note The use of this type of discover is not recommended for
      discovering in ranges across multiple services/characteristics
      as it may incur in more round trips.
```

enumerator BT_GATT_DISCOVER_STD_CHAR_DESC

Discover standard characteristic descriptor values.

```
Discover standard characteristic descriptor values and their
properties.
Supported descriptors:
 - Characteristic Extended Properties
 - Client Characteristic Configuration
 - Server Characteristic Configuration
 - Characteristic Presentation Format
```

Subscription flags.

Values:

enumerator BT_GATT_SUBSCRIBE_FLAG_VOLATILE

Persistence flag.

```
If set, indicates that the subscription is not saved
on the GATT server side. Therefore, upon disconnection,
the subscription will be automatically removed
from the client's subscriptions list and
when the client reconnects, it will have to
issue a new subscription.
```

enumerator BT_GATT_SUBSCRIBE_FLAG_NO_RESUB

No resubscribe flag.

```
By default when BT_GATT_SUBSCRIBE_FLAG_VOLATILE is unset, the
subscription will be automatically renewed when the client
reconnects, as a workaround for GATT servers that do not persist
subscriptions.

This flag will disable the automatic resubscription. It is useful
if the application layer knows that the GATT server remembers
subscriptions from previous connections and wants to avoid renewing
the subscriptions.
```

enumerator **BT_GATT_SUBSCRIBE_FLAG_WRITE_PENDING**

Write pending flag.

```
If set, indicates write operation is pending waiting remote end to
respond.

@note Internal use only.
```

enumerator **BT_GATT_SUBSCRIBE_FLAG_SENT**

Sent flag.

```
If set, indicates that a subscription request (CCC write) has
already been sent in the active connection.

Used to avoid sending subscription requests multiple times when the
\htmlonly <code>CONFIG_BT_GATT_AUTO_RESUBSCRIBE</code> \endhtmlonly \
↪xmlonly <verbatim>embed:rst:inline :kconfig:option:`CONFIG_BT_GATT_AUTO_
↪RESUBSCRIBE`</verbatim> \endxmlonly quirk is enabled.

@note Internal use only.
```

enumerator **BT_GATT_SUBSCRIBE_NUM_FLAGS**

Functions

int **bt_gatt_exchange_mtu**(struct bt_conn *conn, struct *bt_gatt_exchange_params* *params)

Exchange MTU.

This client procedure can be used to set the MTU to the maximum possible size the buffers can hold.

The Response comes in callback `params->func`. The callback is run from the context specified by 'config BT_RECV_CONTEXT'. `params` must remain valid until start of callback.

This function will block while the ATT request queue is full, except when called from the BT RX thread, as this would cause a deadlock.

Note

Shall only be used once per connection.

Parameters

- **conn** – Connection object.
- **params** – Exchange MTU parameters.

Return values

- **0** – Successfully queued request. Will call `params->func` on resolution.
- **-ENOMEM** – ATT request queue is full and blocking would cause deadlock. Allow a pending request to resolve before retrying, or call this function outside the BT RX thread to get blocking behavior. Queue size is controlled by `CONFIG_BT_ATT_TX_COUNT` .
- **-EALREADY** – The MTU exchange procedure has been already performed.

int **bt_gatt_discover**(struct bt_conn *conn, struct *bt_gatt_discover_params* *params)

GATT Discover function.

This procedure is used by a client to discover attributes on a server.

Primary Service Discovery: Procedure allows to discover primary services either by Discover All Primary Services or Discover Primary Services by Service UUID. Include Service Discovery: Procedure allows to discover all Include Services within specified range. Characteristic Discovery: Procedure allows to discover all characteristics within specified handle range as well as discover characteristics with specified UUID. Descriptors Discovery: Procedure allows to discover all characteristic descriptors within specified range.

For each attribute found the callback is called which can then decide whether to continue discovering or stop.

The Response comes in callback `params->func`. The callback is run from the BT RX thread. `params` must remain valid until start of callback where iter `attr` is `NULL` or callback will return `BT_GATT_ITER_STOP`.

This function will block while the ATT request queue is full, except when called from the BT RX thread, as this would cause a deadlock.

Parameters

- **conn** – Connection object.
- **params** – Discover parameters.

Return values

- **0** – Successfully queued request. Will call `params->func` on resolution.
- **-ENOMEM** – ATT request queue is full and blocking would cause deadlock. Allow a pending request to resolve before retrying, or call this function outside the BT RX thread to get blocking behavior. Queue size is controlled by `CONFIG_BT_ATT_TX_COUNT` .

int **bt_gatt_read**(struct bt_conn *conn, struct *bt_gatt_read_params* *params)

Read Attribute Value by handle.

This procedure read the attribute value and return it to the callback.

When reading attributes by UUID the callback can be called multiple times depending on how many instances of given the UUID exists with the start_handle being updated for each instance.

To perform a GATT Long Read procedure, start with a Characteristic Value Read (by setting `offset` `0` and `handle_count` `1`) and then return *BT_GATT_ITER_CONTINUE* from the callback. This is equivalent to calling *bt_gatt_read* again, but with the correct offset to continue the read. This may be repeated until the procedure is complete, which is signaled by the callback being called with `data` set to `NULL`.

Note that returning *BT_GATT_ITER_CONTINUE* is really starting a new ATT operation, so this can fail to allocate resources. However, all API errors are reported as if the server returned *BT_ATT_ERR_UNLIKELY*. There is no way to distinguish between this condition and a *BT_ATT_ERR_UNLIKELY* response from the server itself.

Note that the effect of returning *BT_GATT_ITER_CONTINUE* from the callback varies depending on the type of read operation.

The Response comes in callback `params->func`. The callback is run from the context specified by 'config BT_RECV_CONTEXT'. `params` must remain valid until start of callback.

This function will block while the ATT request queue is full, except when called from the BT RX thread, as this would cause a deadlock.

Parameters

- **conn** – Connection object.
- **params** – Read parameters.

Return values

- **0** – Successfully queued request. Will call `params->func` on resolution.
- **-ENOMEM** – ATT request queue is full and blocking would cause deadlock. Allow a pending request to resolve before retrying, or call this function outside the BT RX thread to get blocking behavior. Queue size is controlled by `CONFIG_BT_ATT_TX_COUNT` .

int **bt_gatt_write**(struct bt_conn *conn, struct *bt_gatt_write_params* *params)

Write Attribute Value by handle.

The Response comes in callback `params->func`. The callback is run from the context specified by 'config BT_RECV_CONTEXT'. `params` must remain valid until start of callback.

This function will block while the ATT request queue is full, except when called from Bluetooth event context. When called from Bluetooth context, this function will instead instead return `-ENOMEM` if it would block to avoid a deadlock.

Parameters

- **conn** – Connection object.
- **params** – Write parameters.

Return values

- **0** – Successfully queued request. Will call `params->func` on resolution.
- **-ENOMEM** – ATT request queue is full and blocking would cause deadlock. Allow a pending request to resolve before retrying, or call this function outside Bluetooth event context to get blocking behavior. Queue size is controlled by `CONFIG_BT_ATT_TX_COUNT` .

int **bt_gatt_write_without_response_cb**(struct bt_conn *conn, uint16_t handle, const void *data, uint16_t length, bool sign, *bt_gatt_complete_func_t* func, void *user_data)

Write Attribute Value by handle without response with callback.

This function works in the same way as *bt_gatt_write_without_response*. With the addition that after sending the write the callback function will be called.

The callback is run from System Workqueue context. When called from the System Workqueue context this API will not wait for resources for the callback but instead return an error. The number of pending callbacks can be increased with the `CONFIG_BT_CONN_TX_MAX` option.

This function will block while the ATT request queue is full, except when called from the BT RX thread, as this would cause a deadlock.

Note

By using a callback it also disable the internal flow control which would prevent sending multiple commands without waiting for their transmissions to complete, so if that is required the caller shall not submit more data until the callback is called.

Parameters

- `conn` – Connection object.
- `handle` – Attribute handle.
- `data` – Data to be written.
- `length` – Data length.
- `sign` – Whether to sign data
- `func` – Transmission complete callback.
- `user_data` – User data to be passed back to callback.

Return values

- `0` – Successfully queued request.
- `-ENOMEM` – ATT request queue is full and blocking would cause deadlock. Allow a pending request to resolve before retrying, or call this function outside the BT RX thread to get blocking behavior. Queue size is controlled by `CONFIG_BT_ATT_TX_COUNT`.

static inline int **bt_gatt_write_without_response**(struct bt_conn *conn, uint16_t handle, const void *data, uint16_t length, bool sign)

Write Attribute Value by handle without response.

This procedure write the attribute value without requiring an acknowledgment that the write was successfully performed

This function will block while the ATT request queue is full, except when called from the BT RX thread, as this would cause a deadlock.

Parameters

- `conn` – Connection object.
- `handle` – Attribute handle.
- `data` – Data to be written.
- `length` – Data length.
- `sign` – Whether to sign data

Return values

- `0` – Successfully queued request.
- `-ENOMEM` – ATT request queue is full and blocking would cause deadlock. Allow a pending request to resolve before retrying, or call this function outside the BT RX thread to get blocking behavior. Queue size is controlled by `CONFIG_BT_ATT_TX_COUNT`.

int **bt_gatt_subscribe**(struct bt_conn *conn, struct *bt_gatt_subscribe_params* *params)

Subscribe Attribute Value Notification.

This procedure subscribe to value notification using the Client Characteristic Configuration handle. If notification received subscribe value callback is called to return notified value. One may then decide whether to unsubscribe directly from this callback. Notification callback with NULL data will not be called if subscription was removed by this method.

The Response comes in callback `params->subscribe`. The callback is run from the context specified by 'config BT_RECV_CONTEXT'. The Notification callback `params->notify` is also called from the BT RX thread.

This function will block while the ATT request queue is full, except when called from the BT RX thread, as this would cause a deadlock.

Note

Notifications are asynchronous therefore the `params` must remain valid while subscribed and cannot be reused for additional subscriptions whilst active.

Parameters

- **conn** – Connection object.
- **params** – Subscribe parameters.

Return values

- **0** – Successfully queued request. Will call `params->write` on resolution.
- **-ENOMEM** – ATT request queue is full and blocking would cause deadlock. Allow a pending request to resolve before retrying, or call this function outside the BT RX thread to get blocking behavior. Queue size is controlled by `CONFIG_BT_ATT_TX_COUNT` .
- **-EALREADY** – if there already exist a subscription using the `params`.
- **-EBUSY** – if `params.ccc_handle` is 0 and `CONFIG_BT_GATT_AUTO_DISCOVER_CCC` is enabled and discovery for the `params` is already in progress.

int **bt_gatt_resubscribe**(uint8_t id, const *bt_addr_le_t* *peer, struct *bt_gatt_subscribe_params* *params)

Resubscribe Attribute Value Notification subscription.

Resubscribe to Attribute Value Notification when already subscribed from a previous connection. The GATT server will remember subscription from previous connections when bonded, so resubscribing can be done without performing a new subscribe procedure after a power cycle.

Note

Notifications are asynchronous therefore the parameters need to remain valid while subscribed.

Parameters

- **id** – Local identity (in most cases BT_ID_DEFAULT).

- **peer** – Remote address.
- **params** – Subscribe parameters.

Returns
0 in case of success or negative value in case of error.

int **bt_gatt_unsubscribe**(struct bt_conn *conn, struct *bt_gatt_subscribe_params* *params)

Unsubscribe Attribute Value Notification.

This procedure unsubscribe to value notification using the Client Characteristic Configuration handle. Notification callback with NULL data will be called if subscription was removed by this call, until then the parameters cannot be reused.

The Response comes in callback `params->func`. The callback is run from the BT RX thread.

This function will block while the ATT request queue is full, except when called from the BT RX thread, as this would cause a deadlock.

Parameters

- **conn** – Connection object.
- **params** – Subscribe parameters. The parameters shall be a *bt_gatt_subscribe_params* from a previous call to *bt_gatt_subscribe()*.

Return values

- **0** – Successfully queued request. Will call `params->write` on resolution.
- **-ENOMEM** – ATT request queue is full and blocking would cause deadlock. Allow a pending request to resolve before retrying, or call this function outside the BT RX thread to get blocking behavior. Queue size is controlled by `CONFIG_BT_ATT_TX_COUNT`.

void **bt_gatt_cancel**(struct bt_conn *conn, void *params)

Try to cancel the first pending request identified by `params`.

This function does not release `params` for reuse. The usual callbacks for the request still apply. A successful cancel simulates a *BT_ATT_ERR_UNLIKELY* response from the server.

This function can cancel the following request functions:

- *bt_gatt_exchange_mtu*
- *bt_gatt_discover*
- *bt_gatt_read*
- *bt_gatt_write*
- *bt_gatt_subscribe*
- *bt_gatt_unsubscribe*

Parameters

- **conn** – The connection the request was issued on.
- **params** – The address `params` used in the request function call.

struct **bt_gatt_exchange_params**

#include <gatt.h> GATT Exchange MTU parameters.

Public Members

void (*func)(struct bt_conn *conn, uint8_t err, struct *bt_gatt_exchange_params* *params)

Response callback.

struct **bt_gatt_discover_params**

#include <*gatt.h*> GATT Discover Attributes parameters.

Public Members

const struct *bt_uuid* ***uuid**

Discover UUID type.

bt_gatt_discover_func_t **func**

Discover attribute callback.

uint16_t **attr_handle**

Include service attribute declaration handle.

uint16_t **start_handle**

Included service start handle.

Discover start handle.

uint16_t **end_handle**

Included service end handle.

Discover end handle.

uint8_t **type**

Discover type.

struct *bt_gatt_subscribe_params* ***sub_params**

Only for stack-internal use, used for automatic discovery.

struct **bt_gatt_read_params**

#include <*gatt.h*> GATT Read parameters.

Public Members

bt_gatt_read_func_t **func**

Read attribute callback.

size_t **handle_count**

If equals to 1 single.handle and single.offset are used.

If greater than 1 multiple.handles are used. If equals to 0 by_uuid is used for Read Using Characteristic UUID.

uint16_t **handle**

Attribute handle.

uint16_t **offset**

Attribute data offset.

uint16_t ***handles**

Attribute handles to read with Read Multiple Characteristic Values.

bool **variable**

If true use Read Multiple Variable Length Characteristic Values procedure.

The values of the set of attributes may be of variable or unknown length. If false use Read Multiple Characteristic Values procedure. The values of the set of attributes must be of a known fixed length, with the exception of the last value that can have a variable length.

uint16_t **start_handle**

First requested handle number.

uint16_t **end_handle**

Last requested handle number.

const struct *bt_uuid* ***uuid**

2 or 16 octet UUID.

struct **bt_gatt_write_params**

#include <gatt.h> GATT Write parameters.

Public Members

bt_gatt_write_func_t **func**

Response callback.

uint16_t **handle**

Attribute handle.

uint16_t **offset**

Attribute data offset.

const void ***data**

Data to be written.

uint16_t **length**

Length of the data.

struct **bt_gatt_subscribe_params**

#include <gatt.h> GATT Subscribe parameters.

Public Members

bt_gatt_notify_func_t **notify**

Notification value callback.

bt_gatt_subscribe_func_t **subscribe**

Subscribe CCC write request response callback If given, called with the subscription parameters given when subscribing.

uint16_t **value_handle**

Subscribe value handle.

uint16_t **ccc_handle**

Subscribe CCC handle.

uint16_t **end_handle**

Subscribe End handle (for automatic discovery)

struct *bt_gatt_discover_params* ***disc_params**

Discover parameters used when ccc_handle = 0.

uint16_t **value**

Subscribe value.

bt_security_t **min_security**

Minimum required security for received notification.

Notifications and indications received over a connection with a lower security level are silently discarded.

atomic_t **flags**[*ATOMIC_BITMAP_SIZE*(BT_GATT_SUBSCRIBE_NUM_FLAGS)]

Subscription flags.

Attribute Protocol (ATT)

API Reference

group **bt_att**

Attribute Protocol (ATT)

Defines

BT_ATT_ERR_SUCCESS

The ATT operation was successful.

BT_ATT_ERR_INVALID_HANDLE

The attribute handle given was not valid on the server.

BT_ATT_ERR_READ_NOT_PERMITTED
: The attribute cannot be read.

BT_ATT_ERR_WRITE_NOT_PERMITTED
: The attribute cannot be written.

BT_ATT_ERR_INVALID_PDU
: The attribute PDU was invalid.

BT_ATT_ERR_AUTHENTICATION
: The attribute requires authentication before it can be read or written.

BT_ATT_ERR_NOT_SUPPORTED
: The ATT Server does not support the request received from the client.

BT_ATT_ERR_INVALID_OFFSET
: Offset specified was past the end of the attribute.

BT_ATT_ERR_AUTHORIZATION
: The attribute requires authorization before it can be read or written.

BT_ATT_ERR_PREPARE_QUEUE_FULL
: Too many prepare writes have been queued.

BT_ATT_ERR_ATTRIBUTE_NOT_FOUND
: No attribute found within the given attribute handle range.

BT_ATT_ERR_ATTRIBUTE_NOT_LONG
: The attribute cannot be read using the ATT_READ_BLOB_REQ PDU.

BT_ATT_ERR_ENCRYPTION_KEY_SIZE
: The Encryption Key Size used for encrypting this link is too short.

BT_ATT_ERR_INVALID_ATTRIBUTE_LEN
: The attribute value length is invalid for the operation.

BT_ATT_ERR_UNLIKELY
: The attribute request that was requested has encountered an error that was unlikely.

  The attribute request could therefore not be completed as requested

BT_ATT_ERR_INSUFFICIENT_ENCRYPTION
: The attribute requires encryption before it can be read or written.

BT_ATT_ERR_UNSUPPORTED_GROUP_TYPE
: The attribute type is not a supported grouping attribute.

  The attribute type is not a supported grouping attribute as defined by a higher layer specification.

BT_ATT_ERR_INSUFFICIENT_RESOURCES
: Insufficient Resources to complete the request.

BT_ATT_ERR_DB_OUT_OF_SYNC
: The server requests the client to rediscover the database.

BT_ATT_ERR_VALUE_NOT_ALLOWED
: The attribute parameter value was not allowed.

BT_ATT_ERR_WRITE_REQ_REJECTED
: Write Request Rejected.

BT_ATT_ERR_CCC_IMPROPER_CONF
: Client Characteristic Configuration Descriptor Improperly Configured.

BT_ATT_ERR_PROCEDURE_IN_PROGRESS
: Procedure Already in Progress.

BT_ATT_ERR_OUT_OF_RANGE
: Out of Range.

BT_ATT_MAX_ATTRIBUTE_LEN

BT_ATT_FIRST_ATTRIBUTE_HANDLE

BT_ATT_LAST_ATTRIBUTE_HANDLE

Enums

enum **bt_att_chan_opt**
: ATT channel option bit field values.

> **Note**
>
> *BT_ATT_CHAN_OPT_UNENHANCED_ONLY* and *BT_ATT_CHAN_OPT_ENHANCED_ONLY* are mutually exclusive and both bits may not be set.

Values:

enumerator **BT_ATT_CHAN_OPT_NONE** = 0x0
: Both Enhanced and Unenhanced channels can be used

enumerator **BT_ATT_CHAN_OPT_UNENHANCED_ONLY** = *BIT*(0)
: Only Unenhanced channels will be used

enumerator **BT_ATT_CHAN_OPT_ENHANCED_ONLY** = *BIT*(1)
: Only Enhanced channels will be used

Functions

static inline const char *bt_att_err_to_str(uint8_t att_err)

Converts a ATT error to string.

The error codes are described in the Bluetooth Core specification, Vol 3, Part F, Section 3.4.1.1 and in The Supplement to the Bluetooth Core Specification (CSS), v11, Part B, Section 1.2.

The ATT and GATT documentation found in Vol 4, Part F and Part G describe when the different error codes are used.

See also the defined BT_ATT_ERR_* macros.

Returns
The string representation of the ATT error code. If CONFIG_BT_ATT_ERR_TO_STR is not enabled, this just returns the empty string

int bt_eatt_connect(struct bt_conn *conn, size_t num_channels)

Connect Enhanced ATT channels.

Sends a series of Credit Based Connection Requests to connect num_channels Enhanced ATT channels. The peer may have limited resources and fewer channels may be created.

Parameters
- conn – The connection to send the request on
- num_channels – The number of Enhanced ATT beares to request. Must be in the range 1 - CONFIG_BT_EATT_MAX , inclusive.

Return values
- -EINVAL – if num_channels is not in the allowed range or conn is NULL.
- -ENOMEM – if less than num_channels are allocated.
- 0 – in case of success

Returns
0 in case of success or negative value in case of error.

size_t bt_eatt_count(struct bt_conn *conn)

Get number of EATT channels connected.

Parameters
- conn – The connection to get the number of EATT channels for.

Returns
The number of EATT channels connected. Returns 0 if conn is NULL or not connected.

Bluetooth Mesh

Bluetooth Mesh Profile The Bluetooth Mesh profile adds secure wireless multi-hop communication for Bluetooth Low Energy. This module implements the Bluetooth Mesh Protocol Specification v1.1.

Read more about Bluetooth Mesh on the Bluetooth SIG Website.

Core The core provides functionality for managing the general Bluetooth Mesh state.

Low Power Node The Low Power Node (LPN) role allows battery powered devices to participate in a mesh network as a leaf node. An LPN interacts with the mesh network through a Friend node, which is responsible for relaying any messages directed to the LPN. The LPN saves power by keeping its radio turned off, and only wakes up to either send messages or poll the Friend node for any incoming messages.

The radio control and polling is managed automatically by the mesh stack, but the LPN API allows the application to trigger the polling at any time through *bt_mesh_lpn_poll()*. The LPN operation parameters, including poll interval, poll event timing and Friend requirements is controlled through the CONFIG_BT_MESH_LOW_POWER option and related configuration options.

When using the LPN feature with logging, it is strongly recommended to only use the CONFIG_LOG_MODE_DEFERRED option. Log modes other than the deferred may cause unintended delays during processing of log messages. This in turns will affect scheduling of the receive delay and receive window. The same limitation applies for the CONFIG_BT_MESH_FRIEND option.

Replay Protection List The Replay Protection List (RPL) is used to hold recently received sequence numbers from elements within the mesh network to perform protection against replay attacks.

To keep a node protected against replay attacks after reboot, it needs to store the entire RPL in the persistent storage before it is powered off. Depending on the amount of traffic in a mesh network, storing recently seen sequence numbers can make flash wear out sooner or later. To mitigate this, CONFIG_BT_MESH_RPL_STORE_TIMEOUT can be used. This option postpones storing of RPL entries in the persistent storage.

This option, however, doesn't completely solve the issue as the node may get powered off before the timer to store the RPL is fired. To ensure that messages can not be replayed, the node can initiate storage of the pending RPL entry (or entries) at any time (or sufficiently before power loss) by calling *bt_mesh_rpl_pending_store()*. This is up to the node to decide, which RPL entries are to be stored in this case.

Setting CONFIG_BT_MESH_RPL_STORE_TIMEOUT to -1 allows to completely switch off the timer, which can help to significantly reduce flash wear out. This moves the responsibility of storing RPL to the user application and requires that sufficient power backup is available from the time this API is called until all RPL entries are written to the flash.

Finding the right balance between CONFIG_BT_MESH_RPL_STORE_TIMEOUT and calling *bt_mesh_rpl_pending_store()* may reduce a risk of security vulnerability and flash wear out.

Persistent storage The mesh stack uses the *Settings Subsystem* for storing the device configuration persistently. When the stack configuration changes and the change needs to be stored persistently, the stack schedules a work item. The delay between scheduling the work item and submitting it to the workqueue is defined by the CONFIG_BT_MESH_STORE_TIMEOUT option. Once storing of data is scheduled, it can not be rescheduled until the work item is processed. Exceptions are made in certain cases as described below.

When IV index, Sequence Number or CDB configuration have to be stored, the work item is submitted to the workqueue without the delay. If the work item was previously scheduled, it will be rescheduled without the delay.

The Replay Protection List uses the same work item to store RPL entries. If storing of RPL entries is requested and no other configuration is pending to be stored, the delay is set to CONFIG_BT_MESH_RPL_STORE_TIMEOUT. If other stack configuration has to be stored, the delay defined by the CONFIG_BT_MESH_STORE_TIMEOUT option is less than CONFIG_BT_MESH_RPL_STORE_TIMEOUT, and the work item was scheduled by the Replay Protection List, the work item will be rescheduled.

When the work item is running, the stack will store all pending configuration, including the RPL entries.

Work item execution context The CONFIG_BT_MESH_SETTINGS_WORKQ option configures the context from which the work item is executed. This option is enabled by default, and results in stack using a dedicated cooperative thread to process the work item. This allows the stack to process other incoming and outgoing messages, as well as other work items submitted to the system workqueue, while the stack configuration is being stored.

When this option is disabled, the work item is submitted to the system workqueue. This means that the system workqueue is blocked for the time it takes to store the stack's configuration. It is not recommended to disable this option as this will make the device non-responsive for a noticeable amount of time.

Advertisement identity All mesh stack bearers advertise data with the *BT_ID_DEFAULT* local identity. The value is preset in the mesh stack implementation. When Bluetooth® Low Energy (LE) and Bluetooth Mesh coexist on the same device, the application should allocate and configure another local identity for Bluetooth LE purposes before starting the communication.

API reference

group **bt_mesh**

Bluetooth Mesh.

Defines

BT_MESH_NET_PRIMARY

Primary Network Key index.

BT_MESH_FEAT_RELAY

Relay feature.

BT_MESH_FEAT_PROXY

GATT Proxy feature.

BT_MESH_FEAT_FRIEND

Friend feature.

BT_MESH_FEAT_LOW_POWER

Low Power Node feature.

BT_MESH_FEAT_SUPPORTED

Supported heartbeat publication features.

BT_MESH_LPN_CB_DEFINE(_name)

Register a callback structure for Friendship events.

Parameters

- **_name** – Name of callback structure.

BT_MESH_FRIEND_CB_DEFINE(_name)

Register a callback structure for Friendship events.

Registers a callback structure that will be called whenever Friendship gets established or terminated.

Parameters

- **_name** – Name of callback structure.

Functions

int **bt_mesh_init**(const struct *bt_mesh_prov* *prov, const struct *bt_mesh_comp* *comp)

Initialize Mesh support.

After calling this API, the node will not automatically advertise as unprovisioned, rather the *bt_mesh_prov_enable()* API needs to be called to enable unprovisioned advertising on one or more provisioning bearers.

Parameters

- **prov** – Node provisioning information.
- **comp** – Node Composition.

Returns

Zero on success or (negative) error code otherwise.

void **bt_mesh_reset**(void)

Reset the state of the local Mesh node.

Resets the state of the node, which means that it needs to be reprovisioned to become an active node in a Mesh network again.

After calling this API, the node will not automatically advertise as unprovisioned, rather the *bt_mesh_prov_enable()* API needs to be called to enable unprovisioned advertising on one or more provisioning bearers.

int **bt_mesh_suspend**(void)

Suspend the Mesh network temporarily.

This API can be used for power saving purposes, but the user should be aware that leaving the local node suspended for a long period of time may cause it to become permanently disconnected from the Mesh network. If at all possible, the Friendship feature should be used instead, to make the node into a Low Power Node.

Returns

0 on success, or (negative) error code on failure.

int **bt_mesh_resume**(void)

Resume a suspended Mesh network.

This API resumes the local node, after it has been suspended using the *bt_mesh_suspend()* API.

Returns

0 on success, or (negative) error code on failure.

void **bt_mesh_iv_update_test**(bool enable)

Toggle the IV Update test mode.

This API is only available if the IV Update test mode has been enabled in Kconfig. It is needed for passing most of the IV Update qualification test cases.

Parameters

- **enable** – true to enable IV Update test mode, false to disable it.

bool **bt_mesh_iv_update**(void)

Toggle the IV Update state.

This API is only available if the IV Update test mode has been enabled in Kconfig. It is needed for passing most of the IV Update qualification test cases.

Returns
true if IV Update In Progress state was entered, false otherwise.

int **bt_mesh_lpn_set**(bool enable)

Toggle the Low Power feature of the local device.

Enables or disables the Low Power feature of the local device. This is exposed as a runtime feature, since the device might want to change this e.g. based on being plugged into a stable power source or running from a battery power source.

Parameters

- **enable** – true to enable LPN functionality, false to disable it.

Returns
Zero on success or (negative) error code otherwise.

int **bt_mesh_lpn_poll**(void)

Send out a Friend Poll message.

Send a Friend Poll message to the Friend of this node. If there is no established Friendship the function will return an error.

Returns
Zero on success or (negative) error code otherwise.

int **bt_mesh_friend_terminate**(uint16_t lpn_addr)

Terminate Friendship.

Terminated Friendship for given LPN.

Parameters

- **lpn_addr** – Low Power Node address.

Returns
Zero on success or (negative) error code otherwise.

void **bt_mesh_rpl_pending_store**(uint16_t addr)

Store pending RPL entry(ies) in the persistent storage.

This API allows the user to store pending RPL entry(ies) in the persistent storage without waiting for the timeout.

Note

When flash is used as the persistent storage, calling this API too frequently may wear it out.

Parameters

- **addr** – Address of the node which RPL entry needs to be stored or *BT_MESH_ADDR_ALL_NODES* to store all pending RPL entries.

const uint8_t *__bt_mesh_va_uuid_get__(uint16_t addr, const uint8_t *uuid, uint16_t *retaddr)

Iterate stored Label UUIDs.

When `addr` is *BT_MESH_ADDR_UNASSIGNED*, this function iterates over all available addresses starting with `uuid`. In this case, use `retaddr` to get virtual address representation of the returned Label UUID. When `addr` is a virtual address, this function returns next Label UUID corresponding to the `addr`. When `uuid` is NULL, this function returns the first available UUID. If `uuid` is previously returned uuid, this function returns following uuid.

Parameters

- `addr` – Virtual address to search for, or *BT_MESH_ADDR_UNASSIGNED*.
- `uuid` – Pointer to the previously returned Label UUID or NULL.
- `retaddr` – Pointer to a memory where virtual address representation of the returning UUID is to be stored to.

Returns
Pointer to Label UUID, or NULL if no more entries found.

struct `bt_mesh_lpn_cb`

#include <main.h> Low Power Node callback functions.

Public Members

void (*`established`)(uint16_t net_idx, uint16_t friend_addr, uint8_t queue_size, uint8_t recv_window)

Friendship established.

This callback notifies the application that friendship has been successfully established.

Param net_idx
NetKeyIndex used during friendship establishment.
Param friend_addr
Friend address.
Param queue_size
Friend queue size.
Param recv_window
Low Power Node's listens duration for Friend response.

void (*`terminated`)(uint16_t net_idx, uint16_t friend_addr)

Friendship terminated.

This callback notifies the application that friendship has been terminated.

Param net_idx
NetKeyIndex used during friendship establishment.
Param friend_addr
Friend address.

void (*`polled`)(uint16_t net_idx, uint16_t friend_addr, bool retry)

Local Poll Request.

This callback notifies the application that the local node has polled the friend node.

This callback will be called before *bt_mesh_lpn_cb::established* when attempting to establish a friendship.

Param net_idx
NetKeyIndex used during friendship establishment.
Param friend_addr
Friend address.
Param retry
Retry or first poll request for each transaction.

struct `bt_mesh_friend_cb`

#include <main.h> Friend Node callback functions.

Public Members

void (*`established`)(uint16_t net_idx, uint16_t lpn_addr, uint8_t recv_delay, uint32_t polltimeout)

Friendship established.

This callback notifies the application that friendship has been successfully established.

Param net_idx
NetKeyIndex used during friendship establishment.

Param lpn_addr
Low Power Node address.

Param recv_delay
Receive Delay in units of 1 millisecond.

Param polltimeout
PollTimeout in units of 1 millisecond.

void (*`terminated`)(uint16_t net_idx, uint16_t lpn_addr)

Friendship terminated.

This callback notifies the application that friendship has been terminated.

Param net_idx
NetKeyIndex used during friendship establishment.

Param lpn_addr
Low Power Node address.

void (*`polled`)(uint16_t net_idx, uint16_t lpn_addr)

Friend Poll Request.

This callback notifies the application that the low power node has polled the friend node.

This callback will be called before *bt_mesh_friend_cb::established* when attempting to establish a friendship.

Param net_idx
NetKeyIndex used during friendship establishment.

Param lpn_addr
LPN address.

Access layer The access layer is the application's interface to the Bluetooth Mesh network. The access layer provides mechanisms for compartmentalizing the node behavior into elements and models, which are implemented by the application.

Mesh models The functionality of a mesh node is represented by models. A model implements a single behavior the node supports, like being a light, a sensor or a thermostat. The mesh models are grouped into *elements*. Each element is assigned its own unicast address, and may only contain one of each type of model. Conventionally, each element represents a single aspect of the mesh node behavior. For instance, a node that contains a sensor, two lights and a power outlet would spread this functionality across four elements, with each element instantiating all the models required for a single aspect of the supported behavior.

The node's element and model structure is specified in the node composition data, which is passed to *`bt_mesh_init()`* during initialization. The Bluetooth SIG have defined a set of foundation models (see *Mesh models*) and a set of models for implementing common behavior in the Bluetooth Mesh Model Specification. All models not specified by the Bluetooth SIG are vendor models, and must be tied to a Company ID.

Mesh models have several parameters that can be configured either through initialization of the mesh stack or with the *Configuration Server*:

Opcode list The opcode list contains all message opcodes the model can receive, as well as the minimum acceptable payload length and the callback to pass them to. Models can support any number of opcodes, but each opcode can only be listed by one model in each element.

The full opcode list must be passed to the model structure in the composition data, and cannot be changed at runtime. The end of the opcode list is determined by the special *BT_MESH_MODEL_OP_END* entry. This entry must always be present in the opcode list, unless the list is empty. In that case, *BT_MESH_MODEL_NO_OPS* should be used in place of a proper opcode list definition.

AppKey list The AppKey list contains all the application keys the model can receive messages on. Only messages encrypted with application keys in the AppKey list will be passed to the model.

The maximum number of supported application keys each model can hold is configured with the CONFIG_BT_MESH_MODEL_KEY_COUNT configuration option. The contents of the AppKey list is managed by the *Configuration Server*.

Subscription list A model will process all messages addressed to the unicast address of their element (given that the utilized application key is present in the AppKey list). Additionally, the model will process packets addressed to any group or virtual address in its subscription list. This allows nodes to address multiple nodes throughout the mesh network with a single message.

The maximum number of supported addresses in the Subscription list each model can hold is configured with the CONFIG_BT_MESH_MODEL_GROUP_COUNT configuration option. The contents of the subscription list is managed by the *Configuration Server*.

Model publication The models may send messages in two ways:

- By specifying a set of message parameters in a *bt_mesh_msg_ctx*, and calling *bt_mesh_model_send()*.
- By setting up a *bt_mesh_model_pub* structure and calling *bt_mesh_model_publish()*.

When publishing messages with *bt_mesh_model_publish()*, the model will use the publication parameters configured by the *Configuration Server*. This is the recommended way to send unprompted model messages, as it passes the responsibility of selecting message parameters to the network administrator, which likely knows more about the mesh network than the individual nodes will.

To support publishing with the publication parameters, the model must allocate a packet buffer for publishing, and pass it to *bt_mesh_model_pub.msg*. The Config Server may also set up period publication for the publication message. To support this, the model must populate the *bt_mesh_model_pub.update* callback. The *bt_mesh_model_pub.update* callback will be called right before the message is published, allowing the model to change the payload to reflect its current state.

By setting *bt_mesh_model_pub.retr_update* to 1, the model can configure the *bt_mesh_model_pub.update* callback to be triggered on every retransmission. This can, for example, be used by models that make use of a Delay parameter, which can be adjusted for every retransmission. The *bt_mesh_model_pub_is_retransmission()* function can be used to differentiate a first publication and a retransmission. The *BT_MESH_PUB_MSG_TOTAL* and *BT_MESH_PUB_MSG_NUM* macros can be used to return total number of transmissions and the retransmission number within one publication interval.

Extended models The Bluetooth Mesh specification allows the mesh models to extend each other. When a model extends another, it inherits that model's functionality, and extension can be used to construct complex models out of simple ones, leveraging the existing model functionality to avoid defining new opcodes. Models may extend any number of models, from any element. When one model extends another in the same element, the two models will share subscription lists. The mesh stack implements this by merging the subscription lists of the two models into one, combining the number of subscriptions the models can have in total. Models may extend models that extend others, creating an "extension tree". All models in an extension tree share a single subscription list per element it spans.

Model extensions are done by calling *bt_mesh_model_extend()* during initialization. A model can only be extended by one other model, and extensions cannot be circular. Note that binding of node states and other relationships between the models must be defined by the model implementations.

The model extension concept adds some overhead in the access layer packet processing, and must be explicitly enabled with CONFIG_BT_MESH_MODEL_EXTENSIONS to have any effect.

Model data storage Mesh models may have data associated with each model instance that needs to be stored persistently. The access API provides a mechanism for storing this data, leveraging the internal model instance encoding scheme. Models can store one user defined data entry per instance by calling *bt_mesh_model_data_store()*. To be able to read out the data the next time the device reboots, the model's *bt_mesh_model_cb.settings_set* callback must be populated. This callback gets called when model specific data is found in the persistent storage. The model can retrieve the data by calling the read_cb passed as a parameter to the callback. See the *Settings* module documentation for details.

When model data changes frequently, storing it on every change may lead to increased wear of flash. To reduce the wear, the model can postpone storing of data by calling *bt_mesh_model_data_store_schedule()*. The stack will schedule a work item with delay defined by the CONFIG_BT_MESH_STORE_TIMEOUT option. When the work item is running, the stack will call the *bt_mesh_model_cb.pending_store* callback for every model that has requested storing of data. The model can then call *bt_mesh_model_data_store()* to store the data.

If CONFIG_BT_MESH_SETTINGS_WORKQ is enabled, the *bt_mesh_model_cb.pending_store* callback is called from a dedicated thread. This allows the stack to process other incoming and outgoing messages while model data is being stored. It is recommended to use this option and the *bt_mesh_model_data_store_schedule()* function when large amount of data needs to be stored.

Composition Data The Composition Data provides information about a mesh device. A device's Composition Data holds information about the elements on the device, the models that it supports, and other features. The Composition Data is split into different pages, where each page contains specific feature information about the device. In order to access this information, the user may use the *Configuration Server* model or, if supported, the *Large Composition Data Server* model.

Composition Data Page 0 Composition Data Page 0 provides the fundamental information about a device, and is mandatory for all mesh devices. It contains the element and model composition, the supported features, and manufacturer information.

Composition Data Page 1 Composition Data Page 1 provides information about the relationships between models, and is mandatory for all mesh devices. A model may extend and/or correspond to one or more models. A model can extend another model by calling *bt_mesh_model_extend()*, or correspond to another model by calling *bt_mesh_model_correspond()*. CONFIG_BT_MESH_MODEL_EXTENSION_LIST_SIZE specifies how many model relations can be stored in the composition on a device, and this number should reflect the number of *bt_mesh_model_extend()* and *bt_mesh_model_correspond()* calls.

Composition Data Page 2 Composition Data Page 2 provides information for supported mesh profiles. Mesh profile specifications define product requirements for devices that want to support a specific Bluetooth SIG defined profile. Currently supported profiles can be found in section 3.12 in Bluetooth SIG Assigned Numbers. Composition Data Page 2 is only mandatory for devices that claim support for one or more mesh profile(s).

Composition Data Pages 128, 129 and 130 Composition Data Pages 128, 129 and 130 mirror Composition Data Pages 0, 1 and 2 respectively. They are used to represent the new content of the mirrored pages when the Composition Data will change after a firmware update. See *Composition Data and Models Metadata* for details.

Delayable messages The delayable message functionality is enabled with Kconfig option `CONFIG_BT_MESH_ACCESS_DELAYABLE_MSG`. This is an optional functionality that implements specification recommendations for messages that are transmitted by a model in a response to a received message, also called response messages.

Response messages should be sent with the following random delays:

- Between 20 and 50 milliseconds if the received message was sent to a unicast address
- Between 20 and 500 milliseconds if the received message was sent to a group or virtual address

The delayable message functionality is triggered if the *`bt_mesh_msg_ctx.rnd_delay`* flag is set. The delayable message functionality stores messages in the local memory while they are waiting for the random delay expiration.

If the transport layer doesn't have sufficient memory to send a message at the moment the random delay expires, the message is postponed for another 10 milliseconds. If the transport layer cannot send a message for any other reason, the delayable message functionality raises the *`bt_mesh_send_cb.start`* callback with a transport layer error code.

If the delayable message functionality cannot find enough free memory to store an incoming message, it will send messages with delay close to expiration to free memory.

When the mesh stack is suspended or reset, messages not yet sent are removed and the *`bt_mesh_send_cb.start`* callback is raised with an error code.

Note

When a model sends several messages in a row, it may happen that the messages are not sent in the order they were passed to the access layer. This is because some messages can be delayed for a longer time than the others.

Disable the randomization by setting the *`bt_mesh_msg_ctx.rnd_delay`* to `false`, when a set of messages originated by the same model needs to be sent in a certain order.

Delayable publications The delayable publication functionality implements the specification recommendations for message publication delays in the following cases:

- Between 20 to 500 milliseconds when the Bluetooth Mesh stack starts or when the publication is triggered by the *`bt_mesh_model_publish()`* function
- Between 20 to 50 milliseconds for periodically published messages

This feature is optional and enabled with the `CONFIG_BT_MESH_DELAYABLE_PUBLICATION` Kconfig option. When enabled, each model can enable or disable the delayable publication by setting the *`bt_mesh_model_pub.delayable`* bit field to `1` or `0` correspondingly. This bit field can be changed at any time.

API reference

group `bt_mesh_access`

Access layer.

Group addresses

`BT_MESH_ADDR_UNASSIGNED`
: unassigned

`BT_MESH_ADDR_ALL_NODES`
: all-nodes

`BT_MESH_ADDR_RELAYS`
: all-relays

`BT_MESH_ADDR_FRIENDS`
: all-friends

`BT_MESH_ADDR_PROXIES`
: all-proxies

`BT_MESH_ADDR_DFW_NODES`
: all-directed-forwarding-nodes

`BT_MESH_ADDR_IP_NODES`
: all-ipt-nodes

`BT_MESH_ADDR_IP_BR_ROUTERS`
: all-ipt-border-routers

Predefined key indexes

`BT_MESH_KEY_UNUSED`
: Key unused.

`BT_MESH_KEY_ANY`
: Any key index.

`BT_MESH_KEY_DEV`
: Device key.

`BT_MESH_KEY_DEV_LOCAL`
: Local device key.

`BT_MESH_KEY_DEV_REMOTE`
: Remote device key.

BT_MESH_KEY_DEV_ANY
: Any device key.

Foundation Models

BT_MESH_MODEL_ID_CFG_SRV
: Configuration Server.

BT_MESH_MODEL_ID_CFG_CLI
: Configuration Client.

BT_MESH_MODEL_ID_HEALTH_SRV
: Health Server.

BT_MESH_MODEL_ID_HEALTH_CLI
: Health Client.

BT_MESH_MODEL_ID_REMOTE_PROV_SRV
: Remote Provisioning Server.

BT_MESH_MODEL_ID_REMOTE_PROV_CLI
: Remote Provisioning Client.

BT_MESH_MODEL_ID_PRIV_BEACON_SRV
: Private Beacon Server.

BT_MESH_MODEL_ID_PRIV_BEACON_CLI
: Private Beacon Client.

BT_MESH_MODEL_ID_SAR_CFG_SRV
: SAR Configuration Server.

BT_MESH_MODEL_ID_SAR_CFG_CLI
: SAR Configuration Client.

BT_MESH_MODEL_ID_OP_AGG_SRV
: Opcodes Aggregator Server.

BT_MESH_MODEL_ID_OP_AGG_CLI
: Opcodes Aggregator Client.

BT_MESH_MODEL_ID_LARGE_COMP_DATA_SRV
: Large Composition Data Server.

BT_MESH_MODEL_ID_LARGE_COMP_DATA_CLI
: Large Composition Data Client.

`BT_MESH_MODEL_ID_SOL_PDU_RPL_SRV`
: Solicitation PDU RPL Configuration Client.

`BT_MESH_MODEL_ID_SOL_PDU_RPL_CLI`
: Solicitation PDU RPL Configuration Server.

`BT_MESH_MODEL_ID_ON_DEMAND_PROXY_SRV`
: Private Proxy Server.

`BT_MESH_MODEL_ID_ON_DEMAND_PROXY_CLI`
: Private Proxy Client.

Models from the Mesh Model Specification

`BT_MESH_MODEL_ID_GEN_ONOFF_SRV`
: Generic OnOff Server.

`BT_MESH_MODEL_ID_GEN_ONOFF_CLI`
: Generic OnOff Client.

`BT_MESH_MODEL_ID_GEN_LEVEL_SRV`
: Generic Level Server.

`BT_MESH_MODEL_ID_GEN_LEVEL_CLI`
: Generic Level Client.

`BT_MESH_MODEL_ID_GEN_DEF_TRANS_TIME_SRV`
: Generic Default Transition Time Server.

`BT_MESH_MODEL_ID_GEN_DEF_TRANS_TIME_CLI`
: Generic Default Transition Time Client.

`BT_MESH_MODEL_ID_GEN_POWER_ONOFF_SRV`
: Generic Power OnOff Server.

`BT_MESH_MODEL_ID_GEN_POWER_ONOFF_SETUP_SRV`
: Generic Power OnOff Setup Server.

`BT_MESH_MODEL_ID_GEN_POWER_ONOFF_CLI`
: Generic Power OnOff Client.

`BT_MESH_MODEL_ID_GEN_POWER_LEVEL_SRV`
: Generic Power Level Server.

`BT_MESH_MODEL_ID_GEN_POWER_LEVEL_SETUP_SRV`
: Generic Power Level Setup Server.

`BT_MESH_MODEL_ID_GEN_POWER_LEVEL_CLI`
: Generic Power Level Client.

`BT_MESH_MODEL_ID_GEN_BATTERY_SRV`
: Generic Battery Server.

`BT_MESH_MODEL_ID_GEN_BATTERY_CLI`
: Generic Battery Client.

`BT_MESH_MODEL_ID_GEN_LOCATION_SRV`
: Generic Location Server.

`BT_MESH_MODEL_ID_GEN_LOCATION_SETUPSRV`
: Generic Location Setup Server.

`BT_MESH_MODEL_ID_GEN_LOCATION_CLI`
: Generic Location Client.

`BT_MESH_MODEL_ID_GEN_ADMIN_PROP_SRV`
: Generic Admin Property Server.

`BT_MESH_MODEL_ID_GEN_MANUFACTURER_PROP_SRV`
: Generic Manufacturer Property Server.

`BT_MESH_MODEL_ID_GEN_USER_PROP_SRV`
: Generic User Property Server.

`BT_MESH_MODEL_ID_GEN_CLIENT_PROP_SRV`
: Generic Client Property Server.

`BT_MESH_MODEL_ID_GEN_PROP_CLI`
: Generic Property Client.

`BT_MESH_MODEL_ID_SENSOR_SRV`
: Sensor Server.

`BT_MESH_MODEL_ID_SENSOR_SETUP_SRV`
: Sensor Setup Server.

`BT_MESH_MODEL_ID_SENSOR_CLI`
: Sensor Client.

`BT_MESH_MODEL_ID_TIME_SRV`
: Time Server.

`BT_MESH_MODEL_ID_TIME_SETUP_SRV`
: Time Setup Server.

`BT_MESH_MODEL_ID_TIME_CLI`
: Time Client.

`BT_MESH_MODEL_ID_SCENE_SRV`
: Scene Server.

`BT_MESH_MODEL_ID_SCENE_SETUP_SRV`
: Scene Setup Server.

`BT_MESH_MODEL_ID_SCENE_CLI`
: Scene Client.

`BT_MESH_MODEL_ID_SCHEDULER_SRV`
: Scheduler Server.

`BT_MESH_MODEL_ID_SCHEDULER_SETUP_SRV`
: Scheduler Setup Server.

`BT_MESH_MODEL_ID_SCHEDULER_CLI`
: Scheduler Client.

`BT_MESH_MODEL_ID_LIGHT_LIGHTNESS_SRV`
: Light Lightness Server.

`BT_MESH_MODEL_ID_LIGHT_LIGHTNESS_SETUP_SRV`
: Light Lightness Setup Server.

`BT_MESH_MODEL_ID_LIGHT_LIGHTNESS_CLI`
: Light Lightness Client.

`BT_MESH_MODEL_ID_LIGHT_CTL_SRV`
: Light CTL Server.

`BT_MESH_MODEL_ID_LIGHT_CTL_SETUP_SRV`
: Light CTL Setup Server.

`BT_MESH_MODEL_ID_LIGHT_CTL_CLI`
: Light CTL Client.

`BT_MESH_MODEL_ID_LIGHT_CTL_TEMP_SRV`
: Light CTL Temperature Server.

`BT_MESH_MODEL_ID_LIGHT_HSL_SRV`
: Light HSL Server.

`BT_MESH_MODEL_ID_LIGHT_HSL_SETUP_SRV`
: Light HSL Setup Server.

`BT_MESH_MODEL_ID_LIGHT_HSL_CLI`
: Light HSL Client.

`BT_MESH_MODEL_ID_LIGHT_HSL_HUE_SRV`
: Light HSL Hue Server.

`BT_MESH_MODEL_ID_LIGHT_HSL_SAT_SRV`
: Light HSL Saturation Server.

`BT_MESH_MODEL_ID_LIGHT_XYL_SRV`
: Light xyL Server.

`BT_MESH_MODEL_ID_LIGHT_XYL_SETUP_SRV`
: Light xyL Setup Server.

`BT_MESH_MODEL_ID_LIGHT_XYL_CLI`
: Light xyL Client.

`BT_MESH_MODEL_ID_LIGHT_LC_SRV`
: Light LC Server.

`BT_MESH_MODEL_ID_LIGHT_LC_SETUPSRV`
: Light LC Setup Server.

`BT_MESH_MODEL_ID_LIGHT_LC_CLI`
: Light LC Client.

Models from the Mesh Binary Large Object Transfer Model Specification

`BT_MESH_MODEL_ID_BLOB_SRV`
: BLOB Transfer Server.

`BT_MESH_MODEL_ID_BLOB_CLI`
: BLOB Transfer Client.

Models from the Mesh Device Firmware Update Model Specification

`BT_MESH_MODEL_ID_DFU_SRV`
: Firmware Update Server.

`BT_MESH_MODEL_ID_DFU_CLI`
: Firmware Update Client.

`BT_MESH_MODEL_ID_DFD_SRV`
: Firmware Distribution Server.

`BT_MESH_MODEL_ID_DFD_CLI`
: Firmware Distribution Client.

Defines

BT_MESH_ADDR_IS_UNICAST(addr)
Check if a Bluetooth Mesh address is a unicast address.

BT_MESH_ADDR_IS_GROUP(addr)
Check if a Bluetooth Mesh address is a group address.

BT_MESH_ADDR_IS_FIXED_GROUP(addr)
Check if a Bluetooth Mesh address is a fixed group address.

BT_MESH_ADDR_IS_VIRTUAL(addr)
Check if a Bluetooth Mesh address is a virtual address.

BT_MESH_ADDR_IS_RFU(addr)
Check if a Bluetooth Mesh address is an RFU address.

BT_MESH_IS_DEV_KEY(key)
Check if a Bluetooth Mesh key is a device key.

BT_MESH_APP_SEG_SDU_MAX
Maximum size of an access message segment (in octets).

BT_MESH_APP_UNSEG_SDU_MAX
Maximum payload size of an unsegmented access message (in octets).

BT_MESH_RX_SEG_MAX
Maximum number of segments supported for incoming messages.

BT_MESH_TX_SEG_MAX
Maximum number of segments supported for outgoing messages.

BT_MESH_TX_SDU_MAX
Maximum possible payload size of an outgoing access message (in octets).

BT_MESH_RX_SDU_MAX
Maximum possible payload size of an incoming access message (in octets).

BT_MESH_ELEM(_loc, _mods, _vnd_mods)
Helper to define a mesh element within an array.

In case the element has no SIG or Vendor models the helper macro BT_MESH_MODEL_NONE can be given instead.

Parameters

- **_loc** – Location Descriptor.
- **_mods** – Array of models.
- **_vnd_mods** – Array of vendor models.

BT_MESH_MODEL_OP_1(b0)

BT_MESH_MODEL_OP_2(b0, b1)

BT_MESH_MODEL_OP_3(b0, cid)

BT_MESH_LEN_EXACT(len)

Macro for encoding exact message length for fixed-length messages.

BT_MESH_LEN_MIN(len)

Macro for encoding minimum message length for variable-length messages.

BT_MESH_MODEL_OP_END

End of the opcode list.

Must always be present.

BT_MESH_MODEL_NO_OPS

Helper to define an empty opcode list.

This macro uses compound literal feature of C99 standard and thus is available only from C, not C++.

BT_MESH_MODEL_NONE

Helper to define an empty model array.

This macro uses compound literal feature of C99 standard and thus is available only from C, not C++.

BT_MESH_MODEL_CNT_CB(_id, _op, _pub, _user_data, _keys, _grps, _cb)

Composition data SIG model entry with callback functions with specific number of keys & groups.

This macro uses compound literal feature of C99 standard and thus is available only from C, not C++.

Parameters

- **_id** – Model ID.
- **_op** – Array of model opcode handlers.
- **_pub** – Model publish parameters.
- **_user_data** – User data for the model.
- **_keys** – Number of keys that can be bound to the model. Shall not exceed CONFIG_BT_MESH_MODEL_KEY_COUNT .
- **_grps** – Number of addresses that the model can be subscribed to. Shall not exceed CONFIG_BT_MESH_MODEL_GROUP_COUNT .
- **_cb** – Callback structure, or NULL to keep no callbacks.

BT_MESH_MODEL_CNT_VND_CB(_company, _id, _op, _pub, _user_data, _keys, _grps, _cb)

Composition data vendor model entry with callback functions with specific number of keys & groups.

This macro uses compound literal feature of C99 standard and thus is available only from C, not C++.

Parameters

- **_company** – Company ID.
- **_id** – Model ID.
- **_op** – Array of model opcode handlers.
- **_pub** – Model publish parameters.
- **_user_data** – User data for the model.

- **_keys** – Number of keys that can be bound to the model. Shall not exceed CONFIG_BT_MESH_MODEL_KEY_COUNT .
- **_grps** – Number of addresses that the model can be subscribed to. Shall not exceed CONFIG_BT_MESH_MODEL_GROUP_COUNT .
- **_cb** – Callback structure, or NULL to keep no callbacks.

BT_MESH_MODEL_CB(_id, _op, _pub, _user_data, _cb)

Composition data SIG model entry with callback functions.

This macro uses compound literal feature of C99 standard and thus is available only from C, not C++.

Parameters

- **_id** – Model ID.
- **_op** – Array of model opcode handlers.
- **_pub** – Model publish parameters.
- **_user_data** – User data for the model.
- **_cb** – Callback structure, or NULL to keep no callbacks.

BT_MESH_MODEL_METADATA_CB(_id, _op, _pub, _user_data, _cb, _metadata)

Composition data SIG model entry with callback functions and metadata.

This macro uses compound literal feature of C99 standard and thus is available only from C, not C++.

Parameters

- **_id** – Model ID.
- **_op** – Array of model opcode handlers.
- **_pub** – Model publish parameters.
- **_user_data** – User data for the model.
- **_cb** – Callback structure, or NULL to keep no callbacks.
- **_metadata** – Metadata structure. Used if CONFIG_BT_MESH_LARGE_COMP_DATA_SRV is enabled.

BT_MESH_MODEL_VND_CB(_company, _id, _op, _pub, _user_data, _cb)

Composition data vendor model entry with callback functions.

This macro uses compound literal feature of C99 standard and thus is available only from C, not C++.

Parameters

- **_company** – Company ID.
- **_id** – Model ID.
- **_op** – Array of model opcode handlers.
- **_pub** – Model publish parameters.
- **_user_data** – User data for the model.
- **_cb** – Callback structure, or NULL to keep no callbacks.

BT_MESH_MODEL_VND_METADATA_CB(_company, _id, _op, _pub, _user_data, _cb, _metadata)

Composition data vendor model entry with callback functions and metadata.

This macro uses compound literal feature of C99 standard and thus is available only from C, not C++.

Parameters

- **_company** – Company ID.
- **_id** – Model ID.
- **_op** – Array of model opcode handlers.
- **_pub** – Model publish parameters.
- **_user_data** – User data for the model.
- **_cb** – Callback structure, or NULL to keep no callbacks.
- **_metadata** – Metadata structure. Used if CONFIG_BT_MESH_LARGE_COMP_DATA_SRV is enabled.

BT_MESH_MODEL(_id, _op, _pub, _user_data)

Composition data SIG model entry.

This macro uses compound literal feature of C99 standard and thus is available only from C, not C++.

Parameters

- **_id** – Model ID.
- **_op** – Array of model opcode handlers.
- **_pub** – Model publish parameters.
- **_user_data** – User data for the model.

BT_MESH_MODEL_VND(_company, _id, _op, _pub, _user_data)

Composition data vendor model entry.

This macro uses compound literal feature of C99 standard and thus is available only from C, not C++.

Parameters

- **_company** – Company ID.
- **_id** – Model ID.
- **_op** – Array of model opcode handlers.
- **_pub** – Model publish parameters.
- **_user_data** – User data for the model.

BT_MESH_TRANSMIT(count, int_ms)

Encode transmission count & interval steps.

Parameters

- **count** – Number of retransmissions (first transmission is excluded).
- **int_ms** – Interval steps in milliseconds. Must be greater than 0, less than or equal to 320, and a multiple of 10.

Returns

Mesh transmit value that can be used e.g. for the default values of the configuration model data.

BT_MESH_TRANSMIT_COUNT(transmit)

Decode transmit count from a transmit value.

Parameters

- **transmit** – Encoded transmit count & interval value.

Returns
Transmission count (actual transmissions is N + 1).

`BT_MESH_TRANSMIT_INT`(transmit)
Decode transmit interval from a transmit value.

Parameters

- `transmit` – Encoded transmit count & interval value.

Returns
Transmission interval in milliseconds.

`BT_MESH_PUB_TRANSMIT`(count, int_ms)
Encode Publish Retransmit count & interval steps.

Parameters

- `count` – Number of retransmissions (first transmission is excluded).
- `int_ms` – Interval steps in milliseconds. Must be greater than 0 and a multiple of 50.

Returns
Mesh transmit value that can be used e.g. for the default values of the configuration model data.

`BT_MESH_PUB_TRANSMIT_COUNT`(transmit)
Decode Publish Retransmit count from a given value.

Parameters

- `transmit` – Encoded Publish Retransmit count & interval value.

Returns
Retransmission count (actual transmissions is N + 1).

`BT_MESH_PUB_TRANSMIT_INT`(transmit)
Decode Publish Retransmit interval from a given value.

Parameters

- `transmit` – Encoded Publish Retransmit count & interval value.

Returns
Transmission interval in milliseconds.

`BT_MESH_PUB_MSG_TOTAL`(pub)
Get total number of messages within one publication interval including initial publication.

Parameters

- `pub` – Model publication context.

Returns
total number of messages.

`BT_MESH_PUB_MSG_NUM`(pub)
Get message number within one publication interval.

Meant to be used inside *bt_mesh_model_pub::update*.

Parameters

- `pub` – Model publication context.

Returns
message number starting from 1.

BT_MESH_MODEL_PUB_DEFINE(_name, _update, _msg_len)

Define a model publication context.

Parameters

- **_name** – Variable name given to the context.
- **_update** – Optional message update callback (may be NULL).
- **_msg_len** – Length of the publication message.

BT_MESH_MODELS_METADATA_ENTRY(_len, _id, _data)

Initialize a Models Metadata entry structure in a list.

Parameters

- **_len** – Length of the metadata entry.
- **_id** – ID of the Models Metadata entry.
- **_data** – Pointer to a contiguous memory that contains the metadata.

BT_MESH_MODELS_METADATA_NONE

Helper to define an empty Models metadata array.

BT_MESH_MODELS_METADATA_END

End of the Models Metadata list.

Must always be present.

BT_MESH_TTL_DEFAULT

Special TTL value to request using configured default TTL.

BT_MESH_TTL_MAX

Maximum allowed TTL value.

Functions

int **bt_mesh_model_send**(const struct *bt_mesh_model* *model, struct *bt_mesh_msg_ctx* *ctx, struct *net_buf_simple* *msg, const struct *bt_mesh_send_cb* *cb, void *cb_data)

Send an Access Layer message.

Parameters

- **model** – Mesh (client) Model that the message belongs to.
- **ctx** – Message context, includes keys, TTL, etc.
- **msg** – Access Layer payload (the actual message to be sent).
- **cb** – Optional "message sent" callback.
- **cb_data** – User data to be passed to the callback.

Returns

0 on success, or (negative) error code on failure.

int **bt_mesh_model_publish**(const struct *bt_mesh_model* *model)

Send a model publication message.

Before calling this function, the user needs to ensure that the model publication message (*bt_mesh_model_pub::msg*) contains a valid message to be sent. Note that this API

is only to be used for non-period publishing. For periodic publishing the app only needs to make sure that *bt_mesh_model_pub::msg* contains a valid message whenever the *bt_mesh_model_pub::update* callback is called.

Parameters

- **model** – Mesh (client) Model that's publishing the message.

Returns
0 on success, or (negative) error code on failure.

static inline bool **bt_mesh_model_pub_is_retransmission**(const struct *bt_mesh_model* *model)

Check if a message is being retransmitted.

Meant to be used inside the *bt_mesh_model_pub::update* callback.

Parameters

- **model** – Mesh Model that supports publication.

Returns
true if this is a retransmission, false if this is a first publication.

const struct *bt_mesh_elem* ***bt_mesh_model_elem**(const struct *bt_mesh_model* *mod)

Get the element that a model belongs to.

Parameters

- **mod** – Mesh model.

Returns
Pointer to the element that the given model belongs to.

const struct *bt_mesh_model* ***bt_mesh_model_find**(const struct *bt_mesh_elem* *elem, uint16_t id)

Find a SIG model.

Parameters

- **elem** – Element to search for the model in.
- **id** – Model ID of the model.

Returns
A pointer to the Mesh model matching the given parameters, or NULL if no SIG model with the given ID exists in the given element.

const struct *bt_mesh_model* ***bt_mesh_model_find_vnd**(const struct *bt_mesh_elem* *elem, uint16_t company, uint16_t id)

Find a vendor model.

Parameters

- **elem** – Element to search for the model in.
- **company** – Company ID of the model.
- **id** – Model ID of the model.

Returns
A pointer to the Mesh model matching the given parameters, or NULL if no vendor model with the given ID exists in the given element.

static inline bool **bt_mesh_model_in_primary**(const struct *bt_mesh_model* *mod)

Get whether the model is in the primary element of the device.

Parameters

- **mod** – Mesh model.

Returns
true if the model is on the primary element, false otherwise.

int **bt_mesh_model_data_store**(const struct *bt_mesh_model* *mod, bool vnd, const char *name, const void *data, size_t data_len)

Immediately store the model's user data in persistent storage.

Parameters

- **mod** – Mesh model.
- **vnd** – This is a vendor model.
- **name** – Name/key of the settings item. Only *SETTINGS_MAX_DIR_DEPTH* bytes will be used at most.
- **data** – Model data to store, or NULL to delete any model data.
- **data_len** – Length of the model data.

Returns
0 on success, or (negative) error code on failure.

void **bt_mesh_model_data_store_schedule**(const struct *bt_mesh_model* *mod)

Schedule the model's user data store in persistent storage.

This function triggers the *bt_mesh_model_cb::pending_store* callback for the corresponding model after delay defined by CONFIG_BT_MESH_STORE_TIMEOUT .

The delay is global for all models. Once scheduled, the callback can not be re-scheduled until previous schedule completes.

Parameters

- **mod** – Mesh model.

int **bt_mesh_model_extend**(const struct *bt_mesh_model* *extending_mod, const struct *bt_mesh_model* *base_mod)

Let a model extend another.

Mesh models may be extended to reuse their functionality, forming a more complex model. A Mesh model may extend any number of models, in any element. The extensions may also be nested, ie a model that extends another may itself be extended.

A set of models that extend each other form a model extension list.

All models in an extension list share one subscription list per element. The access layer will utilize the combined subscription list of all models in an extension list and element, giving the models extended subscription list capacity.

If CONFIG_BT_MESH_COMP_PAGE_1 is enabled, it is not allowed to call this function before the *bt_mesh_model_cb::init* callback is called for both models, except if it is called as part of the final callback.

Parameters

- **extending_mod** – Mesh model that is extending the base model.
- **base_mod** – The model being extended.

Return values
0 – Successfully extended the base_mod model.

int **bt_mesh_model_correspond**(const struct *bt_mesh_model* *corresponding_mod, const struct *bt_mesh_model* *base_mod)

Let a model correspond to another.

Mesh models may correspond to each other, which means that if one is present, other must be present too. A Mesh model may correspond to any number of models, in any

element. All models connected together via correspondence form single Correspondence Group, which has it's unique Correspondence ID. Information about Correspondence is used to construct Composition Data Page 1.

This function must be called on already initialized base_mod. Because this function is designed to be called in corresponding_mod initializer, this means that base_mod shall be initialized before corresponding_mod is.

Parameters

- `corresponding_mod` – Mesh model that is corresponding to the base model.
- `base_mod` – The model being corresponded to.

Return values

- `0` – Successfully saved correspondence to the base_mod model.
- `-ENOMEM` – There is no more space to save this relation.
- `-ENOTSUP` – Composition Data Page 1 is not supported.

bool `bt_mesh_model_is_extended`(const struct *bt_mesh_model* *model)

Check if model is extended by another model.

Parameters

- `model` – The model to check.

Return values

`true` – If model is extended by another model, otherwise false

int `bt_mesh_comp_change_prepare`(void)

Indicate that the composition data will change on next bootup.

Tell the config server that the composition data is expected to change on the next bootup, and the current composition data should be backed up.

Returns

Zero on success or (negative) error code otherwise.

int `bt_mesh_models_metadata_change_prepare`(void)

Indicate that the metadata will change on next bootup.

Tell the config server that the models metadata is expected to change on the next bootup, and the current models metadata should be backed up.

Returns

Zero on success or (negative) error code otherwise.

int `bt_mesh_comp2_register`(const struct *bt_mesh_comp2* *comp2)

Register composition data page 2 of the device.

Register Mesh Profiles information (Ref section 3.12 in Bluetooth SIG Assigned Numbers) for composition data page 2 of the device.

Note

There must be at least one record present in `comp2`

Parameters

- `comp2` – Pointer to composition data page 2.

Returns

Zero on success or (negative) error code otherwise.

struct **bt_mesh_elem**

#include <access.h> Abstraction that describes a Mesh Element.

Public Members

const uint16_t **loc**

Location Descriptor (GATT Bluetooth Namespace Descriptors)

const uint8_t **model_count**

The number of SIG models in this element.

const uint8_t **vnd_model_count**

The number of vendor models in this element.

const struct *bt_mesh_model* *const **models**

The list of SIG models in this element.

const struct *bt_mesh_model* *const **vnd_models**

The list of vendor models in this element.

struct **bt_mesh_elem_rt_ctx**

#include <access.h> Mesh Element runtime information.

Public Members

uint16_t **addr**

Unicast Address.

Set at runtime during provisioning.

struct **bt_mesh_model_op**

#include <access.h> Model opcode handler.

Public Members

const uint32_t **opcode**

OpCode encoded using the BT_MESH_MODEL_OP_* macros.

const ssize_t **len**

Message length.

If the message has variable length then this value indicates minimum message length and should be positive. Handler function should verify precise length based on the contents of the message. If the message has fixed length then this value should be negative. Use BT_MESH_LEN_* macros when defining this value.

int (*const **func**)(const struct *bt_mesh_model* *model, struct *bt_mesh_msg_ctx* *ctx, struct *net_buf_simple* *buf)

Handler function for this opcode.

Param model
Model instance receiving the message.

Param ctx
Message context for the message.

Param buf
Message buffer containing the message payload, not including the opcode.

Return
Zero on success or (negative) error code otherwise.

struct **bt_mesh_model_pub**

#include <access.h> Model publication context.

The context should primarily be created using the BT_MESH_MODEL_PUB_DEFINE macro.

Public Members

const struct *bt_mesh_model* ***mod**

The model the context belongs to.

Initialized by the stack.

uint16_t **addr**

Publish Address.

const uint8_t ***uuid**

Label UUID if Publish Address is Virtual Address.

uint16_t **key**

Publish AppKey Index.

uint16_t **cred**

Friendship Credentials Flag.

uint16_t **send_rel**

Force reliable sending (segment acks)

uint16_t **fast_period**

Use FastPeriodDivisor.

uint16_t **retr_update**

Call update callback on every retransmission.

uint8_t **ttl**

Publish Time to Live.

uint8_t **retransmit**

Retransmit Count & Interval Steps.

uint8_t **period**

Publish Period.

uint8_t **period_div**

Divisor for the Period.

uint8_t **count**

Transmissions left.

uint8_t **delayable**

Use random delay for publishing.

uint32_t **period_start**

Start of the current period.

struct *net_buf_simple* ***msg**

Publication buffer, containing the publication message.

This will get correctly created when the publication context has been defined using the BT_MESH_MODEL_PUB_DEFINE macro.

```
BT_MESH_MODEL_PUB_DEFINE(name, update, size);
```

int (***update**)(const struct *bt_mesh_model* *mod)

Callback for updating the publication buffer.

When set to NULL, the model is assumed not to support periodic publishing. When set to non-NULL the callback will be called periodically and is expected to update *bt_mesh_model_pub::msg* with a valid publication message.

If the callback returns non-zero, the publication is skipped and will resume on the next periodic publishing interval.

When *bt_mesh_model_pub::retr_update* is set to 1, the callback will be called on every retransmission.

Param mod
The Model the Publication Context belongs to.

Return
Zero on success or (negative) error code otherwise.

struct *k_work_delayable* **timer**

Publish Period Timer.

Only for stack-internal use.

struct **bt_mesh_models_metadata_entry**

#include <*access.h*> Models Metadata Entry struct.

The struct should primarily be created using the BT_MESH_MODELS_METADATA_ENTRY macro.

struct **bt_mesh_model_cb**

#include <*access.h*> Model callback functions.

Public Members

int (*const **settings_set**)(const struct *bt_mesh_model* *model, const char *name, size_t len_rd, *settings_read_cb* read_cb, void *cb_arg)

Set value handler of user data tied to the model.

See also

settings_handler::h_set

Param model
Model to set the persistent data of.
Param name
Name/key of the settings item.
Param len_rd
The size of the data found in the backend.
Param read_cb
Function provided to read the data from the backend.
Param cb_arg
Arguments for the read function provided by the backend.
Return
0 on success, error otherwise.

int (*const **start**)(const struct *bt_mesh_model* *model)

Callback called when the mesh is started.

This handler gets called after the node has been provisioned, or after all mesh data has been loaded from persistent storage.

When this callback fires, the mesh model may start its behavior, and all Access APIs are ready for use.

Param model
Model this callback belongs to.
Return
0 on success, error otherwise.

int (*const **init**)(const struct *bt_mesh_model* *model)

Model init callback.

Called on every model instance during mesh initialization.

If any of the model init callbacks return an error, the Mesh subsystem initialization will be aborted, and the error will be returned to the caller of *bt_mesh_init*.

Param model
Model to be initialized.
Return
0 on success, error otherwise.

void (*const **reset**)(const struct *bt_mesh_model* *model)

Model reset callback.

Called when the mesh node is reset. All model data is deleted on reset, and the model should clear its state.

> **Note**
>
> If the model stores any persistent data, this needs to be erased manually.

Param model
Model this callback belongs to.

void (*const **pending_store**)(const struct *bt_mesh_model* *model)

Callback used to store pending model's user data.

Triggered by *bt_mesh_model_data_store_schedule*.

To store the user data, call *bt_mesh_model_data_store*.

Param model
Model this callback belongs to.

struct **bt_mesh_mod_id_vnd**

#include <*access.h*> Vendor model ID.

Public Members

uint16_t **company**

Vendor's company ID.

uint16_t **id**

Model ID.

struct **bt_mesh_model**

#include <*access.h*> Abstraction that describes a Mesh Model instance.

Public Members

const uint16_t **id**

SIG model ID.

const struct *bt_mesh_mod_id_vnd* **vnd**

Vendor model ID.

struct *bt_mesh_model_pub* *const **pub**

Model Publication.

uint16_t *const **keys**

AppKey List.

uint16_t *const **groups**

Subscription List (group or virtual addresses)

const uint8_t **const **uuids**

List of Label UUIDs the model is subscribed to.

const struct *bt_mesh_model_op* *const **op**

Opcode handler list.

const struct *bt_mesh_model_cb* *const **cb**

Model callback structure.

struct **bt_mesh_model_rt_ctx**

#include <access.h>

Public Members

void ***user_data**

Model-specific user data.

struct **bt_mesh_send_cb**

#include <access.h> Callback structure for monitoring model message sending.

Public Members

void (***start**)(uint16_t duration, int err, void *cb_data)

Handler called at the start of the transmission.

Param duration
The duration of the full transmission.

Param err
Error occurring during sending.

Param cb_data
Callback data, as passed to the send API.

void (***end**)(int err, void *cb_data)

Handler called at the end of the transmission.

Param err
Error occurring during sending.

Param cb_data
Callback data, as passed to the send API.

struct **bt_mesh_comp**

#include <access.h> Node Composition.

Public Members

uint16_t **cid**

Company ID.

uint16_t **pid**

Product ID.

uint16_t **vid**
Version ID.

size_t **elem_count**
The number of elements in this device.

const struct *bt_mesh_elem* ***elem**
List of elements.

struct **bt_mesh_comp2_record**
#include <*access.h*> Composition data page 2 record.

Public Members

uint16_t **id**
Mesh profile ID.

uint8_t **x**
Major version.

uint8_t **y**
Minor version.

uint8_t **z**
Z version.

struct *bt_mesh_comp2_record* **version**
Mesh Profile Version.

uint8_t **elem_offset_cnt**
Element offset count.

const uint8_t ***elem_offset**
Element offset list.

uint16_t **data_len**
Length of additional data.

const void ***data**
Additional data.

struct **bt_mesh_comp2**
#include <*access.h*> Node Composition data page 2.

Public Members

size_t **record_cnt**

The number of Mesh Profile records on a device.

const struct *bt_mesh_comp2_record* ***record**

List of records.

Mesh models

Foundation models The Bluetooth Mesh specification defines foundation models that can be used by network administrators to configure and diagnose mesh nodes.

Configuration Client The Configuration Client model is a foundation model defined by the Bluetooth Mesh specification. It provides functionality for configuring most parameters of a mesh node, including encryption keys, model configuration and feature enabling.

The Configuration Client model communicates with a *Configuration Server* model using the device key of the target node. The Configuration Client model may communicate with servers on other nodes or self-configure through the local Configuration Server model.

All configuration functions in the Configuration Client API have `net_idx` and `addr` as their first parameters. These should be set to the network index and primary unicast address that the target node was provisioned with.

The Configuration Client model is optional, and it must only be instantiated on the primary element if present in the Composition Data.

API reference

group **bt_mesh_cfg_cli**

Configuration Client Model.

Defines

BT_MESH_MODEL_CFG_CLI(cli_data)

Generic Configuration Client model composition data entry.

Parameters

- **cli_data** – Pointer to a *Configuration Client Model* instance.

BT_MESH_PUB_PERIOD_100MS(steps)

Helper macro to encode model publication period in units of 100ms.

Parameters

- **steps** – Number of 100ms steps.

Returns

Encoded value that can be assigned to *bt_mesh_cfg_cli_mod_pub.period*

BT_MESH_PUB_PERIOD_SEC(steps)

Helper macro to encode model publication period in units of 1 second.

Parameters

- **steps** – Number of 1 second steps.

Returns
Encoded value that can be assigned to *bt_mesh_cfg_cli_mod_pub.period*

`BT_MESH_PUB_PERIOD_10SEC`(steps)
Helper macro to encode model publication period in units of 10 seconds.

Parameters

- **steps** – Number of 10 second steps.

Returns
Encoded value that can be assigned to *bt_mesh_cfg_cli_mod_pub.period*

`BT_MESH_PUB_PERIOD_10MIN`(steps)
Helper macro to encode model publication period in units of 10 minutes.

Parameters

- **steps** – Number of 10 minute steps.

Returns
Encoded value that can be assigned to *bt_mesh_cfg_cli_mod_pub.period*

Functions

int **bt_mesh_cfg_cli_node_reset**(uint16_t net_idx, uint16_t addr, bool *status)
Reset the target node and remove it from the network.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **status** – Status response parameter

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_comp_data_get**(uint16_t net_idx, uint16_t addr, uint8_t page, uint8_t *rsp, struct *net_buf_simple* *comp)
Get the target node's composition data.

If the other device does not have the given composition data page, it will return the largest page number it supports that is less than the requested page index. The actual page the device responds with is returned in `rsp`.

This method can be used asynchronously by setting `rsp` and `comp` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **page** – Composition data page, or 0xff to request the first available page.
- **rsp** – Return parameter for the returned page number, or NULL.
- **comp** – Composition data buffer to fill.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_beacon_get**(uint16_t net_idx, uint16_t addr, uint8_t *status)

Get the target node's network beacon state.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **status** – Status response parameter, returns one of *BT_MESH_BEACON_DISABLED* or *BT_MESH_BEACON_ENABLED* on success.

Returns

0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_krp_get**(uint16_t net_idx, uint16_t addr, uint16_t key_net_idx, uint8_t *status, uint8_t *phase)

Get the target node's network key refresh phase state.

This method can be used asynchronously by setting `status` and `phase` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **key_net_idx** – Network key index.
- **status** – Status response parameter.
- **phase** – Pointer to the Key Refresh variable to fill.

Returns

0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_krp_set**(uint16_t net_idx, uint16_t addr, uint16_t key_net_idx, uint8_t transition, uint8_t *status, uint8_t *phase)

Set the target node's network key refresh phase parameters.

This method can be used asynchronously by setting `status` and `phase` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **key_net_idx** – Network key index.
- **transition** – Transition parameter.
- **status** – Status response parameter.
- **phase** – Pointer to the new Key Refresh phase. Will return the actual Key Refresh phase after updating.

Returns

0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_beacon_set**(uint16_t net_idx, uint16_t addr, uint8_t val, uint8_t *status)

Set the target node's network beacon state.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **val** – New network beacon state, should be one of *BT_MESH_BEACON_DISABLED* or *BT_MESH_BEACON_ENABLED*.
- **status** – Status response parameter. Returns one of *BT_MESH_BEACON_DISABLED* or *BT_MESH_BEACON_ENABLED* on success.

Returns

0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_ttl_get**(uint16_t net_idx, uint16_t addr, uint8_t *ttl)

Get the target node's Time To Live value.

This method can be used asynchronously by setting `ttl` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **ttl** – TTL response buffer.

Returns

0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_ttl_set**(uint16_t net_idx, uint16_t addr, uint8_t val, uint8_t *ttl)

Set the target node's Time To Live value.

This method can be used asynchronously by setting `ttl` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **val** – New Time To Live value.
- **ttl** – TTL response buffer.

Returns

0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_friend_get**(uint16_t net_idx, uint16_t addr, uint8_t *status)

Get the target node's Friend feature status.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.

- **addr** – Target node address.
- **status** – Status response parameter. Returns one of *BT_MESH_FRIEND_DISABLED*, *BT_MESH_FRIEND_ENABLED* or *BT_MESH_FRIEND_NOT_SUPPORTED* on success.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_friend_set**(uint16_t net_idx, uint16_t addr, uint8_t val, uint8_t *status)

Set the target node's Friend feature state.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **val** – New Friend feature state. Should be one of *BT_MESH_FRIEND_DISABLED* or *BT_MESH_FRIEND_ENABLED*.
- **status** – Status response parameter. Returns one of *BT_MESH_FRIEND_DISABLED*, *BT_MESH_FRIEND_ENABLED* or *BT_MESH_FRIEND_NOT_SUPPORTED* on success.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_gatt_proxy_get**(uint16_t net_idx, uint16_t addr, uint8_t *status)

Get the target node's Proxy feature state.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **status** – Status response parameter. Returns one of *BT_MESH_GATT_PROXY_DISABLED*, *BT_MESH_GATT_PROXY_ENABLED* or *BT_MESH_GATT_PROXY_NOT_SUPPORTED* on success.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_gatt_proxy_set**(uint16_t net_idx, uint16_t addr, uint8_t val, uint8_t *status)

Set the target node's Proxy feature state.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.

- `val` – New Proxy feature state. Must be one of *BT_MESH_GATT_PROXY_DISABLED* or *BT_MESH_GATT_PROXY_ENABLED*.
- `status` – Status response parameter. Returns one of *BT_MESH_GATT_PROXY_DISABLED*, *BT_MESH_GATT_PROXY_ENABLED* or *BT_MESH_GATT_PROXY_NOT_SUPPORTED* on success.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_net_transmit_get**(uint16_t net_idx, uint16_t addr, uint8_t *transmit)

Get the target node's network_transmit state.

This method can be used asynchronously by setting `transmit` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `transmit` – Network transmit response parameter. Returns the encoded network transmission parameters on success. Decoded with *BT_MESH_TRANSMIT_COUNT* and *BT_MESH_TRANSMIT_INT*.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_net_transmit_set**(uint16_t net_idx, uint16_t addr, uint8_t val, uint8_t *transmit)

Set the target node's network transmit parameters.

This method can be used asynchronously by setting `transmit` as NULL. This way the method will not wait for response and will return immediately after sending the command.

See also

BT_MESH_TRANSMIT.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `val` – New encoded network transmit parameters.
- `transmit` – Network transmit response parameter. Returns the encoded network transmission parameters on success. Decoded with *BT_MESH_TRANSMIT_COUNT* and *BT_MESH_TRANSMIT_INT*.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_relay_get**(uint16_t net_idx, uint16_t addr, uint8_t *status, uint8_t *transmit)

Get the target node's Relay feature state.

This method can be used asynchronously by setting `status` and `transmit` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **status** – Status response parameter. Returns one of *BT_MESH_RELAY_DISABLED*, *BT_MESH_RELAY_ENABLED* or *BT_MESH_RELAY_NOT_SUPPORTED* on success.
- **transmit** – Transmit response parameter. Returns the encoded relay transmission parameters on success. Decoded with *BT_MESH_TRANSMIT_COUNT* and *BT_MESH_TRANSMIT_INT*.

Returns

0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_relay_set**(uint16_t net_idx, uint16_t addr, uint8_t new_relay, uint8_t new_transmit, uint8_t *status, uint8_t *transmit)

Set the target node's Relay parameters.

This method can be used asynchronously by setting `status` and `transmit` as NULL. This way the method will not wait for response and will return immediately after sending the command.

See also

BT_MESH_TRANSMIT.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **new_relay** – New relay state. Must be one of *BT_MESH_RELAY_DISABLED* or *BT_MESH_RELAY_ENABLED*.
- **new_transmit** – New encoded relay transmit parameters.
- **status** – Status response parameter. Returns one of *BT_MESH_RELAY_DISABLED*, *BT_MESH_RELAY_ENABLED* or *BT_MESH_RELAY_NOT_SUPPORTED* on success.
- **transmit** – Transmit response parameter. Returns the encoded relay transmission parameters on success. Decoded with *BT_MESH_TRANSMIT_COUNT* and *BT_MESH_TRANSMIT_INT*.

Returns

0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_net_key_add**(uint16_t net_idx, uint16_t addr, uint16_t key_net_idx, const uint8_t net_key[16], uint8_t *status)

Add a network key to the target node.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `key_net_idx` – Network key index.
- `net_key` – Network key.
- `status` – Status response parameter.

Returns
0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_cli_net_key_get`(uint16_t net_idx, uint16_t addr, uint16_t *keys, size_t *key_cnt)

Get a list of the target node's network key indexes.

This method can be used asynchronously by setting `keys` or `key_cnt` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `keys` – Net key index list response parameter. Will be filled with all the returned network key indexes it can fill.
- `key_cnt` – Net key index list length. Should be set to the capacity of the `keys` list when calling. Will return the number of returned network key indexes upon success.

Returns
0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_cli_net_key_del`(uint16_t net_idx, uint16_t addr, uint16_t key_net_idx, uint8_t *status)

Delete a network key from the target node.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `key_net_idx` – Network key index.
- `status` – Status response parameter.

Returns
0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_cli_app_key_add`(uint16_t net_idx, uint16_t addr, uint16_t key_net_idx, uint16_t key_app_idx, const uint8_t app_key[16], uint8_t *status)

Add an application key to the target node.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `key_net_idx` – Network key index the application key belongs to.
- `key_app_idx` – Application key index.
- `app_key` – Application key.
- `status` – Status response parameter.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_app_key_get**(uint16_t net_idx, uint16_t addr, uint16_t key_net_idx, uint8_t *status, uint16_t *keys, size_t *key_cnt)

Get a list of the target node's application key indexes for a specific network key.

This method can be used asynchronously by setting `status` and (`keys` or `key_cnt`) as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `key_net_idx` – Network key index to request the app key indexes of.
- `status` – Status response parameter.
- `keys` – App key index list response parameter. Will be filled with all the returned application key indexes it can fill.
- `key_cnt` – App key index list length. Should be set to the capacity of the `keys` list when calling. Will return the number of returned application key indexes upon success.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_app_key_del**(uint16_t net_idx, uint16_t addr, uint16_t key_net_idx, uint16_t key_app_idx, uint8_t *status)

Delete an application key from the target node.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `key_net_idx` – Network key index the application key belongs to.
- `key_app_idx` – Application key index.
- `status` – Status response parameter.

Returns
0 on success, or (negative) error code on failure.

int bt_mesh_cfg_cli_mod_app_bind(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_app_idx, uint16_t mod_id, uint8_t *status)

Bind an application to a SIG model on the target node.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `elem_addr` – Element address the model is in.
- `mod_app_idx` – Application index to bind.
- `mod_id` – Model ID.
- `status` – Status response parameter.

Returns

0 on success, or (negative) error code on failure.

int bt_mesh_cfg_cli_mod_app_unbind(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_app_idx, uint16_t mod_id, uint8_t *status)

Unbind an application from a SIG model on the target node.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `elem_addr` – Element address the model is in.
- `mod_app_idx` – Application index to unbind.
- `mod_id` – Model ID.
- `status` – Status response parameter.

Returns

0 on success, or (negative) error code on failure.

int bt_mesh_cfg_cli_mod_app_bind_vnd(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_app_idx, uint16_t mod_id, uint16_t cid, uint8_t *status)

Bind an application to a vendor model on the target node.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `elem_addr` – Element address the model is in.
- `mod_app_idx` – Application index to bind.

- **mod_id** – Model ID.
- **cid** – Company ID of the model.
- **status** – Status response parameter.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_mod_app_unbind_vnd**(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_app_idx, uint16_t mod_id, uint16_t cid, uint8_t *status)

Unbind an application from a vendor model on the target node.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **elem_addr** – Element address the model is in.
- **mod_app_idx** – Application index to unbind.
- **mod_id** – Model ID.
- **cid** – Company ID of the model.
- **status** – Status response parameter.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_mod_app_get**(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_id, uint8_t *status, uint16_t *apps, size_t *app_cnt)

Get a list of all applications bound to a SIG model on the target node.

This method can be used asynchronously by setting `status` and (`apps` or `app_cnt`) as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **elem_addr** – Element address the model is in.
- **mod_id** – Model ID.
- **status** – Status response parameter.
- **apps** – App index list response parameter. Will be filled with all the returned application key indexes it can fill.
- **app_cnt** – App index list length. Should be set to the capacity of the `apps` list when calling. Will return the number of returned application key indexes upon success.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_mod_app_get_vnd**(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_id, uint16_t cid, uint8_t *status, uint16_t *apps, size_t *app_cnt)

Get a list of all applications bound to a vendor model on the target node.

This method can be used asynchronously by setting `status` and (`apps` or `app_cnt`) as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **elem_addr** – Element address the model is in.
- **mod_id** – Model ID.
- **cid** – Company ID of the model.
- **status** – Status response parameter.
- **apps** – App index list response parameter. Will be filled with all the returned application key indexes it can fill.
- **app_cnt** – App index list length. Should be set to the capacity of the `apps` list when calling. Will return the number of returned application key indexes upon success.

Returns

0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_mod_pub_get**(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_id, struct *bt_mesh_cfg_cli_mod_pub* *pub, uint8_t *status)

Get publish parameters for a SIG model on the target node.

This method can be used asynchronously by setting `status` and `pub` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **elem_addr** – Element address the model is in.
- **mod_id** – Model ID.
- **pub** – Publication parameter return buffer.
- **status** – Status response parameter.

Returns

0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_mod_pub_get_vnd**(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_id, uint16_t cid, struct *bt_mesh_cfg_cli_mod_pub* *pub, uint8_t *status)

Get publish parameters for a vendor model on the target node.

This method can be used asynchronously by setting `status` and `pub` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **elem_addr** – Element address the model is in.
- **mod_id** – Model ID.
- **cid** – Company ID of the model.
- **pub** – Publication parameter return buffer.
- **status** – Status response parameter.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_mod_pub_set**(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_id, struct *bt_mesh_cfg_cli_mod_pub* *pub, uint8_t *status)

Set publish parameters for a SIG model on the target node.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

`pub` shall not be NULL.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **elem_addr** – Element address the model is in.
- **mod_id** – Model ID.
- **pub** – Publication parameters.
- **status** – Status response parameter.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_mod_pub_set_vnd**(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_id, uint16_t cid, struct *bt_mesh_cfg_cli_mod_pub* *pub, uint8_t *status)

Set publish parameters for a vendor model on the target node.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

`pub` shall not be NULL.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **elem_addr** – Element address the model is in.
- **mod_id** – Model ID.
- **cid** – Company ID of the model.
- **pub** – Publication parameters.
- **status** – Status response parameter.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_mod_sub_add**(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t sub_addr, uint16_t mod_id, uint8_t *status)

Add a group address to a SIG model's subscription list.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **elem_addr** – Element address the model is in.
- **sub_addr** – Group address to add to the subscription list.
- **mod_id** – Model ID.
- **status** – Status response parameter.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_mod_sub_add_vnd**(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t sub_addr, uint16_t mod_id, uint16_t cid, uint8_t *status)

Add a group address to a vendor model's subscription list.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **elem_addr** – Element address the model is in.
- **sub_addr** – Group address to add to the subscription list.
- **mod_id** – Model ID.
- **cid** – Company ID of the model.
- **status** – Status response parameter.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_mod_sub_del**(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t sub_addr, uint16_t mod_id, uint8_t *status)

Delete a group address in a SIG model's subscription list.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **elem_addr** – Element address the model is in.

- **sub_addr** – Group address to add to the subscription list.
- **mod_id** – Model ID.
- **status** – Status response parameter.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_mod_sub_del_vnd**(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t sub_addr, uint16_t mod_id, uint16_t cid, uint8_t *status)

Delete a group address in a vendor model's subscription list.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **elem_addr** – Element address the model is in.
- **sub_addr** – Group address to add to the subscription list.
- **mod_id** – Model ID.
- **cid** – Company ID of the model.
- **status** – Status response parameter.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_mod_sub_overwrite**(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t sub_addr, uint16_t mod_id, uint8_t *status)

Overwrite all addresses in a SIG model's subscription list with a group address.

Deletes all subscriptions in the model's subscription list, and adds a single group address instead.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **elem_addr** – Element address the model is in.
- **sub_addr** – Group address to add to the subscription list.
- **mod_id** – Model ID.
- **status** – Status response parameter.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_mod_sub_overwrite_vnd**(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t sub_addr, uint16_t mod_id, uint16_t cid, uint8_t *status)

Overwrite all addresses in a vendor model's subscription list with a group address.

Deletes all subscriptions in the model's subscription list, and adds a single group address instead.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `elem_addr` – Element address the model is in.
- `sub_addr` – Group address to add to the subscription list.
- `mod_id` – Model ID.
- `cid` – Company ID of the model.
- `status` – Status response parameter.

Returns

0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_mod_sub_va_add**(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, const uint8_t label[16], uint16_t mod_id, uint16_t *virt_addr, uint8_t *status)

Add a virtual address to a SIG model's subscription list.

This method can be used asynchronously by setting `status` and `virt_addr` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `elem_addr` – Element address the model is in.
- `label` – Virtual address label to add to the subscription list.
- `mod_id` – Model ID.
- `virt_addr` – Virtual address response parameter.
- `status` – Status response parameter.

Returns

0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_mod_sub_va_add_vnd**(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, const uint8_t label[16], uint16_t mod_id, uint16_t cid, uint16_t *virt_addr, uint8_t *status)

Add a virtual address to a vendor model's subscription list.

This method can be used asynchronously by setting `status` and `virt_addr` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- `net_idx` – Network index to encrypt with.

- `addr` – Target node address.
- `elem_addr` – Element address the model is in.
- `label` – Virtual address label to add to the subscription list.
- `mod_id` – Model ID.
- `cid` – Company ID of the model.
- `virt_addr` – Virtual address response parameter.
- `status` – Status response parameter.

Returns
0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_cli_mod_sub_va_del`(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, const uint8_t label[16], uint16_t mod_id, uint16_t *virt_addr, uint8_t *status)

Delete a virtual address in a SIG model's subscription list.

This method can be used asynchronously by setting `status` and `virt_addr` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `elem_addr` – Element address the model is in.
- `label` – Virtual address parameter to add to the subscription list.
- `mod_id` – Model ID.
- `virt_addr` – Virtual address response parameter.
- `status` – Status response parameter.

Returns
0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_cli_mod_sub_va_del_vnd`(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, const uint8_t label[16], uint16_t mod_id, uint16_t cid, uint16_t *virt_addr, uint8_t *status)

Delete a virtual address in a vendor model's subscription list.

This method can be used asynchronously by setting `status` and `virt_addr` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `elem_addr` – Element address the model is in.
- `label` – Virtual address label to add to the subscription list.
- `mod_id` – Model ID.
- `cid` – Company ID of the model.
- `virt_addr` – Virtual address response parameter.
- `status` – Status response parameter.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_mod_sub_va_overwrite**(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, const uint8_t label[16], uint16_t mod_id, uint16_t *virt_addr, uint8_t *status)

Overwrite all addresses in a SIG model's subscription list with a virtual address.

Deletes all subscriptions in the model's subscription list, and adds a single group address instead.

This method can be used asynchronously by setting `status` and `virt_addr` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **elem_addr** – Element address the model is in.
- **label** – Virtual address label to add to the subscription list.
- **mod_id** – Model ID.
- **virt_addr** – Virtual address response parameter.
- **status** – Status response parameter.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_mod_sub_va_overwrite_vnd**(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, const uint8_t label[16], uint16_t mod_id, uint16_t cid, uint16_t *virt_addr, uint8_t *status)

Overwrite all addresses in a vendor model's subscription list with a virtual address.

Deletes all subscriptions in the model's subscription list, and adds a single group address instead.

This method can be used asynchronously by setting `status` and `virt_addr` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **elem_addr** – Element address the model is in.
- **label** – Virtual address label to add to the subscription list.
- **mod_id** – Model ID.
- **cid** – Company ID of the model.
- **virt_addr** – Virtual address response parameter.
- **status** – Status response parameter.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_mod_sub_get**(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_id, uint8_t *status, uint16_t *subs, size_t *sub_cnt)

Get the subscription list of a SIG model on the target node.

This method can be used asynchronously by setting `status` and (`subs` or `sub_cnt`) as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **elem_addr** – Element address the model is in.
- **mod_id** – Model ID.
- **status** – Status response parameter.
- **subs** – Subscription list response parameter. Will be filled with all the returned subscriptions it can fill.
- **sub_cnt** – Subscription list element count. Should be set to the capacity of the `subs` list when calling. Will return the number of returned subscriptions upon success.

Returns

0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_mod_sub_get_vnd**(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_id, uint16_t cid, uint8_t *status, uint16_t *subs, size_t *sub_cnt)

Get the subscription list of a vendor model on the target node.

This method can be used asynchronously by setting `status` and (`subs` or `sub_cnt`) as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **elem_addr** – Element address the model is in.
- **mod_id** – Model ID.
- **cid** – Company ID of the model.
- **status** – Status response parameter.
- **subs** – Subscription list response parameter. Will be filled with all the returned subscriptions it can fill.
- **sub_cnt** – Subscription list element count. Should be set to the capacity of the `subs` list when calling. Will return the number of returned subscriptions upon success.

Returns

0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_hb_sub_set**(uint16_t net_idx, uint16_t addr, struct *bt_mesh_cfg_cli_hb_sub* *sub, uint8_t *status)

Set the target node's Heartbeat subscription parameters.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

`sub` shall not be null.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **sub** – New Heartbeat subscription parameters.
- **status** – Status response parameter.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_hb_sub_get**(uint16_t net_idx, uint16_t addr, struct *bt_mesh_cfg_cli_hb_sub* *sub, uint8_t *status)

Get the target node's Heartbeat subscription parameters.

This method can be used asynchronously by setting `status` and `sub` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **sub** – Heartbeat subscription parameter return buffer.
- **status** – Status response parameter.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_hb_pub_set**(uint16_t net_idx, uint16_t addr, const struct *bt_mesh_cfg_cli_hb_pub* *pub, uint8_t *status)

Set the target node's Heartbeat publication parameters.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

`pub` shall not be NULL;

> **Note**
>
> The target node must already have received the specified network key.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **pub** – New Heartbeat publication parameters.
- **status** – Status response parameter.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_hb_pub_get**(uint16_t net_idx, uint16_t addr, struct *bt_mesh_cfg_cli_hb_pub* *pub, uint8_t *status)

Get the target node's Heartbeat publication parameters.

This method can be used asynchronously by setting `status` and `pub` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **pub** – Heartbeat publication parameter return buffer.
- **status** – Status response parameter.

Returns

0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_mod_sub_del_all**(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_id, uint8_t *status)

Delete all group addresses in a SIG model's subscription list.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **elem_addr** – Element address the model is in.
- **mod_id** – Model ID.
- **status** – Status response parameter.

Returns

0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_mod_sub_del_all_vnd**(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_id, uint16_t cid, uint8_t *status)

Delete all group addresses in a vendor model's subscription list.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **elem_addr** – Element address the model is in.
- **mod_id** – Model ID.
- **cid** – Company ID of the model.
- **status** – Status response parameter.

Returns

0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_net_key_update**(uint16_t net_idx, uint16_t addr, uint16_t key_net_idx, const uint8_t net_key[16], uint8_t *status)

Update a network key to the target node.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **key_net_idx** – Network key index.
- **net_key** – Network key.
- **status** – Status response parameter.

Returns

0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_app_key_update**(uint16_t net_idx, uint16_t addr, uint16_t key_net_idx, uint16_t key_app_idx, const uint8_t app_key[16], uint8_t *status)

Update an application key to the target node.

This method can be used asynchronously by setting `status` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **key_net_idx** – Network key index the application key belongs to.
- **key_app_idx** – Application key index.
- **app_key** – Application key.
- **status** – Status response parameter.

Returns

0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_node_identity_set**(uint16_t net_idx, uint16_t addr, uint16_t key_net_idx, uint8_t new_identity, uint8_t *status, uint8_t *identity)

Set the Node Identity parameters.

This method can be used asynchronously by setting `status` and `identity` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **new_identity** – New identity state. Must be one of *BT_MESH_NODE_IDENTITY_STOPPED* or *BT_MESH_NODE_IDENTITY_RUNNING*
- **key_net_idx** – Network key index the application key belongs to.

- **status** – Status response parameter.
- **identity** – Identity response parameter.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_node_identity_get**(uint16_t net_idx, uint16_t addr, uint16_t key_net_idx, uint8_t *status, uint8_t *identity)

Get the Node Identity parameters.

This method can be used asynchronously by setting `status` and `identity` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **key_net_idx** – Network key index the application key belongs to.
- **status** – Status response parameter.
- **identity** – Identity response parameter. Must be one of *BT_MESH_NODE_IDENTITY_STOPPED* or *BT_MESH_NODE_IDENTITY_RUNNING*

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_cfg_cli_lpn_timeout_get**(uint16_t net_idx, uint16_t addr, uint16_t unicast_addr, int32_t *polltimeout)

Get the Low Power Node Polltimeout parameters.

This method can be used asynchronously by setting `polltimeout` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **unicast_addr** – LPN unicast address.
- **polltimeout** – Poll timeout response parameter.

Returns
0 on success, or (negative) error code on failure.

int32_t **bt_mesh_cfg_cli_timeout_get**(void)

Get the current transmission timeout value.

Returns
The configured transmission timeout in milliseconds.

void **bt_mesh_cfg_cli_timeout_set**(int32_t timeout)

Set the transmission timeout value.

Parameters

- **timeout** – The new transmission timeout.

int **bt_mesh_comp_p0_get**(struct *bt_mesh_comp_p0* *comp, struct *net_buf_simple* *buf)

Create a composition data page 0 representation from a buffer.

The composition data page object will take ownership over the buffer, which should not be manipulated directly after this call.

This function can be used in combination with *bt_mesh_cfg_cli_comp_data_get* to read out composition data page 0 from other devices:

```
NET_BUF_SIMPLE_DEFINE(buf, BT_MESH_RX_SDU_MAX);
struct bt_mesh_comp_p0 comp;

err = bt_mesh_cfg_cli_comp_data_get(net_idx, addr, 0, &page, &buf);
if (!err) {
        bt_mesh_comp_p0_get(&comp, &buf);
}
```

Parameters

- **buf** – Network buffer containing composition data.
- **comp** – Composition data structure to fill.

Returns

0 on success, or (negative) error code on failure.

struct *bt_mesh_comp_p0_elem* *__bt_mesh_comp_p0_elem_pull__(const struct *bt_mesh_comp_p0* *comp, struct *bt_mesh_comp_p0_elem* *elem)

Pull a composition data page 0 element from a composition data page 0 instance.

Each call to this function will pull out a new element from the composition data page, until all elements have been pulled.

Parameters

- **comp** – Composition data page
- **elem** – Element to fill.

Returns

A pointer to `elem` on success, or NULL if no more elements could be pulled.

uint16_t **bt_mesh_comp_p0_elem_mod**(struct *bt_mesh_comp_p0_elem* *elem, int idx)

Get a SIG model from the given composition data page 0 element.

Parameters

- **elem** – Element to read the model from.
- **idx** – Index of the SIG model to read.

Returns

The Model ID of the SIG model at the given index, or 0xffff if the index is out of bounds.

struct *bt_mesh_mod_id_vnd* **bt_mesh_comp_p0_elem_mod_vnd**(struct *bt_mesh_comp_p0_elem* *elem, int idx)

Get a vendor model from the given composition data page 0 element.

Parameters

- **elem** – Element to read the model from.

- **idx** – Index of the vendor model to read.

Returns
The model ID of the vendor model at the given index, or {0xffff, 0xffff} if the index is out of bounds.

struct *bt_mesh_comp_p1_elem* ***bt_mesh_comp_p1_elem_pull**(struct *net_buf_simple* *buf, struct *bt_mesh_comp_p1_elem* *elem)

Pull a Composition Data Page 1 Element from a composition data page 1 instance.

Each call to this function will pull out a new element from the composition data page, until all elements have been pulled.

Parameters

- **buf** – Composition data page 1 buffer
- **elem** – Element to fill.

Returns
A pointer to `elem` on success, or NULL if no more elements could be pulled.

struct *bt_mesh_comp_p1_model_item* ***bt_mesh_comp_p1_item_pull**(struct *bt_mesh_comp_p1_elem* *elem, struct *bt_mesh_comp_p1_model_item* *item)

Pull a Composition Data Page 1 Model Item from a Composition Data Page 1 Element.

Each call to this function will pull out a new item from the Composition Data Page 1 Element, until all items have been pulled.

Parameters

- **elem** – Composition data page 1 Element
- **item** – Model Item to fill.

Returns
A pointer to `item` on success, or NULL if no more elements could be pulled.

struct *bt_mesh_comp_p1_ext_item* ***bt_mesh_comp_p1_pull_ext_item**(struct *bt_mesh_comp_p1_model_item* *item, struct *bt_mesh_comp_p1_ext_item* *ext_item)

Pull Extended Model Item contained in Model Item.

Each call to this function will pull out a new element from the Extended Model Item, until all elements have been pulled.

Parameters

- **item** – Model Item to pull Extended Model Items from
- **ext_item** – Extended Model Item to fill

Returns
A pointer to `ext_item` on success, or NULL if item could not be pulled

struct *bt_mesh_comp_p2_record* ***bt_mesh_comp_p2_record_pull**(struct *net_buf_simple* *buf, struct *bt_mesh_comp_p2_record* *record)

Pull a Composition Data Page 2 Record from a composition data page 2 instance.

Each call to this function will pull out a new element from the composition data page, until all elements have been pulled.

Parameters

- `buf` – Composition data page 2 buffer
- `record` – Record to fill.

Returns
A pointer to `record` on success, or NULL if no more elements could be pulled.

int **bt_mesh_key_idx_unpack_list**(struct *net_buf_simple* *buf, uint16_t *dst_arr, size_t *dst_cnt)

Unpack a list of key index entries from a buffer.

On success, `dst_cnt` is set to the amount of unpacked key index entries.

Parameters

- `buf` – Message buffer containing encoded AppKey or NetKey Indexes.
- `dst_arr` – Destination array for the unpacked list.
- `dst_cnt` – Size of the destination array.

Returns
0 on success.

Returns
-EMSGSIZE if dst_arr size is to small to parse full message.

struct **bt_mesh_cfg_cli_cb**

#include <*cfg_cli.h*> Mesh Configuration Client Status messages callback.

Public Members

void (***comp_data**)(struct *bt_mesh_cfg_cli* *cli, uint16_t addr, uint8_t page, struct *net_buf_simple* *buf)

Optional callback for Composition data messages.

Handles received Composition data messages from a server.

> **Note**
>
> For decoding `buf`, please refer to *bt_mesh_comp_p0_get* and *bt_mesh_comp_p1_elem_pull*.

Param cli
Client that received the status message.
Param addr
Address of the sender.
Param page
Composition data page.
Param buf
Composition data buffer.

void (*`mod_pub_status`)(struct *bt_mesh_cfg_cli* *cli, uint16_t addr, uint8_t status, uint16_t elem_addr, uint16_t mod_id, uint16_t cid, struct *bt_mesh_cfg_cli_mod_pub* *pub)

Optional callback for Model Pub status messages.

Handles received Model Pub status messages from a server.

Param cli
Client that received the status message.
Param addr
Address of the sender.
Param status
Status code for the message.
Param elem_addr
Address of the element.
Param mod_id
Model ID.
Param cid
Company ID.
Param pub
Publication configuration parameters.

void (*`mod_sub_status`)(struct *bt_mesh_cfg_cli* *cli, uint16_t addr, uint8_t status, uint16_t elem_addr, uint16_t sub_addr, uint32_t mod_id)

Optional callback for Model Sub Status messages.

Handles received Model Sub Status messages from a server.

Param cli
Client that received the status message.
Param addr
Address of the sender.
Param status
Status Code for requesting message.
Param elem_addr
The unicast address of the element.
Param sub_addr
The sub address.
Param mod_id
The model ID within the element.

void (*`mod_sub_list`)(struct *bt_mesh_cfg_cli* *cli, uint16_t addr, uint8_t status, uint16_t elem_addr, uint16_t mod_id, uint16_t cid, struct *net_buf_simple* *buf)

Optional callback for Model Sub list messages.

Handles received Model Sub list messages from a server.

Note

The `buf` parameter should be decoded using *net_buf_simple_pull_le16* in iteration, as long as `buf->len` is greater than or equal to 2.

Param cli
Client that received the status message.
Param addr
Address of the sender.
Param status
Status code for the message.

Param elem_addr
Address of the element.
Param mod_id
Model ID.
Param cid
Company ID.
Param buf
Message buffer containing subscription addresses.

void (*`node_reset_status`)(struct *bt_mesh_cfg_cli* *cli, uint16_t addr)

Optional callback for Node Reset Status messages.

Handles received Node Reset Status messages from a server.
Param cli
Client that received the status message.
Param addr
Address of the sender.

void (*`beacon_status`)(struct *bt_mesh_cfg_cli* *cli, uint16_t addr, uint8_t status)

Optional callback for Beacon Status messages.

Handles received Beacon Status messages from a server.
Param cli
Client that received the status message.
Param addr
Address of the sender.
Param status
Status Code for requesting message.

void (*`ttl_status`)(struct *bt_mesh_cfg_cli* *cli, uint16_t addr, uint8_t status)

Optional callback for Default TTL Status messages.

Handles received Default TTL Status messages from a server.
Param cli
Client that received the status message.
Param addr
Address of the sender.
Param status
Status Code for requesting message.

void (*`friend_status`)(struct *bt_mesh_cfg_cli* *cli, uint16_t addr, uint8_t status)

Optional callback for Friend Status messages.

Handles received Friend Status messages from a server.
Param cli
Client that received the status message.
Param addr
Address of the sender.
Param status
Status Code for requesting message.

void (*`gatt_proxy_status`)(struct *bt_mesh_cfg_cli* *cli, uint16_t addr, uint8_t status)

Optional callback for GATT Proxy Status messages.

Handles received GATT Proxy Status messages from a server.
Param cli
Client that received the status message.

Param addr
Address of the sender.
Param status
Status Code for requesting message.

void (*`network_transmit_status`)(struct *bt_mesh_cfg_cli* *cli, uint16_t addr, uint8_t status)

Optional callback for Network Transmit Status messages.

Handles received Network Transmit Status messages from a server.
Param cli
Client that received the status message.
Param addr
Address of the sender.
Param status
Status Code for requesting message.

void (*`relay_status`)(struct *bt_mesh_cfg_cli* *cli, uint16_t addr, uint8_t status, uint8_t transmit)

Optional callback for Relay Status messages.

Handles received Relay Status messages from a server.
Param cli
Client that received the status message.
Param addr
Address of the sender.
Param status
Status Code for requesting message.
Param transmit
The relay retransmit count and interval steps.

void (*`net_key_status`)(struct *bt_mesh_cfg_cli* *cli, uint16_t addr, uint8_t status, uint16_t net_idx)

Optional callback for NetKey Status messages.

Handles received NetKey Status messages from a server.
Param cli
Client that received the status message.
Param addr
Address of the sender.
Param status
Status Code for requesting message.
Param net_idx
The index of the NetKey.

void (*`net_key_list`)(struct *bt_mesh_cfg_cli* *cli, uint16_t addr, struct *net_buf_simple* *buf)

Optional callback for Netkey list messages.

Handles received Netkey list messages from a server.

Note

The `buf` parameter should be decoded using the *bt_mesh_key_idx_unpack_list* helper function.

Param cli
Client that received the status message.

Param addr
Address of the sender.
Param buf
Message buffer containing key indexes.

void (*`app_key_status`)(struct *bt_mesh_cfg_cli* *cli, uint16_t addr, uint8_t status, uint16_t net_idx, uint16_t app_idx)

Optional callback for AppKey Status messages.

Handles received AppKey Status messages from a server.

Param cli
Client that received the status message.
Param addr
Address of the sender.
Param status
Status Code for requesting message.
Param net_idx
The index of the NetKey.
Param app_idx
The index of the AppKey.

void (*`app_key_list`)(struct *bt_mesh_cfg_cli* *cli, uint16_t addr, uint8_t status, uint16_t net_idx, struct *net_buf_simple* *buf)

Optional callback for Appkey list messages.

Handles received Appkey list messages from a server.

Note

The `buf` parameter should be decoded using the *bt_mesh_key_idx_unpack_list* helper function.

Param cli
Client that received the status message.
Param addr
Address of the sender.
Param status
Status code for the message.
Param net_idx
The index of the NetKey.
Param buf
Message buffer containing key indexes.

void (*`mod_app_status`)(struct *bt_mesh_cfg_cli* *cli, uint16_t addr, uint8_t status, uint16_t elem_addr, uint16_t app_idx, uint32_t mod_id)

Optional callback for Model App Status messages.

Handles received Model App Status messages from a server.

Param cli
Client that received the status message.
Param addr
Address of the sender.
Param status
Status Code for requesting message.
Param elem_addr
The unicast address of the element.
Param app_idx
The sub address.

Param mod_id
The model ID within the element.

void (*`mod_app_list`)(struct *bt_mesh_cfg_cli* *cli, uint16_t addr, uint8_t status, uint16_t elem_addr, uint16_t mod_id, uint16_t cid, struct *net_buf_simple* *buf)

Optional callback for Model App list messages.

Handles received Model App list messages from a server.

> **Note**
>
> The `buf` parameter should be decoded using the *bt_mesh_key_idx_unpack_list* helper function.

Param cli
Client that received the status message.
Param addr
Address of the sender.
Param status
Status code for the message.
Param elem_addr
Address of the element.
Param mod_id
Model ID.
Param cid
Company ID.
Param buf
Message buffer containing key indexes.

void (*`node_identity_status`)(struct *bt_mesh_cfg_cli* *cli, uint16_t addr, uint8_t status, uint16_t net_idx, uint8_t identity)

Optional callback for Node Identity Status messages.

Handles received Node Identity Status messages from a server.
Param cli
Client that received the status message.
Param addr
Address of the sender.
Param status
Status Code for requesting message.
Param net_idx
The index of the NetKey.
Param identity
The node identity state.

void (*`lpn_timeout_status`)(struct *bt_mesh_cfg_cli* *cli, uint16_t addr, uint16_t elem_addr, uint32_t timeout)

Optional callback for LPN PollTimeout Status messages.

Handles received LPN PollTimeout Status messages from a server.
Param cli
Client that received the status message.
Param addr
Address of the sender.
Param elem_addr
The unicast address of the LPN.
Param timeout
Current value of PollTimeout timer of the LPN.

void (*krp_status)(struct *bt_mesh_cfg_cli* *cli, uint16_t addr, uint8_t status, uint16_t net_idx, uint8_t phase)

Optional callback for Key Refresh Phase status messages.

Handles received Key Refresh Phase status messages from a server.

Param cli
Client that received the status message.

Param addr
Address of the sender.

Param status
Status code for the message.

Param net_idx
The index of the NetKey.

Param phase
Phase of the KRP.

void (*hb_pub_status)(struct *bt_mesh_cfg_cli* *cli, uint16_t addr, uint8_t status, struct *bt_mesh_cfg_cli_hb_pub* *pub)

Optional callback for Heartbeat pub status messages.

Handles received Heartbeat pub status messages from a server.

Param cli
Client that received the status message.

Param addr
Address of the sender.

Param status
Status code for the message.

Param pub
HB publication configuration parameters.

void (*hb_sub_status)(struct *bt_mesh_cfg_cli* *cli, uint16_t addr, uint8_t status, struct *bt_mesh_cfg_cli_hb_sub* *sub)

Optional callback for Heartbeat Sub status messages.

Handles received Heartbeat Sub status messages from a server.

Param cli
Client that received the status message.

Param addr
Address of the sender.

Param status
Status code for the message.

Param sub
HB subscription configuration parameters.

struct bt_mesh_cfg_cli

#include <*cfg_cli.h*> Mesh Configuration Client Model Context.

Public Members

const struct *bt_mesh_model* *model

Composition data model entry pointer.

const struct *bt_mesh_cfg_cli_cb* *cb

Optional callback for Mesh Configuration Client Status messages.

struct **bt_mesh_cfg_cli_mod_pub**

#include <cfg_cli.h> Model publication configuration parameters.

Public Members

uint16_t **addr**

Publication destination address.

const uint8_t ***uuid**

Virtual address UUID, or NULL if this is not a virtual address.

uint16_t **app_idx**

Application index to publish with.

bool **cred_flag**

Friendship credential flag.

uint8_t **ttl**

Time To Live to publish with.

uint8_t **period**

Encoded publish period.

See also

BT_MESH_PUB_PERIOD_100MS, *BT_MESH_PUB_PERIOD_SEC*, *BT_MESH_PUB_PERIOD_10SEC*, *BT_MESH_PUB_PERIOD_10MIN*

uint8_t **transmit**

Encoded transmit parameters.

See also

BT_MESH_TRANSMIT

struct **bt_mesh_cfg_cli_hb_sub**

#include <cfg_cli.h> Heartbeat subscription configuration parameters.

Public Members

uint16_t **src**

Source address to receive Heartbeat messages from.

uint16_t **dst**

Destination address to receive Heartbeat messages on.

uint8_t **period**

Logarithmic subscription period to keep listening for.

The decoded subscription period is (1 « (period - 1)) seconds, or 0 seconds if period is 0.

uint8_t **count**

Logarithmic Heartbeat subscription receive count.

The decoded Heartbeat count is (1 « (count - 1)) if count is between 1 and 0xfe, 0 if count is 0 and 0xffff if count is 0xff.

Ignored in Heartbeat subscription set.

uint8_t **min**

Minimum hops in received messages, ie the shortest registered path from the publishing node to the subscribing node.

A Heartbeat received from an immediate neighbor has hop count = 1.

Ignored in Heartbeat subscription set.

uint8_t **max**

Maximum hops in received messages, ie the longest registered path from the publishing node to the subscribing node.

A Heartbeat received from an immediate neighbor has hop count = 1.

Ignored in Heartbeat subscription set.

struct **bt_mesh_cfg_cli_hb_pub**

#include <cfg_cli.h> Heartbeat publication configuration parameters.

Public Members

uint16_t **dst**

Heartbeat destination address.

uint8_t **count**

Logarithmic Heartbeat count.

Decoded as (1 « (count - 1)) if count is between 1 and 0x11, 0 if count is 0, or “indefinitely” if count is 0xff.

When used in Heartbeat publication set, this parameter denotes the number of Heartbeat messages to send.

When returned from Heartbeat publication get, this parameter denotes the number of Heartbeat messages remaining to be sent.

uint8_t **period**

Logarithmic Heartbeat publication transmit interval in seconds.

Decoded as (1 « (period - 1)) if period is between 1 and 0x11. If period is 0, Heartbeat publication is disabled.

uint8_t **ttl**

Publication message Time To Live value.

uint16_t **feat**

Bitmap of features that trigger Heartbeat publications.

Legal values are *BT_MESH_FEAT_RELAY*, *BT_MESH_FEAT_PROXY*, *BT_MESH_FEAT_FRIEND* and *BT_MESH_FEAT_LOW_POWER*

uint16_t **net_idx**

Network index to publish with.

struct **bt_mesh_comp_p0**

#include <cfg_cli.h> Parsed Composition data page 0 representation.

Should be pulled from the return buffer passed to *bt_mesh_cfg_cli_comp_data_get* using *bt_mesh_comp_p0_get*.

Public Members

uint16_t **cid**

Company ID.

uint16_t **pid**

Product ID.

uint16_t **vid**

Version ID.

uint16_t **crpl**

Replay protection list size.

uint16_t **feat**

Supported features, see *BT_MESH_FEAT_SUPPORTED*.

struct **bt_mesh_comp_p0_elem**

#include <cfg_cli.h> Composition data page 0 element representation.

Public Members

uint16_t **loc**

Element location.

size_t **nsig**

The number of SIG models in this element.

size_t **nvnd**
The number of vendor models in this element.

struct **bt_mesh_comp_p1_elem**
#include <cfg_cli.h> Composition data page 1 element representation.

Public Members

size_t **nsig**
The number of SIG models in this element.

size_t **nvnd**
The number of vendor models in this element.

struct **bt_mesh_comp_p1_model_item**
#include <cfg_cli.h> Composition data page 1 model item representation.

Public Members

bool **cor_present**
Corresponding_Group_ID field indicator.

bool **format**
Determines the format of Extended Model Item.

uint8_t **ext_item_cnt**
Number of items in Extended Model Items.

uint8_t **cor_id**
Buffer containing Extended Model Items.
If cor_present is set to 1 it starts with Corresponding_Group_ID

struct **bt_mesh_comp_p1_item_short**
#include <cfg_cli.h> Extended Model Item in short representation.

Public Members

uint8_t **elem_offset**
Element address modifier.

uint8_t **mod_item_idx**
Model Index.

struct **bt_mesh_comp_p1_item_long**
#include <cfg_cli.h> Extended Model Item in long representation.

Public Members

uint8_t **elem_offset**
Element address modifier.

uint8_t **mod_item_idx**
Model Index.

struct **bt_mesh_comp_p1_ext_item**
#include <*cfg_cli.h*> Extended Model Item.

Public Members

struct *bt_mesh_comp_p1_item_short* **short_item**
Item in short representation.

struct *bt_mesh_comp_p1_item_long* **long_item**
Item in long representation.

struct **bt_mesh_comp_p2_record**
#include <*cfg_cli.h*> Composition data page 2 record parsing structure.

Public Members

uint16_t **id**
Mesh profile ID.

uint8_t **x**
Major version.

uint8_t **y**
Minor version.

uint8_t **z**
Z version.

struct *bt_mesh_comp_p2_record* **version**
Mesh Profile Version.

struct *net_buf_simple* ***elem_buf**
Element offset buffer.

struct *net_buf_simple* ***data_buf**
Additional data buffer.

Configuration Server The Configuration Server model is a foundation model defined by the Bluetooth Mesh specification. The Configuration Server model controls most parameters of the mesh node. It does not have an API of its own, but relies on a *Configuration Client* to control it.

The Configuration Server model is mandatory on all Bluetooth Mesh nodes, and must only be instantiated on the primary element.

API reference

group `bt_mesh_cfg_srv`

Configuration Server Model.

Defines

`BT_MESH_MODEL_CFG_SRV`

Generic Configuration Server model composition data entry.

Health Client The Health Client model interacts with a Health Server model to read out diagnostics and control the node's attention state.

All message passing functions in the Health Client API have `cli` as their first parameter. This is a pointer to the client model instance to be used in this function call. The second parameter is the `ctx` or message context. Message context contains netkey index, appkey index and unicast address that the target node uses.

The Health Client model is optional, and may be instantiated on any element. However, if a Health Client model is instantiated on an element other than the primary, an instance must also be present on the primary element.

See *Health faults* for a list of specification defined fault values.

API reference

group `bt_mesh_health_cli`

Health Client Model.

Defines

`BT_MESH_MODEL_HEALTH_CLI`(cli_data)

Generic Health Client model composition data entry.

Parameters

- **`cli_data`** – Pointer to a *Health Client Model* instance.

Functions

int **`bt_mesh_health_cli_fault_get`**(struct *bt_mesh_health_cli* *cli, struct *bt_mesh_msg_ctx* *ctx, uint16_t cid, uint8_t *test_id, uint8_t *faults, size_t *fault_count)

Get the registered fault state for the given Company ID.

This method can be used asynchronously by setting `test_id` and (`faults` or `fault_count`) as NULL This way the method will not wait for response and will return immediately after sending the command.

To process the response arguments of an async method, register the `fault_status` callback in *bt_mesh_health_cli* struct.

See also

Health faults

Parameters

- **cli** – Client model to send on.
- **ctx** – Message context, or NULL to use the configured publish parameters.
- **cid** – Company ID to get the registered faults of.
- **test_id** – Test ID response buffer.
- **faults** – Fault array response buffer.
- **fault_count** – Fault count response buffer.

Returns

0 on success, or (negative) error code on failure.

int **bt_mesh_health_cli_fault_clear**(struct *bt_mesh_health_cli* *cli, struct *bt_mesh_msg_ctx* *ctx, uint16_t cid, uint8_t *test_id, uint8_t *faults, size_t *fault_count)

Clear the registered faults for the given Company ID.

This method can be used asynchronously by setting `test_id` and (`faults` or `fault_count`) as NULL This way the method will not wait for response and will return immediately after sending the command.

To process the response arguments of an async method, register the `fault_status` callback in *bt_mesh_health_cli* struct.

See also

Health faults

Parameters

- **cli** – Client model to send on.
- **ctx** – Message context, or NULL to use the configured publish parameters.
- **cid** – Company ID to clear the registered faults for.
- **test_id** – Test ID response buffer.
- **faults** – Fault array response buffer.

- **fault_count** – Fault count response buffer.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_health_cli_fault_clear_unack**(struct *bt_mesh_health_cli* *cli, struct *bt_mesh_msg_ctx* *ctx, uint16_t cid)

Clear the registered faults for the given Company ID (unacked).

See also

Health faults

Parameters

- **cli** – Client model to send on.
- **ctx** – Message context, or NULL to use the configured publish parameters.
- **cid** – Company ID to clear the registered faults for.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_health_cli_fault_test**(struct *bt_mesh_health_cli* *cli, struct *bt_mesh_msg_ctx* *ctx, uint16_t cid, uint8_t test_id, uint8_t *faults, size_t *fault_count)

Invoke a self-test procedure for the given Company ID.

This method can be used asynchronously by setting `faults` or `fault_count` as NULL This way the method will not wait for response and will return immediately after sending the command.

To process the response arguments of an async method, register the `fault_status` callback in *`bt_mesh_health_cli`* struct.

Parameters

- **cli** – Client model to send on.
- **ctx** – Message context, or NULL to use the configured publish parameters.
- **cid** – Company ID to invoke the test for.
- **test_id** – Test ID response buffer.
- **faults** – Fault array response buffer.
- **fault_count** – Fault count response buffer.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_health_cli_fault_test_unack**(struct *bt_mesh_health_cli* *cli, struct *bt_mesh_msg_ctx* *ctx, uint16_t cid, uint8_t test_id)

Invoke a self-test procedure for the given Company ID (unacked).

Parameters

- **cli** – Client model to send on.

- **ctx** – Message context, or NULL to use the configured publish parameters.
- **cid** – Company ID to invoke the test for.
- **test_id** – Test ID response buffer.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_health_cli_period_get**(struct *bt_mesh_health_cli* *cli, struct *bt_mesh_msg_ctx* *ctx, uint8_t *divisor)

Get the target node's Health fast period divisor.

The health period divisor is used to increase the publish rate when a fault is registered. Normally, the Health server will publish with the period in the configured publish parameters. When a fault is registered, the publish period is divided by (1 « divisor). For example, if the target node's Health server is configured to publish with a period of 16 seconds, and the Health fast period divisor is 5, the Health server will publish with an interval of 500 ms when a fault is registered.

This method can be used asynchronously by setting `divisor` as NULL. This way the method will not wait for response and will return immediately after sending the command.

To process the response arguments of an async method, register the `period_status` callback in *`bt_mesh_health_cli`* struct.

Parameters

- **cli** – Client model to send on.
- **ctx** – Message context, or NULL to use the configured publish parameters.
- **divisor** – Health period divisor response buffer.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_health_cli_period_set**(struct *bt_mesh_health_cli* *cli, struct *bt_mesh_msg_ctx* *ctx, uint8_t divisor, uint8_t *updated_divisor)

Set the target node's Health fast period divisor.

The health period divisor is used to increase the publish rate when a fault is registered. Normally, the Health server will publish with the period in the configured publish parameters. When a fault is registered, the publish period is divided by (1 « divisor). For example, if the target node's Health server is configured to publish with a period of 16 seconds, and the Health fast period divisor is 5, the Health server will publish with an interval of 500 ms when a fault is registered.

This method can be used asynchronously by setting `updated_divisor` as NULL. This way the method will not wait for response and will return immediately after sending the command.

To process the response arguments of an async method, register the `period_status` callback in *`bt_mesh_health_cli`* struct.

Parameters

- **cli** – Client model to send on.
- **ctx** – Message context, or NULL to use the configured publish parameters.
- **divisor** – New Health period divisor.

- **updated_divisor** – Health period divisor response buffer.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_health_cli_period_set_unack**(struct *bt_mesh_health_cli* *cli, struct *bt_mesh_msg_ctx* *ctx, uint8_t divisor)

Set the target node's Health fast period divisor (unacknowledged).

This is an unacknowledged version of this API.

Parameters

- **cli** – Client model to send on.
- **ctx** – Message context, or NULL to use the configured publish parameters.
- **divisor** – New Health period divisor.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_health_cli_attention_get**(struct *bt_mesh_health_cli* *cli, struct *bt_mesh_msg_ctx* *ctx, uint8_t *attention)

Get the current attention timer value.

This method can be used asynchronously by setting `attention` as NULL. This way the method will not wait for response and will return immediately after sending the command.

To process the response arguments of an async method, register the `attention_status` callback in *bt_mesh_health_cli* struct.

Parameters

- **cli** – Client model to send on.
- **ctx** – Message context, or NULL to use the configured publish parameters.
- **attention** – Attention timer response buffer, measured in seconds.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_health_cli_attention_set**(struct *bt_mesh_health_cli* *cli, struct *bt_mesh_msg_ctx* *ctx, uint8_t attention, uint8_t *updated_attention)

Set the attention timer.

This method can be used asynchronously by setting `updated_attention` as NULL. This way the method will not wait for response and will return immediately after sending the command.

To process the response arguments of an async method, register the `attention_status` callback in *bt_mesh_health_cli* struct.

Parameters

- **cli** – Client model to send on.
- **ctx** – Message context, or NULL to use the configured publish parameters.
- **attention** – New attention timer time, in seconds.
- **updated_attention** – Attention timer response buffer, measured in seconds.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_health_cli_attention_set_unack**(struct *bt_mesh_health_cli* *cli, struct *bt_mesh_msg_ctx* *ctx, uint8_t attention)

Set the attention timer (unacknowledged).

Parameters

- **cli** – Client model to send on.
- **ctx** – Message context, or NULL to use the configured publish parameters.
- **attention** – New attention timer time, in seconds.

Returns
0 on success, or (negative) error code on failure.

int32_t **bt_mesh_health_cli_timeout_get**(void)

Get the current transmission timeout value.

Returns
The configured transmission timeout in milliseconds.

void **bt_mesh_health_cli_timeout_set**(int32_t timeout)

Set the transmission timeout value.

Parameters

- **timeout** – The new transmission timeout.

struct **bt_mesh_health_cli**

#include <health_cli.h> Health Client Model Context.

Public Members

const struct *bt_mesh_model* ***model**

Composition data model entry pointer.

struct *bt_mesh_model_pub* **pub**

Publication structure instance.

struct *net_buf_simple* **pub_buf**

Publication buffer.

uint8_t **pub_data**[*BT_MESH_MODEL_BUF_LEN*(*BT_MESH_MODEL_OP_2*(0x80, 0x32), 3)]

Publication data.

void (***period_status**)(struct *bt_mesh_health_cli* *cli, uint16_t addr, uint8_t divisor)

Optional callback for Health Period Status messages.

Handles received Health Period Status messages from a Health server. The `divisor` param represents the period divisor value.

Param cli
Health client that received the status message.

Param addr
Address of the sender.

Param divisor
Health Period Divisor value.

void (*`attention_status`)(struct *bt_mesh_health_cli* *cli, uint16_t addr, uint8_t attention)

Optional callback for Health Attention Status messages.

Handles received Health Attention Status messages from a Health server. The `attention` param represents the current attention value.

Param cli
Health client that received the status message.
Param addr
Address of the sender.
Param attention
Current attention value.

void (*`fault_status`)(struct *bt_mesh_health_cli* *cli, uint16_t addr, uint8_t test_id, uint16_t cid, uint8_t *faults, size_t fault_count)

Optional callback for Health Fault Status messages.

Handles received Health Fault Status messages from a Health server. The `fault` array represents all faults that are currently present in the server's element.

See also

Health faults

Param cli
Health client that received the status message.
Param addr
Address of the sender.
Param test_id
Identifier of a most recently performed test.
Param cid
Company Identifier of the node.
Param faults
Array of faults.
Param fault_count
Number of faults in the fault array.

void (*`current_status`)(struct *bt_mesh_health_cli* *cli, uint16_t addr, uint8_t test_id, uint16_t cid, uint8_t *faults, size_t fault_count)

Optional callback for Health Current Status messages.

Handles received Health Current Status messages from a Health server. The `fault` array represents all faults that are currently present in the server's element.

See also

Health faults

Param cli
Health client that received the status message.

Param addr
Address of the sender.
Param test_id
Identifier of a most recently performed test.
Param cid
Company Identifier of the node.
Param faults
Array of faults.
Param fault_count
Number of faults in the fault array.

Health Server The Health Server model provides attention callbacks and node diagnostics for *Health Client* models. It is primarily used to report faults in the mesh node and map the mesh nodes to their physical location.

If present, the Health Server model must be instantiated on the primary element.

Faults The Health Server model may report a list of faults that have occurred in the device's lifetime. Typically, the faults are events or conditions that may alter the behavior of the node, like power outages or faulty peripherals. Faults are split into warnings and errors. Warnings indicate conditions that are close to the limits of what the node is designed to withstand, but not necessarily damaging to the device. Errors indicate conditions that are outside of the node's design limits, and may have caused invalid behavior or permanent damage to the device.

Fault values `0x01` to `0x7f` are reserved for the Bluetooth Mesh specification, and the full list of specification defined faults are available in *Health faults*. Fault values `0x80` to `0xff` are vendor specific. The list of faults are always reported with a company ID to help interpreting the vendor specific faults.

Attention state The attention state is used to make the device call attention to itself through some physical behavior like blinking, playing a sound or vibrating. The attention state may be used during provisioning to let the user know which device they're provisioning, as well as through the Health models at runtime.

The attention state is always assigned a timeout in the range of one to 255 seconds when enabled. The Health Server API provides two callbacks for the application to run their attention calling behavior: *bt_mesh_health_srv_cb.attn_on* is called at the beginning of the attention period, *bt_mesh_health_srv_cb.attn_off* is called at the end.

The remaining time for the attention period may be queried through *bt_mesh_health_srv.attn_timer*.

API reference

group `bt_mesh_health_srv`

Health Server Model.

Defines

`BT_MESH_HEALTH_PUB_DEFINE`(_name, _max_faults)

A helper to define a health publication context.

Parameters

- `_name` – Name given to the publication context variable.
- `_max_faults` – Maximum number of faults the element can have.

BT_MESH_MODEL_HEALTH_SRV(srv, pub, ...)

Define a new health server model.

Note that this API needs to be repeated for each element that the application wants to have a health server model on. Each instance also needs a unique *bt_mesh_health_srv* and *bt_mesh_model_pub* context.

Parameters

- **srv** – Pointer to a unique struct *bt_mesh_health_srv*.
- **pub** – Pointer to a unique struct *bt_mesh_model_pub*.
- **...** – Optional Health Server metadata if application is compiled with Large Composition Data Server support, otherwise this parameter is ignored.

Returns

New mesh model instance.

BT_MESH_HEALTH_TEST_INFO_METADATA_ID

Health Test Information Metadata ID.

BT_MESH_HEALTH_TEST_INFO_METADATA(tests)

BT_MESH_HEALTH_TEST_INFO(cid, tests...)

Define a Health Test Info Metadata array.

Parameters

- **cid** – Company ID of the Health Test suite.
- **tests** – A comma separated list of tests.

Returns

A comma separated list of values that make Health Test Info Metadata

Functions

int **bt_mesh_health_srv_fault_update**(const struct *bt_mesh_elem* *elem)

Notify the stack that the fault array state of the given element has changed.

This prompts the Health server on this element to publish the current fault array if periodic publishing is disabled.

Parameters

- **elem** – Element to update the fault state of.

Returns

0 on success, or (negative) error code otherwise.

struct **bt_mesh_health_srv_cb**

#include <health_srv.h> Callback function for the Health Server model.

Public Members

int (***fault_get_cur**)(const struct *bt_mesh_model* *model, uint8_t *test_id, uint16_t *company_id, uint8_t *faults, uint8_t *fault_count)

Callback for fetching current faults.

Fault values may either be defined by the specification, or by a vendor. Vendor specific faults should be interpreted in the context of the accompanying Company ID. Specification defined faults may be reported for any Company ID, and the same fault may be presented for multiple Company IDs.

All faults shall be associated with at least one Company ID, representing the device vendor or some other vendor whose vendor specific fault values are used.

If there are multiple Company IDs that have active faults, return only the faults associated with one of them at the time. To report faults for multiple Company IDs, interleave which Company ID is reported for each call.

Param model
Health Server model instance to get faults of.

Param test_id
Test ID response buffer.

Param company_id
Company ID response buffer.

Param faults
Array to fill with current faults.

Param fault_count
The number of faults the fault array can fit. Should be updated to reflect the number of faults copied into the array.

Return
0 on success, or (negative) error code otherwise.

int (*`fault_get_reg`)(const struct *bt_mesh_model* *model, uint16_t company_id, uint8_t *test_id, uint8_t *faults, uint8_t *fault_count)

Callback for fetching all registered faults.

Registered faults are all past and current faults since the last call to `fault_clear`. Only faults associated with the given Company ID should be reported.

Fault values may either be defined by the specification, or by a vendor. Vendor specific faults should be interpreted in the context of the accompanying Company ID. Specification defined faults may be reported for any Company ID, and the same fault may be presented for multiple Company IDs.

Param model
Health Server model instance to get faults of.

Param company_id
Company ID to get faults for.

Param test_id
Test ID response buffer.

Param faults
Array to fill with registered faults.

Param fault_count
The number of faults the fault array can fit. Should be updated to reflect the number of faults copied into the array.

Return
0 on success, or (negative) error code otherwise.

int (*`fault_clear`)(const struct *bt_mesh_model* *model, uint16_t company_id)

Clear all registered faults associated with the given Company ID.

Param model
Health Server model instance to clear faults of.

Param company_id
Company ID to clear faults for.

Return
0 on success, or (negative) error code otherwise.

int (*`fault_test`)(const struct *bt_mesh_model* *model, uint8_t test_id, uint16_t company_id)

Run a self-test.

The Health server may support up to 256 self-tests for each Company ID. The behavior for all test IDs are vendor specific, and should be interpreted based on the accompanying Company ID. Test failures should result in changes to the fault array.

Param model
Health Server model instance to run test for.
Param test_id
Test ID to run.
Param company_id
Company ID to run test for.
Return
0 if the test execution was started successfully, or (negative) error code otherwise. Note that the fault array will not be reported back to the client if the test execution didn't start.

void (*`attn_on`)(const struct *bt_mesh_model* *model)

Start calling attention to the device.

The attention state is used to map an element address to a physical device. When this callback is called, the device should start some physical procedure meant to call attention to itself, like blinking, buzzing, vibrating or moving. If there are multiple Health server instances on the device, the attention state should also help identify the specific element the server is in.

The attention calling behavior should continue until the `attn_off` callback is called.

Param model
Health Server model to start the attention state of.

void (*`attn_off`)(const struct *bt_mesh_model* *model)

Stop the attention state.

Any physical activity started to call attention to the device should be stopped.

Param model

struct `bt_mesh_health_srv`

#include <health_srv.h> Mesh Health Server Model Context.

Public Members

const struct *bt_mesh_model* *`model`

Composition data model entry pointer.

const struct *bt_mesh_health_srv_cb* *`cb`

Optional callback struct.

struct *k_work_delayable* `attn_timer`

Attention Timer state.

Health faults Fault values defined by the Bluetooth Mesh specification.

group `bt_mesh_health_faults`

List of specification defined Health fault values.

Defines

`BT_MESH_HEALTH_FAULT_NO_FAULT`

No fault has occurred.

`BT_MESH_HEALTH_FAULT_BATTERY_LOW_WARNING`

`BT_MESH_HEALTH_FAULT_BATTERY_LOW_ERROR`

`BT_MESH_HEALTH_FAULT_SUPPLY_VOLTAGE_TOO_LOW_WARNING`

`BT_MESH_HEALTH_FAULT_SUPPLY_VOLTAGE_TOO_LOW_ERROR`

`BT_MESH_HEALTH_FAULT_SUPPLY_VOLTAGE_TOO_HIGH_WARNING`

`BT_MESH_HEALTH_FAULT_SUPPLY_VOLTAGE_TOO_HIGH_ERROR`

`BT_MESH_HEALTH_FAULT_POWER_SUPPLY_INTERRUPTED_WARNING`

`BT_MESH_HEALTH_FAULT_POWER_SUPPLY_INTERRUPTED_ERROR`

`BT_MESH_HEALTH_FAULT_NO_LOAD_WARNING`

`BT_MESH_HEALTH_FAULT_NO_LOAD_ERROR`

`BT_MESH_HEALTH_FAULT_OVERLOAD_WARNING`

`BT_MESH_HEALTH_FAULT_OVERLOAD_ERROR`

`BT_MESH_HEALTH_FAULT_OVERHEAT_WARNING`

`BT_MESH_HEALTH_FAULT_OVERHEAT_ERROR`

`BT_MESH_HEALTH_FAULT_CONDENSATION_WARNING`

`BT_MESH_HEALTH_FAULT_CONDENSATION_ERROR`

`BT_MESH_HEALTH_FAULT_VIBRATION_WARNING`

`BT_MESH_HEALTH_FAULT_VIBRATION_ERROR`

BT_MESH_HEALTH_FAULT_CONFIGURATION_WARNING

BT_MESH_HEALTH_FAULT_CONFIGURATION_ERROR

BT_MESH_HEALTH_FAULT_ELEMENT_NOT_CALIBRATED_WARNING

BT_MESH_HEALTH_FAULT_ELEMENT_NOT_CALIBRATED_ERROR

BT_MESH_HEALTH_FAULT_MEMORY_WARNING

BT_MESH_HEALTH_FAULT_MEMORY_ERROR

BT_MESH_HEALTH_FAULT_SELF_TEST_WARNING

BT_MESH_HEALTH_FAULT_SELF_TEST_ERROR

BT_MESH_HEALTH_FAULT_INPUT_TOO_LOW_WARNING

BT_MESH_HEALTH_FAULT_INPUT_TOO_LOW_ERROR

BT_MESH_HEALTH_FAULT_INPUT_TOO_HIGH_WARNING

BT_MESH_HEALTH_FAULT_INPUT_TOO_HIGH_ERROR

BT_MESH_HEALTH_FAULT_INPUT_NO_CHANGE_WARNING

BT_MESH_HEALTH_FAULT_INPUT_NO_CHANGE_ERROR

BT_MESH_HEALTH_FAULT_ACTUATOR_BLOCKED_WARNING

BT_MESH_HEALTH_FAULT_ACTUATOR_BLOCKED_ERROR

BT_MESH_HEALTH_FAULT_HOUSING_OPENED_WARNING

BT_MESH_HEALTH_FAULT_HOUSING_OPENED_ERROR

BT_MESH_HEALTH_FAULT_TAMPER_WARNING

BT_MESH_HEALTH_FAULT_TAMPER_ERROR

BT_MESH_HEALTH_FAULT_DEVICE_MOVED_WARNING

BT_MESH_HEALTH_FAULT_DEVICE_MOVED_ERROR

BT_MESH_HEALTH_FAULT_DEVICE_DROPPED_WARNING

`BT_MESH_HEALTH_FAULT_DEVICE_DROPPED_ERROR`

`BT_MESH_HEALTH_FAULT_OVERFLOW_WARNING`

`BT_MESH_HEALTH_FAULT_OVERFLOW_ERROR`

`BT_MESH_HEALTH_FAULT_EMPTY_WARNING`

`BT_MESH_HEALTH_FAULT_EMPTY_ERROR`

`BT_MESH_HEALTH_FAULT_INTERNAL_BUS_WARNING`

`BT_MESH_HEALTH_FAULT_INTERNAL_BUS_ERROR`

`BT_MESH_HEALTH_FAULT_MECHANISM_JAMMED_WARNING`

`BT_MESH_HEALTH_FAULT_MECHANISM_JAMMED_ERROR`

`BT_MESH_HEALTH_FAULT_VENDOR_SPECIFIC_START`

Start of the vendor specific fault values.

All values below this are reserved for the Bluetooth Specification.

Large Composition Data Client The Large Composition Data Client model is a foundation model defined by the Bluetooth Mesh specification. The model is optional, and is enabled through the `CONFIG_BT_MESH_LARGE_COMP_DATA_CLI` option.

The Large Composition Data Client model was introduced in the Bluetooth Mesh Protocol Specification version 1.1, and supports the functionality of reading pages of Composition Data that do not fit in a Config Composition Data Status message and reading the metadata of the model instances on a node that supports the *Large Composition Data Server* model.

The Large Composition Data Client model communicates with a Large Composition Data Server model using the device key of the node containing the target Large Composition Data Server model instance.

If present, the Large Composition Data Client model must only be instantiated on the primary element.

API reference

group `bt_mesh_large_comp_data_cli`

Defines

`BT_MESH_MODEL_LARGE_COMP_DATA_CLI(cli_data)`

Large Composition Data Client model Composition Data entry.

Parameters

- `cli_data` – Pointer to a *Large Composition Data Client model* instance.

Functions

int **bt_mesh_large_comp_data_get**(uint16_t net_idx, uint16_t addr, uint8_t page, size_t offset, struct *bt_mesh_large_comp_data_rsp* *rsp)

Send Large Composition Data Get message.

This API is used to read a portion of a Composition Data Page.

This API can be used asynchronously by setting `rsp` as NULL. This way, the method will not wait for a response and will return immediately after sending the command.

When `rsp` is set, the user is responsible for providing a buffer for the Composition Data in *bt_mesh_large_comp_data_rsp::data*. If a buffer is not provided, the metadata won't be copied.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node element address.
- **page** – Composition Data Page to read.
- **offset** – Offset within the Composition Data Page.
- **rsp** – Pointer to a struct storing the received response from the server, or NULL to not wait for a response.

Returns

0 on success, or (negative) error code on failure.

int **bt_mesh_models_metadata_get**(uint16_t net_idx, uint16_t addr, uint8_t page, size_t offset, struct *bt_mesh_large_comp_data_rsp* *rsp)

Send Models Metadata Get message.

This API is used to read a portion of a Models Metadata Page.

This API can be used asynchronously by setting `rsp` as NULL. This way, the method will not wait for a response and will return immediately after sending the command.

When `rsp` is set, a user is responsible for providing a buffer for metadata in *bt_mesh_large_comp_data_rsp::data*. If a buffer is not provided, the metadata won't be copied.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node element address.
- **page** – Models Metadata Page to read.
- **offset** – Offset within the Models Metadata Page.
- **rsp** – Pointer to a struct storing the received response from the server, or NULL to not wait for a response.

Returns

0 on success, or (negative) error code on failure.

struct **bt_mesh_large_comp_data_rsp**

#include <large_comp_data_cli.h> Large Composition Data response.

Public Members

uint8_t **page**

Page number.

uint16_t **offset**

Offset within the page.

uint16_t **total_size**

Total size of the page.

struct *net_buf_simple* ***data**

Pointer to allocated buffer for storing received data.

struct **bt_mesh_large_comp_data_cli_cb**

#include <large_comp_data_cli.h> Large Composition Data Status messages callbacks.

Public Members

void (***large_comp_data_status**)(struct *bt_mesh_large_comp_data_cli* *cli, uint16_t addr, struct *bt_mesh_large_comp_data_rsp* *rsp)

Optional callback for Large Composition Data Status message.

Handles received Large Composition Data Status messages from a Large Composition Data Server.

If the content of `rsp` is needed after exiting this callback, a user should deep copy it.

Param cli
Large Composition Data Client context.

Param addr
Address of the sender.

Param rsp
Response received from the server.

void (***models_metadata_status**)(struct *bt_mesh_large_comp_data_cli* *cli, uint16_t addr, struct *bt_mesh_large_comp_data_rsp* *rsp)

Optional callback for Models Metadata Status message.

Handles received Models Metadata Status messages from a Large Composition Data Server.

If the content of `rsp` is needed after exiting this callback, a user should deep copy it.

Param cli
Large Composition Data Client context.

Param addr
Address of the sender.

Param rsp
Response received from the server.

struct **bt_mesh_large_comp_data_cli**

#include <large_comp_data_cli.h> Large Composition Data Client model context.

Public Members

const struct *bt_mesh_model* *model

Model entry pointer.

struct *bt_mesh_msg_ack_ctx* ack_ctx

Internal parameters for tracking message responses.

const struct *bt_mesh_large_comp_data_cli_cb* *cb

Optional callback for Large Composition Data Status messages.

Large Composition Data Server The Large Composition Data Server model is a foundation model defined by the Bluetooth Mesh specification. The model is optional, and is enabled through the CONFIG_BT_MESH_LARGE_COMP_DATA_SRV option.

The Large Composition Data Server model was introduced in the Bluetooth Mesh Protocol Specification version 1.1, and is used to support the functionality of exposing pages of Composition Data that do not fit in a Config Composition Data Status message and to expose metadata of the model instances.

The Large Composition Data Server does not have an API of its own and relies on a *Large Composition Data Client* to control it. The model only accepts messages encrypted with the node's device key.

If present, the Large Composition Data Server model must only be instantiated on the primary element.

Models metadata The Large Composition Data Server model allows each model to have a list of model's specific metadata that can be read by the Large Composition Data Client model. The metadata list can be associated with the *bt_mesh_model* through the bt_mesh_model.metadata field. The metadata list consists of one or more entries defined by the *bt_mesh_models_metadata_entry* structure. Each entry contains the length and ID of the metadata, and a pointer to the raw data. Entries can be created using the *BT_MESH_MODELS_METADATA_ENTRY* macro. The *BT_MESH_MODELS_METADATA_END* macro marks the end of the metadata list and must always be present. If the model has no metadata, the helper macro *BT_MESH_MODELS_METADATA_NONE* can be used instead.

API reference

group bt_mesh_large_comp_data_srv

Defines

BT_MESH_MODEL_LARGE_COMP_DATA_SRV

Large Composition Data Server model composition data entry.

On-Demand Private Proxy Client The On-Demand Private Proxy Client model is a foundation model defined by the Bluetooth Mesh specification. The model is optional, and is enabled with the CONFIG_BT_MESH_OD_PRIV_PROXY_CLI option.

The On-Demand Private Proxy Client model was introduced in the Bluetooth Mesh Protocol Specification version 1.1, and is used to set and retrieve the On-Demand Private GATT Proxy state. The

state defines how long a node will advertise Mesh Proxy Service with Private Network Identity type after it receives a Solicitation PDU.

The On-Demand Private Proxy Client model communicates with an On-Demand Private Proxy Server model using the device key of the node containing the target On-Demand Private Proxy Server model instance.

If present, the On-Demand Private Proxy Client model must only be instantiated on the primary element.

Configurations The On-Demand Private Proxy Client model behavior can be configured with the transmission timeout option `CONFIG_BT_MESH_OD_PRIV_PROXY_CLI_TIMEOUT`. The `CONFIG_BT_MESH_OD_PRIV_PROXY_CLI_TIMEOUT` controls how long the Client waits for a state response message to arrive in milliseconds. This value can be changed at runtime using *bt_mesh_od_priv_proxy_cli_timeout_set()*.

API reference

group **bt_mesh_od_priv_proxy_cli**

Defines

BT_MESH_MODEL_OD_PRIV_PROXY_CLI(cli_data)

On-Demand Private Proxy Client model composition data entry.

Functions

int **bt_mesh_od_priv_proxy_cli_get**(uint16_t net_idx, uint16_t addr, uint8_t *val_rsp)

Get the target's On-Demand Private GATT Proxy state.

This method can be used asynchronously by setting `val_rsp` as NULL. This way the method will not wait for response and will return immediately after sending the command.

To process the response arguments of an async method, register the `od_status` callback in *bt_mesh_od_priv_proxy_cli* struct.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **val_rsp** – Response buffer for On-Demand Private GATT Proxy value.

Returns
0 on success, or (negative) error code otherwise.

int **bt_mesh_od_priv_proxy_cli_set**(uint16_t net_idx, uint16_t addr, uint8_t val, uint8_t *val_rsp)

Set the target's On-Demand Private GATT Proxy state.

This method can be used asynchronously by setting `val_rsp` as NULL. This way the method will not wait for response and will return immediately after sending the command.

To process the response arguments of an async method, register the `od_status` callback in *bt_mesh_od_priv_proxy_cli* struct.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **val** – On-Demand Private GATT Proxy state to be set
- **val_rsp** – Response buffer for On-Demand Private GATT Proxy value.

Returns
0 on success, or (negative) error code otherwise.

void **bt_mesh_od_priv_proxy_cli_timeout_set**(int32_t timeout)

Set the transmission timeout value.

Parameters

- **timeout** – The new transmission timeout in milliseconds.

struct **bt_mesh_od_priv_proxy_cli**

#include <od_priv_proxy_cli.h> On-Demand Private Proxy Client Model Context.

Public Members

const struct *bt_mesh_model* ***model**

Solicitation PDU RPL model entry pointer.

void (***od_status**)(struct *bt_mesh_od_priv_proxy_cli* *cli, uint16_t addr, uint8_t state)

Optional callback for On-Demand Private Proxy Status messages.

Handles received On-Demand Private Proxy Status messages from a On-Demand Private Proxy server.The `state` param represents state of On-Demand Private Proxy server.

Param cli
On-Demand Private Proxy client that received the status message.
Param addr
Address of the sender.
Param state
State value.

On-Demand Private Proxy Server The On-Demand Private Proxy Server model is a foundation model defined by the Bluetooth Mesh specification. It is enabled with the `CONFIG_BT_MESH_OD_PRIV_PROXY_SRV` option.

The On-Demand Private Proxy Server model was introduced in the Bluetooth Mesh Protocol Specification version 1.1, and supports the configuration of advertising with Private Network Identity type of a node that is a recipient of Solicitation PDUs by managing its On-Demand Private GATT Proxy state.

When enabled, the *Solicitation PDU RPL Configuration Server* is also enabled. The On-Demand Private Proxy Server is dependent on the *Private Beacon Server* to be present on the node.

The On-Demand Private Proxy Server does not have an API of its own, and relies on a *On-Demand Private Proxy Client* to control it. The On-Demand Private Proxy Server model only accepts messages encrypted with the node's device key.

If present, the On-Demand Private Proxy Server model must only be instantiated on the primary element.

API reference

group `bt_mesh_od_priv_proxy_srv`

Defines

`BT_MESH_MODEL_OD_PRIV_PROXY_SRV`

On-Demand Private Proxy Server model composition data entry.

Opcodes Aggregator Client The Opcodes Aggregator Client model is a foundation model defined by the Bluetooth Mesh specification. It is an optional model, enabled with the `CONFIG_BT_MESH_OP_AGG_CLI` option.

The Opcodes Aggregator Client model is introduced in the Bluetooth Mesh Protocol Specification version 1.1, and is used to support the functionality of dispatching a sequence of access layer messages to nodes supporting the *Opcodes Aggregator Server* model.

The Opcodes Aggregator Client model communicates with an Opcodes Aggregator Server model using the device key of the target node or the application keys configured by the Configuration Client.

If present, the Opcodes Aggregator Client model must only be instantiated on the primary element.

The Opcodes Aggregator Client model is implicitly bound to the device key on initialization. It should be bound to the same application keys as the client models that are used to produce the sequence of messages.

To be able to aggregate a message from a client model, it should support an asynchronous API, for example through callbacks.

API reference

group `bt_mesh_op_agg_cli`

Defines

`BT_MESH_MODEL_OP_AGG_CLI`

Opcodes Aggregator Client model composition data entry.

Functions

int **`bt_mesh_op_agg_cli_seq_start`**(uint16_t net_idx, uint16_t app_idx, uint16_t dst, uint16_t elem_addr)

Configure Opcodes Aggregator context.

Parameters

- `net_idx` – NetKey index to encrypt with.
- `app_idx` – AppKey index to encrypt with.
- `dst` – Target Opcodes Aggregator Server address.
- `elem_addr` – Target node element address for the sequence message.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_op_agg_cli_seq_send**(void)
Opcodes Aggregator message send.

Uses previously configured context and sends aggregated message to target node.

Returns
0 on success, or (negative) error code on failure.

void **bt_mesh_op_agg_cli_seq_abort**(void)
Abort Opcodes Aggregator context.

bool **bt_mesh_op_agg_cli_seq_is_started**(void)
Check if Opcodes Aggregator Sequence context is started.

Returns
true if it is started, otherwise false.

size_t **bt_mesh_op_agg_cli_seq_tailroom**(void)
Get Opcodes Aggregator context tailroom.

Returns
Remaining tailroom of Opcodes Aggregator SDU.

int32_t **bt_mesh_op_agg_cli_timeout_get**(void)
Get the current transmission timeout value.

Returns
The configured transmission timeout in milliseconds.

void **bt_mesh_op_agg_cli_timeout_set**(int32_t timeout)
Set the transmission timeout value.

Parameters

- **timeout** – The new transmission timeout.

Opcodes Aggregator Server The Opcodes Aggregator Server model is a foundation model defined by the Bluetooth mesh specification. It is an optional model, enabled with the CONFIG_BT_MESH_OP_AGG_SRV option.

The Opcodes Aggregator Server model is introduced in the Bluetooth Mesh Protocol Specification version 1.1, and is used to support the functionality of processing a sequence of access layer messages.

The Opcodes Aggregator Server model accepts messages encrypted with the node's device key or the application keys.

If present, the Opcodes Aggregator Server model must only be instantiated on the primary element.

The targeted server models should be bound to the same application key that is used to encrypt the sequence of access layer messages sent to the Opcodes Aggregator Server.

The Opcodes Aggregator Server handles aggregated messages and dispatches them to the respective models and their message handlers. Current implementation assumes that responses are sent from the same execution context as the received message and doesn't allow to send a postponed response, for example from a work queue.

API reference

group bt_mesh_op_agg_srv

Defines

`BT_MESH_MODEL_OP_AGG_SRV`

Opcodes Aggretator Server model composition data entry.

> **Note**
>
> The Opcodes Aggregator Server handles aggregated messages and dispatches them to the respective models and their message handlers. Current implementation assumes that responses are sent from the same execution context as the received message and doesn't allow to send a postponed response, e.g. from workqueue.

Private Beacon Client The Private Beacon Client model is a foundation model defined by the Bluetooth mesh specification. It is enabled with the `CONFIG_BT_MESH_PRIV_BEACON_CLI` option.

The Private Beacon Client model is introduced in the Bluetooth Mesh Protocol Specification version 1.1, and provides functionality for configuring the *Private Beacon Server* models.

The Private Beacons feature adds privacy to the different Bluetooth Mesh beacons by periodically randomizing the beacon input data. This protects the mesh node from being tracked by devices outside the mesh network, and hides the network's IV index, IV update and the Key Refresh state.

The Private Beacon Client model communicates with a *Private Beacon Server* model using the device key of the target node. The Private Beacon Client model may communicate with servers on other nodes or self-configure through the local Private Beacon Server model.

All configuration functions in the Private Beacon Client API have `net_idx` and `addr` as their first parameters. These should be set to the network index and the primary unicast address the target node was provisioned with.

If present, the Private Beacon Client model must only be instantiated on the primary element.

API reference

group `bt_mesh_priv_beacon_cli`

Defines

`BT_MESH_MODEL_PRIV_BEACON_CLI`(cli_data)

Private Beacon Client model composition data entry.

Parameters

- `cli_data` – Pointer to a *Bluetooth Mesh Private Beacon Client* instance.

Functions

int `bt_mesh_priv_beacon_cli_set`(uint16_t net_idx, uint16_t addr, struct *bt_mesh_priv_beacon* *val, struct *bt_mesh_priv_beacon* *rsp)

Set the target's Private Beacon state.

This method can be used asynchronously by setting `rsp` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `val` – New Private Beacon value.
- `rsp` – If set, returns response status on success.

Returns
0 on success, or (negative) error code otherwise.

int `bt_mesh_priv_beacon_cli_get`(uint16_t net_idx, uint16_t addr, struct *bt_mesh_priv_beacon* *val)

Get the target's Private Beacon state.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `val` – Response buffer for Private Beacon value.

Returns
0 on success, or (negative) error code otherwise.

int `bt_mesh_priv_beacon_cli_gatt_proxy_set`(uint16_t net_idx, uint16_t addr, uint8_t val, uint8_t *rsp)

Set the target's Private GATT Proxy state.

This method can be used asynchronously by setting `rsp` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `val` – New Private GATT Proxy value.
- `rsp` – If set, returns response status on success.

Returns
0 on success, or (negative) error code otherwise.

int `bt_mesh_priv_beacon_cli_gatt_proxy_get`(uint16_t net_idx, uint16_t addr, uint8_t *val)

Get the target's Private GATT Proxy state.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `val` – Response buffer for Private GATT Proxy value.

Returns
0 on success, or (negative) error code otherwise.

int `bt_mesh_priv_beacon_cli_node_id_set`(uint16_t net_idx, uint16_t addr, struct *bt_mesh_priv_node_id* *val, struct *bt_mesh_priv_node_id* *rsp)

Set the target's Private Node Identity state.

This method can be used asynchronously by setting `rsp` as NULL. This way the method will not wait for response and will return immediately after sending the command.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **val** – New Private Node Identity value.
- **rsp** – If set, returns response status on success.

Returns
0 on success, or (negative) error code otherwise.

int **bt_mesh_priv_beacon_cli_node_id_get**(uint16_t net_idx, uint16_t addr, uint16_t key_net_idx, struct *bt_mesh_priv_node_id* *val)

Get the target's Private Node Identity state.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **key_net_idx** – Network index to get the Private Node Identity state of.
- **val** – Response buffer for Private Node Identity value.

Returns
0 on success, or (negative) error code otherwise.

struct **bt_mesh_priv_beacon**

#include <priv_beacon_cli.h> Private Beacon.

Public Members

uint8_t **enabled**

Private beacon is enabled.

uint8_t **rand_interval**

Random refresh interval (in 10 second steps), or 0 to keep current value.

struct **bt_mesh_priv_node_id**

#include <priv_beacon_cli.h> Private Node Identity.

Public Members

uint16_t **net_idx**

Index of the NetKey.

uint8_t **state**

Private Node Identity state.

uint8_t **status**

Response status code.

struct **bt_mesh_priv_beacon_cli_cb**

#include <priv_beacon_cli.h> Private Beacon Client Status messages callbacks.

Public Members

void (*priv_beacon_status)(struct *bt_mesh_priv_beacon_cli* *cli, uint16_t addr, struct *bt_mesh_priv_beacon* *priv_beacon)

Optional callback for Private Beacon Status message.

Handles received Private Beacon Status messages from a Private Beacon server.

Param cli
Private Beacon client context.

Param addr
Address of the sender.

Param priv_beacon
Mesh Private Beacon state received from the server.

void (*priv_gatt_proxy_status)(struct *bt_mesh_priv_beacon_cli* *cli, uint16_t addr, uint8_t gatt_proxy)

Optional callback for Private GATT Proxy Status message.

Handles received Private GATT Proxy Status messages from a Private Beacon server.

Param cli
Private Beacon client context.

Param addr
Address of the sender.

Param gatt_proxy
Private GATT Proxy state received from the server.

void (*priv_node_id_status)(struct *bt_mesh_priv_beacon_cli* *cli, uint16_t addr, struct *bt_mesh_priv_node_id* *priv_node_id)

Optional callback for Private Node Identity Status message.

Handles received Private Node Identity Status messages from a Private Beacon server.

Param cli
Private Beacon client context.

Param addr
Address of the sender.

Param priv_node_id
Private Node Identity state received from the server.

struct bt_mesh_priv_beacon_cli

#include <priv_beacon_cli.h> Mesh Private Beacon Client model.

Public Members

const struct *bt_mesh_priv_beacon_cli_cb* *cb

Optional callback for Private Beacon Client Status messages.

Private Beacon Server The Private Beacon Server model is a foundation model defined by the Bluetooth mesh specification. It is enabled with CONFIG_BT_MESH_PRIV_BEACON_SRV option.

The Private Beacon Server model is introduced in the Bluetooth Mesh Protocol Specification version 1.1, and controls the mesh node's Private Beacon state, Private GATT Proxy state and Private Node Identity state.

The Private Beacons feature adds privacy to the different Bluetooth Mesh beacons by periodically randomizing the beacon input data. This protects the mesh node from being tracked by devices outside the mesh network, and hides the network's IV index, IV update and the Key Refresh state. The Private Beacon Server must be instantiated for the device to support sending of the private beacons, but the node will process received private beacons without it.

The Private Beacon Server does not have an API of its own, but relies on a *Private Beacon Client* to control it. The Private Beacon Server model only accepts messages encrypted with the node's device key.

The application can configure the initial parameters of the Private Beacon Server model through the `bt_mesh_priv_beacon_srv` instance passed to *BT_MESH_MODEL_PRIV_BEACON_SRV*. Note that if the mesh node stored changes to this configuration in the settings subsystem, the initial values may be overwritten upon loading.

If present, the Private Beacon Server model must only be instantiated on the primary element.

API reference

group `bt_mesh_priv_beacon_srv`

Defines

`BT_MESH_MODEL_PRIV_BEACON_SRV`

Private Beacon Server model composition data entry.

Remote Provisioning Client The Remote Provisioning Client model is a foundation model defined by the Bluetooth mesh specification. It is enabled with the `CONFIG_BT_MESH_RPR_CLI` option.

The Remote Provisioning Client model is introduced in the Bluetooth Mesh Protocol Specification version 1.1. This model provides functionality to remotely provision devices into a mesh network, and perform Node Provisioning Protocol Interface procedures by interacting with mesh nodes that support the *Remote Provisioning Server* model.

The Remote Provisioning Client model communicates with a Remote Provisioning Server model using the device key of the node containing the target Remote Provisioning Server model instance.

If present, the Remote Provisioning Client model must be instantiated on the primary element.

Scanning The scanning procedure is used to scan for unprovisioned devices located nearby the Remote Provisioning Server. The Remote Provisioning Client starts a scan procedure by using the *bt_mesh_rpr_scan_start()* call:

```
static void rpr_scan_report(struct bt_mesh_rpr_cli *cli,
            const struct bt_mesh_rpr_node *srv,
            struct bt_mesh_rpr_unprov *unprov,
            struct net_buf_simple *adv_data)
{

}

struct bt_mesh_rpr_cli rpr_cli = {
   .scan_report = rpr_scan_report,
};

const struct bt_mesh_rpr_node srv = {
```

(continues on next page)

(continued from previous page)

```
    .addr = 0x0004,
    .net_idx = 0,
    .ttl = BT_MESH_TTL_DEFAULT,
};

struct bt_mesh_rpr_scan_status status;
uint8_t *uuid = NULL;
uint8_t timeout = 10;
uint8_t max_devs = 3;

bt_mesh_rpr_scan_start(&rpr_cli, &srv, uuid, timeout, max_devs, &status);
```

The above example shows pseudo code for starting a scan procedure on the target Remote Provisioning Server node. This procedure will start a ten-second, multiple-device scanning where the generated scan report will contain a maximum of three unprovisioned devices. If the UUID argument was specified, the same procedure would only scan for the device with the corresponding UUID. After the procedure completes, the server sends the scan report that will be handled in the client's *bt_mesh_rpr_cli.scan_report* callback.

Additionally, the Remote Provisioning Client model also supports extended scanning with the *bt_mesh_rpr_scan_start_ext()* call. Extended scanning supplements regular scanning by allowing the Remote Provisioning Server to report additional data for a specific device. The Remote Provisioning Server will use active scanning to request a scan response from the unprovisioned device if it is supported by the unprovisioned device.

Provisioning The Remote Provisioning Client starts a provisioning procedure by using the *bt_mesh_provision_remote()* call:

```
struct bt_mesh_rpr_cli rpr_cli;

const struct bt_mesh_rpr_node srv = {
    .addr = 0x0004,
    .net_idx = 0,
    .ttl = BT_MESH_TTL_DEFAULT,
};

uint8_t uuid[16] = { 0xaa };
uint16_t addr = 0x0006;
uint16_t net_idx = 0;

bt_mesh_provision_remote(&rpr_cli, &srv, uuid, net_idx, addr);
```

The above example shows pseudo code for remotely provisioning a device through a Remote Provisioning Server node. This procedure will attempt to provision the device with the corresponding UUID, and assign the address 0x0006 to its primary element using the network key located at index zero.

Note

During the remote provisioning, the same *bt_mesh_prov* callbacks are triggered as for ordinary provisioning. See section *Provisioning* for further details.

Re-provisioning In addition to scanning and provisioning functionality, the Remote Provisioning Client also provides means to reconfigure node addresses, device keys and Composition Data on devices that support the *Remote Provisioning Server* model. This is provided through the Node Provisioning Protocol Interface (NPPI) which supports the following three procedures:

- Device Key Refresh procedure: Used to change the device key of the Target node without a need to reconfigure the node.
- Node Address Refresh procedure: Used to change the node's device key and unicast address.
- Node Composition Refresh procedure: Used to change the device key of the node, and to add or delete models or features of the node.

The three NPPI procedures can be initiated with the *bt_mesh_reprovision_remote()* call:

```
struct bt_mesh_rpr_cli rpr_cli;
struct bt_mesh_rpr_node srv = {
    .addr = 0x0006,
    .net_idx = 0,
    .ttl = BT_MESH_TTL_DEFAULT,
};

bool composition_changed = false;
uint16_t new_addr = 0x0009;

bt_mesh_reprovision_remote(&rpr_cli, &srv, new_addr, composition_changed);
```

The above example shows pseudo code for triggering a Node Address Refresh procedure on the Target node. The specific procedure is not chosen directly, but rather through the other parameters that are inputted. In the example we can see that the current unicast address of the Target is 0x0006, while the new address is set to 0x0009. If the two addresses were the same, and the `composition_changed` flag was set to true, this code would instead trigger a Node Composition Refresh procedure. If the two addresses were the same, and the `composition_changed` flag was set to false, this code would trigger a Device Key Refresh procedure.

API reference

group **bt_mesh_rpr_cli**

Defines

BT_MESH_RPR_SCAN_MAX_DEVS_ANY

Special value for the `max_devs` parameter of *bt_mesh_rpr_scan_start*.

Tells the Remote Provisioning Server not to put restrictions on the max number of devices reported to the Client.

BT_MESH_MODEL_RPR_CLI(_cli)

Remote Provisioning Client model composition data entry.

Parameters

- **_cli** – Pointer to a *Remote Provisioning Client model* instance.

Functions

int **bt_mesh_rpr_scan_caps_get**(struct *bt_mesh_rpr_cli* *cli, const struct bt_mesh_rpr_node *srv, struct *bt_mesh_rpr_caps* *caps)

Get scanning capabilities of Remote Provisioning Server.

Parameters

- **cli** – Remote Provisioning Client.
- **srv** – Remote Provisioning Server.

- **caps** – Capabilities response buffer.

Returns
0 on success, or (negative) error code otherwise.

int **bt_mesh_rpr_scan_get**(struct *bt_mesh_rpr_cli* *cli, const struct bt_mesh_rpr_node *srv, struct *bt_mesh_rpr_scan_status* *status)

Get current scanning state of Remote Provisioning Server.

Parameters

- **cli** – Remote Provisioning Client.
- **srv** – Remote Provisioning Server.
- **status** – Scan status response buffer.

Returns
0 on success, or (negative) error code otherwise.

int **bt_mesh_rpr_scan_start**(struct *bt_mesh_rpr_cli* *cli, const struct bt_mesh_rpr_node *srv, const uint8_t uuid[16], uint8_t timeout, uint8_t max_devs, struct *bt_mesh_rpr_scan_status* *status)

Start scanning for unprovisioned devices.

Tells the Remote Provisioning Server to start scanning for unprovisioned devices. The Server will report back the results through the *bt_mesh_rpr_cli::scan_report* callback.

Use the `uuid` parameter to scan for a specific device, or leave it as NULL to report all unprovisioned devices.

The Server will ignore duplicates, and report up to `max_devs` number of devices. Requesting a `max_devs` number that's higher than the Server's capability will result in an error.

Parameters

- **cli** – Remote Provisioning Client.
- **srv** – Remote Provisioning Server.
- **uuid** – Device UUID to scan for, or NULL to report all devices.
- **timeout** – Scan timeout in seconds. Must be at least 1 second.
- **max_devs** – Max number of devices to report, or 0 to report as many as possible.
- **status** – Scan status response buffer.

Returns
0 on success, or (negative) error code otherwise.

int **bt_mesh_rpr_scan_start_ext**(struct *bt_mesh_rpr_cli* *cli, const struct bt_mesh_rpr_node *srv, const uint8_t uuid[16], uint8_t timeout, const uint8_t *ad_types, size_t ad_count)

Start extended scanning for unprovisioned devices.

Extended scanning supplements regular unprovisioned scanning, by allowing the Server to report additional data for a specific device. The Remote Provisioning Server will use active scanning to request a scan response from the unprovisioned device, if supported. If no UUID is provided, the Server will report a scan on its own OOB information and advertising data.

Use the ad_types array to specify which AD types to include in the scan report. Some AD types invoke special behavior:

- *BT_DATA_NAME_COMPLETE* Will report both the complete and the shortened name.

- *BT_DATA_URI* If the unprovisioned beacon contains a URI hash, the Server will extend the scanning to include packets other than the scan response, to look for URIs matching the URI hash. Only matching URIs will be reported.

The following AD types should not be used:

- *BT_DATA_NAME_SHORTENED*
- *BT_DATA_UUID16_SOME*
- *BT_DATA_UUID32_SOME*
- *BT_DATA_UUID128_SOME*

Additionally, each AD type should only occur once.

Parameters

- **cli** – Remote Provisioning Client.
- **srv** – Remote Provisioning Server.
- **uuid** – Device UUID to start extended scanning for, or NULL to scan the remote server.
- **timeout** – Scan timeout in seconds. Valid values from BT_MESH_RPR_EXT_SCAN_TIME_MIN to BT_MESH_RPR_EXT_SCAN_TIME_MAX. Ignored if UUID is NULL.
- **ad_types** – List of AD types to include in the scan report. Must contain 1 to CONFIG_BT_MESH_RPR_AD_TYPES_MAX entries.
- **ad_count** – Number of AD types in `ad_types`.

Returns

0 on success, or (negative) error code otherwise.

int **bt_mesh_rpr_scan_stop**(struct *bt_mesh_rpr_cli* *cli, const struct bt_mesh_rpr_node *srv, struct *bt_mesh_rpr_scan_status* *status)

Stop any ongoing scanning on the Remote Provisioning Server.

Parameters

- **cli** – Remote Provisioning Client.
- **srv** – Remote Provisioning Server.
- **status** – Scan status response buffer.

Returns

0 on success, or (negative) error code otherwise.

int **bt_mesh_rpr_link_get**(struct *bt_mesh_rpr_cli* *cli, const struct bt_mesh_rpr_node *srv, struct bt_mesh_rpr_link *rsp)

Get the current link status of the Remote Provisioning Server.

Parameters

- **cli** – Remote Provisioning Client.
- **srv** – Remote Provisioning Server.
- **rsp** – Link status response buffer.

Returns

0 on success, or (negative) error code otherwise.

int **bt_mesh_rpr_link_close**(struct *bt_mesh_rpr_cli* *cli, const struct bt_mesh_rpr_node *srv, struct bt_mesh_rpr_link *rsp)

Close any open link on the Remote Provisioning Server.

Parameters

- `cli` – Remote Provisioning Client.
- `srv` – Remote Provisioning Server.
- `rsp` – Link status response buffer.

Returns
0 on success, or (negative) error code otherwise.

int32_t **bt_mesh_rpr_cli_timeout_get**(void)

Get the current transmission timeout value.

Returns
The configured transmission timeout in milliseconds.

void **bt_mesh_rpr_cli_timeout_set**(int32_t timeout)

Set the transmission timeout value.

The transmission timeout controls the amount of time the Remote Provisioning Client models will wait for a response from the Server.

Parameters

- `timeout` – The new transmission timeout.

struct **bt_mesh_rpr_scan_status**

#include <rpr_cli.h> Scan status response.

Public Members

enum bt_mesh_rpr_status **status**

Current scan status.

enum bt_mesh_rpr_scan **scan**

Current scan state.

uint8_t **max_devs**

Max number of devices to report in current scan.

uint8_t **timeout**

Seconds remaining of the scan.

struct **bt_mesh_rpr_caps**

#include <rpr_cli.h> Remote Provisioning Server scanning capabilities.

Public Members

uint8_t **max_devs**

Max number of scannable devices.

bool **active_scan**

Supports active scan.

struct **bt_mesh_rpr_cli**

#include <rpr_cli.h> Remote Provisioning Client model instance.

Public Members

void (***scan_report**)(struct *bt_mesh_rpr_cli* *cli, const struct bt_mesh_rpr_node *srv, struct bt_mesh_rpr_unprov *unprov, struct *net_buf_simple* *adv_data)

Scan report callback.

Param cli
Remote Provisioning Client.

Param srv
Remote Provisioning Server.

Param unprov
Unprovisioned device.

Param adv_data
Advertisement data for the unprovisioned device, or NULL if extended scanning hasn't been enabled. An empty buffer indicates that the extended scanning finished without collecting additional information.

Remote Provisioning Server The Remote Provisioning Server model is a foundation model defined by the Bluetooth mesh specification. It is enabled with the `CONFIG_BT_MESH_RPR_SRV` option.

The Remote Provisioning Server model is introduced in the Bluetooth Mesh Protocol Specification version 1.1, and is used to support the functionality of remotely provisioning devices into a mesh network.

The Remote Provisioning Server does not have an API of its own, but relies on a *Remote Provisioning Client* to control it. The Remote Provisioning Server model only accepts messages encrypted with the node's device key.

If present, the Remote Provisioning Server model must be instantiated on the primary element.

Note that after refreshing the device key, node address or Composition Data through a Node Provisioning Protocol Interface (NPPI) procedure, the *`bt_mesh_prov.reprovisioned`* callback is triggered. See section *Remote Provisioning Client* for further details.

Limitations The following limitations apply to Remote Provisioning Server model:

- Provisioning of unprovisioned device using PB-GATT is not supported.
- All Node Provisioning Protocol Interface (NPPI) procedures are supported. However, if the composition data of a device gets changed after device firmware update (see *firmware effect*), it is not possible for the device to remain provisioned. The device should be unprovisioned if its composition data is expected to change.

API reference

group **bt_mesh_rpr_srv**

Defines

`BT_MESH_MODEL_RPR_SRV`

Remote Provisioning Server model composition data entry.

SAR Configuration Client The SAR Configuration Client model is a foundation model defined by the Bluetooth Mesh specification. It is an optional model, enabled with the `CONFIG_BT_MESH_SAR_CFG_CLI` configuration option.

The SAR Configuration Client model is introduced in the Bluetooth Mesh Protocol Specification version 1.1, and it supports the configuration of the lower transport layer behavior of a node that supports the *SAR Configuration Server* model.

The model can send messages to query or change the states supported by the SAR Configuration Server (SAR Transmitter and SAR Receiver) using SAR Configuration messages.

The SAR Transmitter procedure is used to determine and configure the SAR Transmitter state of a SAR Configuration Server. Function calls *`bt_mesh_sar_cfg_cli_transmitter_get()`* and *`bt_mesh_sar_cfg_cli_transmitter_set()`* are used to get and set the SAR Transmitter state of the Target node respectively.

The SAR Receiver procedure is used to determine and configure the SAR Receiver state of a SAR Configuration Server. Function calls *`bt_mesh_sar_cfg_cli_receiver_get()`* and *`bt_mesh_sar_cfg_cli_receiver_set()`* are used to get and set the SAR Receiver state of the Target node respectively.

For more information about the two states, see *SAR states*.

An element can send any SAR Configuration Client message at any time to query or change the states supported by the SAR Configuration Server model of a peer node. The SAR Configuration Client model only accepts messages encrypted with the device key of the node supporting the SAR Configuration Server model.

If present, the SAR Configuration Client model must only be instantiated on the primary element.

API reference

group `bt_mesh_sar_cfg_cli`

Bluetooth Mesh.

Defines

`BT_MESH_MODEL_SAR_CFG_CLI`(_cli)

SAR Configuration Client model composition data entry.

Parameters

- **`_cli`** – **[in]** Pointer to a *Bluetooth Mesh SAR Configuration Client Model* instance.

Functions

int `bt_mesh_sar_cfg_cli_transmitter_get`(uint16_t net_idx, uint16_t addr, struct bt_mesh_sar_tx *rsp)

Get the SAR Transmitter state of the target node.

Parameters

- **`net_idx`** – Network index to encrypt with.

- **addr** – Target node address.
- **rsp** – Status response parameter.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_sar_cfg_cli_transmitter_set**(uint16_t net_idx, uint16_t addr, const struct bt_mesh_sar_tx *set, struct bt_mesh_sar_tx *rsp)

Set the SAR Transmitter state of the target node.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **set** – New SAR Transmitter state to set on the target node.
- **rsp** – Status response parameter.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_sar_cfg_cli_receiver_get**(uint16_t net_idx, uint16_t addr, struct bt_mesh_sar_rx *rsp)

Get the SAR Receiver state of the target node.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **rsp** – Status response parameter.

Returns
0 on success, or (negative) error code on failure.

int **bt_mesh_sar_cfg_cli_receiver_set**(uint16_t net_idx, uint16_t addr, const struct bt_mesh_sar_rx *set, struct bt_mesh_sar_rx *rsp)

Set the SAR Receiver state of the target node.

Parameters

- **net_idx** – Network index to encrypt with.
- **addr** – Target node address.
- **set** – New SAR Receiver state to set on the target node.
- **rsp** – Status response parameter.

Returns
0 on success, or (negative) error code on failure.

int32_t **bt_mesh_sar_cfg_cli_timeout_get**(void)

Get the current transmission timeout value.

Returns
The configured transmission timeout in milliseconds.

void **bt_mesh_sar_cfg_cli_timeout_set**(int32_t timeout)

Set the transmission timeout value.

Parameters

- **timeout** – The new transmission timeout.

struct **bt_mesh_sar_cfg_cli**

#include <sar_cfg_cli.h> Mesh SAR Configuration Client Model Context.

Public Members

const struct *bt_mesh_model* ***model**

Access model pointer.

SAR Configuration Server The SAR Configuration Server model is a foundation model defined by the Bluetooth Mesh specification. It is an optional model, enabled with the `CONFIG_BT_MESH_SAR_CFG_SRV` configuration option.

The SAR Configuration Server model is introduced in the Bluetooth Mesh Protocol Specification version 1.1, and it supports the configuration of the *segmentation and reassembly (SAR)* behavior of a Bluetooth Mesh node. The model defines a set of states and messages for the SAR configuration.

The SAR Configuration Server model defines two states, SAR Transmitter state and SAR Receiver state. For more information about the two states, see *SAR states*.

The model also supports the SAR Transmitter and SAR Receiver get and set messages.

The SAR Configuration Server model does not have an API of its own, but relies on a *SAR Configuration Client* to control it. The SAR Configuration Server model only accepts messages encrypted with the node's device key.

If present, the SAR Configuration Server model must only be instantiated on the primary element.

API reference

group **bt_mesh_sar_cfg_srv**

Bluetooth Mesh.

Defines

`BT_MESH_MODEL_SAR_CFG_SRV`

Transport SAR Configuration Server model composition data entry.

Solicitation PDU RPL Configuration Client The Solicitation PDU RPL Configuration Client model is a foundation model defined by the Bluetooth mesh specification. The model is optional, and is enabled through the `CONFIG_BT_MESH_SOL_PDU_RPL_CLI` option.

The Solicitation PDU RPL Configuration Client model was introduced in the Bluetooth Mesh Protocol Specification version 1.1, and supports the functionality of removing addresses from the solicitation replay protection list (SRPL) of a node that supports the *Solicitation PDU RPL Configuration Server* model.

The Solicitation PDU RPL Configuration Client model communicates with a Solicitation PDU RPL Configuration Server model using the application keys configured by the Configuration Client.

If present, the Solicitation PDU RPL Configuration Client model must only be instantiated on the primary element.

Configurations The Solicitation PDU RPL Configuration Client model behavior can be configured with the transmission timeout option `CONFIG_BT_MESH_SOL_PDU_RPL_CLI_TIMEOUT`. The `CONFIG_BT_MESH_SOL_PDU_RPL_CLI_TIMEOUT` controls how long the Solicitation PDU RPL Configuration Client waits for a response message to arrive in milliseconds. This value can be changed at runtime using *bt_mesh_sol_pdu_rpl_cli_timeout_set()*.

API reference

group **bt_mesh_sol_pdu_rpl_cli**

Defines

BT_MESH_MODEL_SOL_PDU_RPL_CLI(cli_data)

Solicitation PDU RPL Client model composition data entry.

Functions

int **bt_mesh_sol_pdu_rpl_clear**(struct *bt_mesh_msg_ctx* *ctx, uint16_t range_start, uint8_t range_len, uint16_t *start_rsp, uint8_t *len_rsp)

Remove entries from Solicitation PDU RPL of addresses in given range.

This method can be used asynchronously by setting `start_rsp` or `len_rsp` as NULL. This way the method will not wait for response and will return immediately after sending the command.

To process the response arguments of an async method, register the `srpl_status` callback in *bt_mesh_sol_pdu_rpl_cli* struct.

Parameters

- **ctx** – Message context for the message.
- **range_start** – Start of Unicast address range.
- **range_len** – Length of Unicast address range. Valid values are 0x00 and 0x02 to 0xff.
- **start_rsp** – Range start response buffer.
- **len_rsp** – Range length response buffer.

Returns

0 on success, or (negative) error code otherwise.

int **bt_mesh_sol_pdu_rpl_clear_unack**(struct *bt_mesh_msg_ctx* *ctx, uint16_t range_start, uint8_t range_len)

Remove entries from Solicitation PDU RPL of addresses in given range (unacked).

Parameters

- **ctx** – Message context for the message.
- **range_start** – Start of Unicast address range.
- **range_len** – Length of Unicast address range. Valid values are 0x00 and 0x02 to 0xff.

Returns

0 on success, or (negative) error code otherwise.

void **bt_mesh_sol_pdu_rpl_cli_timeout_set**(int32_t timeout)

Set the transmission timeout value.

Parameters

- **timeout** – The new transmission timeout in milliseconds.

struct **bt_mesh_sol_pdu_rpl_cli**

#include <*sol_pdu_rpl_cli.h*> Solicitation PDU RPL Client Model Context.

Public Members

const struct *bt_mesh_model* ***model**

Solicitation PDU RPL model entry pointer.

void (***srpl_status**)(struct *bt_mesh_sol_pdu_rpl_cli* *cli, uint16_t addr, uint16_t range_start, uint8_t range_length)

Optional callback for Solicitation PDU RPL Status messages.

Handles received Solicitation PDU RPL Status messages from a Solicitation PDU RPL server.The `start` param represents the start of range that server has cleared. The `length` param represents length of range cleared by server.

Param cli
Solicitation PDU RPL client that received the status message.

Param addr
Address of the sender.

Param range_start
Range start value.

Param range_length
Range length value.

Solicitation PDU RPL Configuration Server The Solicitation PDU RPL Configuration Server model is a foundation model defined by the Bluetooth mesh specification. The model is enabled if the node has the *On-Demand Private Proxy Server* enabled.

The Solicitation PDU RPL Configuration Server model was introduced in the Bluetooth Mesh Protocol Specification version 1.1, and manages the Solicitation Replay Protection List (SRPL) saved on the device. The SRPL is used to reject Solicitation PDUs that are already processed by a node. When a valid Solicitation PDU message is successfully processed by a node, the SSRC field and SSEQ field of the message are stored in the node's SRPL.

The Solicitation PDU RPL Configuration Server does not have an API of its own, and relies on a *Solicitation PDU RPL Configuration Client* to control it. The model only accepts messages encrypted with an application key as configured by the Configuration Client.

If present, the Solicitation PDU RPL Configuration Server model must only be instantiated on the primary element.

Configurations For the Solicitation PDU RPL Configuration Server model, the `CONFIG_BT_MESH_PROXY_SRPL_SIZE` option can be configured to set the size of the SRPL.

API reference

group **bt_mesh_sol_pdu_rpl_srv**

Defines

BT_MESH_MODEL_SOL_PDU_RPL_SRV

Solicitation PDU RPL Server model composition data entry.

Model specification models In addition to the foundation models defined in the Bluetooth Mesh specification, the Bluetooth Mesh Model Specification defines several models, some of which are implemented in Zephyr:

BLOB Transfer models The Binary Large Object (BLOB) Transfer models implement the Bluetooth Mesh Binary Large Object Transfer Model specification version 1.0 and provide functionality for sending large binary objects from a single source to many Target nodes over the Bluetooth Mesh network. It is the underlying transport method for the *Device Firmware Update (DFU)*, but may be used for other object transfer purposes. The implementation is in experimental state.

The BLOB Transfer models support transfers of continuous binary objects of up to 4 GB (2^{32} bytes). The BLOB transfer protocol has built-in recovery procedures for packet losses, and sets up checkpoints to ensure that all targets have received all the data before moving on. Data transfer order is not guaranteed.

BLOB transfers are constrained by the transfer speed and reliability of the underlying mesh network. Under ideal conditions, the BLOBs can be transferred at a rate of up to 1 kbps, allowing a 100 kB BLOB to be transferred in 10-15 minutes. However, network conditions, transfer capabilities and other limiting factors can easily degrade the data rate by several orders of magnitude. Tuning the parameters of the transfer according to the application and network configuration, as well as scheduling it to periods with low network traffic, will offer significant improvements on the speed and reliability of the protocol. However, achieving transfer rates close to the ideal rate is unlikely in actual deployments.

There are two BLOB Transfer models:

BLOB Transfer Server The Binary Large Object (BLOB) Transfer Server model implements reliable receiving of large binary objects. It serves as the backend of the *Firmware Update Server*, but can also be used for receiving other binary images.

BLOBs As described in *BLOB Transfer models*, the binary objects transferred by the BLOB Transfer models are divided into blocks, which are divided into chunks. As the transfer is controlled by the BLOB Transfer Client model, the BLOB Transfer Server must allow blocks to come in any order. The chunks within a block may also come in any order, but all chunks in a block must be received before the next block is started.

The BLOB Transfer Server keeps track of the received blocks and chunks, and will process each block and chunk only once. The BLOB Transfer Server also ensures that any missing chunks are resent by the BLOB Transfer Client.

Usage The BLOB Transfer Server is instantiated on an element with a set of event handler callbacks:

```
static const struct bt_mesh_blob_srv_cb blob_cb = {
    /* Callbacks */
};

static struct bt_mesh_blob_srv blob_srv = {
    .cb = &blob_cb,
```

(continues on next page)

(continued from previous page)

```
};

static const struct bt_mesh_model models[] = {
    BT_MESH_MODEL_BLOB_SRV(&blob_srv),
};
```

A BLOB Transfer Server is capable of receiving a single BLOB transfer at a time. Before the BLOB Transfer Server can receive a transfer, it must be prepared by the user. The transfer ID must be passed to the BLOB Transfer Server through the *bt_mesh_blob_srv_recv()* function before the transfer is started by the BLOB Transfer Client. The ID must be shared between the BLOB Transfer Client and the BLOB Transfer Server through some higher level procedure, like a vendor specific transfer management model.

Once the transfer has been set up on the BLOB Transfer Server, it's ready for receiving the BLOB. The application is notified of the transfer progress through the event handler callbacks, and the BLOB data is sent to the BLOB stream.

The interaction between the BLOB Transfer Server, BLOB stream and application is shown below:

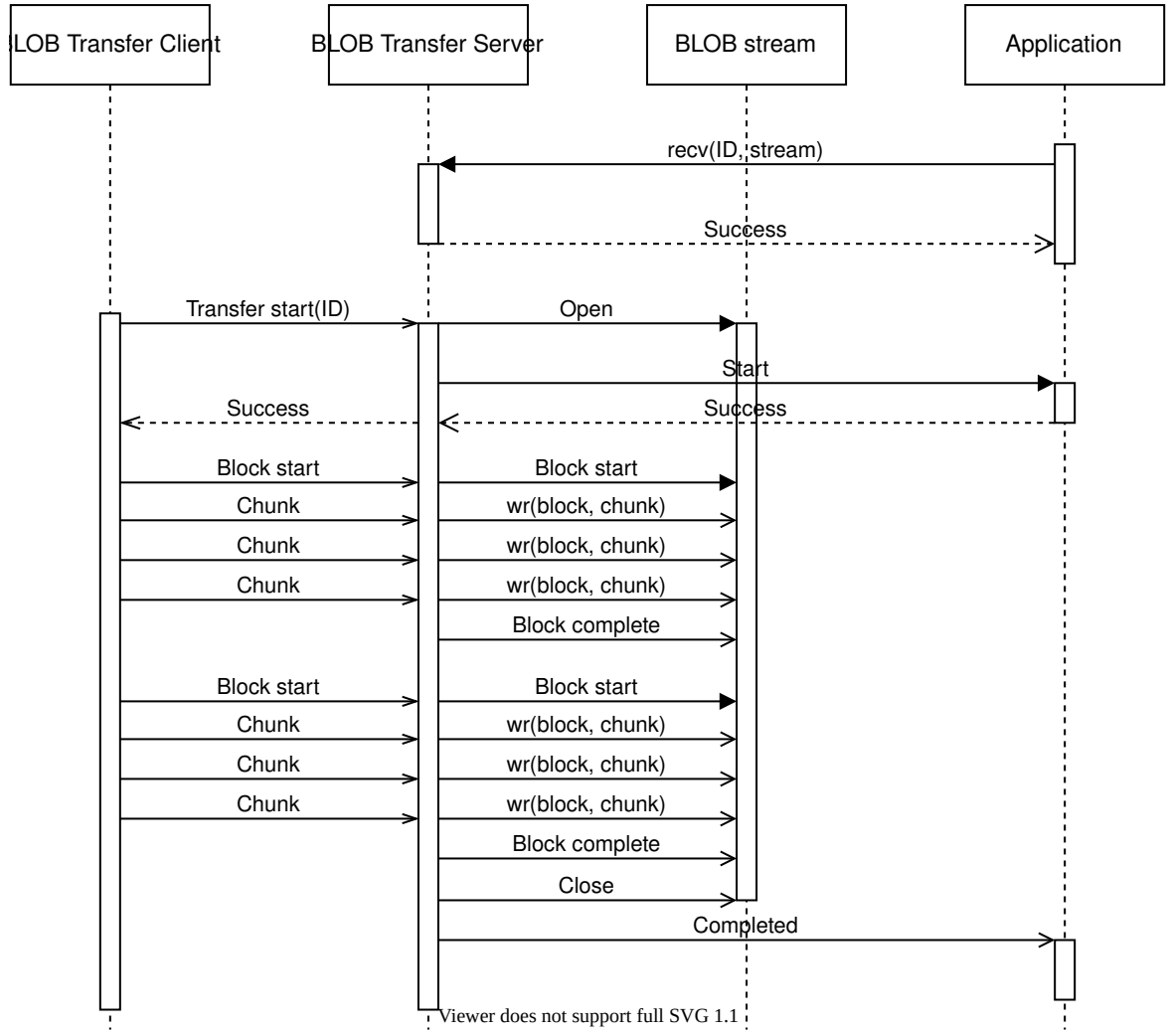


Fig. 8: BLOB Transfer Server model interaction

Transfer suspension The BLOB Transfer Server keeps a running timer during the transfer, that is reset on every received message. If the BLOB Transfer Client does not send a message before the transfer timer expires, the transfer is suspended by the BLOB Transfer Server.

The BLOB Transfer Server notifies the user of the suspension by calling the *suspended* callback. If the BLOB Transfer Server is in the middle of receiving a block, this block is discarded.

The BLOB Transfer Client may resume a suspended transfer by starting a new block transfer. The BLOB Transfer Server notifies the user by calling the *resume* callback.

Transfer recovery The state of the BLOB transfer is stored persistently. If a reboot occurs, the BLOB Transfer Server will attempt to recover the transfer. When the Bluetooth Mesh subsystem is started (for instance by calling *bt_mesh_init()*), the BLOB Transfer Server will check for aborted transfers, and call the *recover* callback if there is any. In the recover callback, the user must provide a BLOB stream to use for the rest of the transfer. If the recover callback doesn't return successfully or does not provide a BLOB stream, the transfer is abandoned. If no recover callback is implemented, transfers are always abandoned after a reboot.

After a transfer is successfully recovered, the BLOB Transfer Server enters the suspended state. It will stay suspended until the BLOB Transfer Client resumes the transfer, or the user cancels it.

Note

The BLOB Transfer Client sending the transfer must support transfer recovery for the transfer to complete. If the BLOB Transfer Client has already given up the transfer, the BLOB Transfer Server will stay suspended until the application calls *bt_mesh_blob_srv_cancel()*.

API reference

group `bt_mesh_blob_srv`

Defines

`BT_MESH_BLOB_BLOCKS_MAX`

Max number of blocks in a single transfer.

`BT_MESH_MODEL_BLOB_SRV`(_srv)

BLOB Transfer Server model composition data entry.

Parameters

- `_srv` – Pointer to a *Bluetooth Mesh BLOB Transfer Server model API* instance.

Functions

int `bt_mesh_blob_srv_recv`(struct *bt_mesh_blob_srv* *srv, uint64_t id, const struct *bt_mesh_blob_io* *io, uint8_t ttl, uint16_t timeout_base)

Prepare BLOB Transfer Server for an incoming transfer.

Before a BLOB Transfer Server can receive a transfer, the transfer must be prepared through some application level mechanism. The BLOB Transfer Server will only accept incoming transfers with a matching BLOB ID.

Parameters

- `srv` – BLOB Transfer Server instance.
- `id` – BLOB ID to accept.

- `io` – BLOB stream to write the incoming BLOB to.
- `ttl` – Time to live value to use in responses to the BLOB Transfer Client.
- `timeout_base` – Extra time for the Client to respond in addition to the base 10 seconds, in 10-second increments.

Returns
0 on success, or (negative) error code on failure.

int `bt_mesh_blob_srv_cancel`(struct *bt_mesh_blob_srv* *srv)

Cancel the current BLOB transfer.

Tells the BLOB Transfer Client to drop this device from the list of Targets for the current transfer. Note that the client may continue sending the transfer to other Targets.

Parameters

- `srv` – BLOB Transfer Server instance.

Returns
0 on success, or (negative) error code on failure.

bool `bt_mesh_blob_srv_is_busy`(const struct *bt_mesh_blob_srv* *srv)

Get the current state of the BLOB Transfer Server.

Parameters

- `srv` – BLOB Transfer Server instance.

Returns
true if the BLOB Transfer Server is currently participating in a transfer, false otherwise.

uint8_t `bt_mesh_blob_srv_progress`(const struct *bt_mesh_blob_srv* *srv)

Get the current progress of the active transfer in percent.

Parameters

- `srv` – BLOB Transfer Server instance.

Returns
The current transfer progress, or 0 if no transfer is active.

struct `bt_mesh_blob_srv_cb`

#include <blob_srv.h> BLOB Transfer Server model event handlers.

All callbacks are optional.

Public Members

int (*`start`)(struct *bt_mesh_blob_srv* *srv, struct *bt_mesh_msg_ctx* *ctx, struct *bt_mesh_blob_xfer* *xfer)

Transfer start callback.

Called when the transfer has started with the prepared BLOB ID.

Param srv
BLOB Transfer Server instance.

Param ctx
Message context for the incoming start message. The entire transfer will be sent from the same source address.

Param xfer
Transfer parameters.

Return
0 on success, or (negative) error code to reject the transfer.

void (*end)(struct *bt_mesh_blob_srv* *srv, uint64_t id, bool success)

Transfer end callback.

Called when the transfer ends, either because it was cancelled, or because it finished successfully. A new transfer may be prepared.

> **Note**
>
> The transfer may end before it's started if the start parameters are invalid.

Param srv
BLOB Transfer Server instance.
Param id
BLOB ID of the cancelled transfer.
Param success
Whether the transfer was successful.

void (*suspended)(struct *bt_mesh_blob_srv* *srv)

Transfer suspended callback.

Called if the Server timed out while waiting for a transfer packet. A suspended transfer may resume later from the start of the current block. Any received chunks in the current block should be discarded, they will be received again if the transfer resumes.

The transfer will call `resumed` again when resuming.

> **Note**
>
> The BLOB Transfer Server does not run a timer in the suspended state, and it's up to the application to determine whether the transfer should be permanently cancelled. Without interaction, the transfer will be suspended indefinitely, and the BLOB Transfer Server will not accept any new transfers.

Param srv
BLOB Transfer Server instance.

void (*resume)(struct *bt_mesh_blob_srv* *srv)

Transfer resume callback.

Called if the transfer is resumed after being suspended.
Param srv
BLOB Transfer Server instance.

int (*recover)(struct *bt_mesh_blob_srv* *srv, struct *bt_mesh_blob_xfer* *xfer, const struct *bt_mesh_blob_io* **io)

Transfer recovery callback.

Called when the Bluetooth Mesh subsystem is started if the device is rebooted in the middle of a transfer.

Transfers will not be resumed after a reboot if this callback is not defined.
Param srv
BLOB Transfer Server instance.

Param xfer
Transfer to resume.
Param io
BLOB stream return parameter. Must be set to a valid BLOB stream by the callback.
Return
0 on success, or (negative) error code to abandon the transfer.

struct **bt_mesh_blob_srv**
#include <blob_srv.h> BLOB Transfer Server instance.

Public Members

const struct *bt_mesh_blob_srv_cb* ***cb**
Event handler callbacks.

struct **bt_mesh_blob_srv_state**
#include <blob_srv.h>

BLOB Transfer Client The Binary Large Object (BLOB) Transfer Client is the sender of the BLOB transfer. It supports sending BLOBs of any size to any number of Target nodes, in both Push BLOB Transfer Mode and Pull BLOB Transfer Mode.

Usage

Initialization The BLOB Transfer Client is instantiated on an element with a set of event handler callbacks:

```
static const struct bt_mesh_blob_cli_cb blob_cb = {
      /* Callbacks */
};

static struct bt_mesh_blob_cli blob_cli = {
      .cb = &blob_cb,
};

static const struct bt_mesh_model models[] = {
      BT_MESH_MODEL_BLOB_CLI(&blob_cli),
};
```

Transfer context Both the transfer capabilities retrieval procedure and the BLOB transfer uses an instance of a *bt_mesh_blob_cli_inputs* to determine how to perform the transfer. The BLOB Transfer Client Inputs structure must at least be initialized with a list of targets, an application key and a time to live (TTL) value before it is used in a procedure:

```
static struct bt_mesh_blob_target targets[3] = {
        { .addr = 0x0001 },
        { .addr = 0x0002 },
        { .addr = 0x0003 },
};
static struct bt_mesh_blob_cli_inputs inputs = {
        .app_idx = MY_APP_IDX,
```

(continues on next page)

(continued from previous page)

```
        .ttl = BT_MESH_TTL_DEFAULT,
};

sys_slist_init(&inputs.targets);
sys_slist_append(&inputs.targets, &targets[0].n);
sys_slist_append(&inputs.targets, &targets[1].n);
sys_slist_append(&inputs.targets, &targets[2].n);
```

Note that all BLOB Transfer Servers in the transfer must be bound to the chosen application key.

Group address The application may additionally specify a group address in the context structure. If the group is not *BT_MESH_ADDR_UNASSIGNED*, the messages in the transfer will be sent to the group address, instead of being sent individually to each Target node. Mesh Manager must ensure that all Target nodes having the BLOB Transfer Server model subscribe to this group address.

Using group addresses for transferring the BLOBs can generally increase the transfer speed, as the BLOB Transfer Client sends each message to all Target nodes at the same time. However, sending large, segmented messages to group addresses in Bluetooth Mesh is generally less reliable than sending them to unicast addresses, as there is no transport layer acknowledgment mechanism for groups. This can lead to longer recovery periods at the end of each block, and increases the risk of losing Target nodes. Using group addresses for BLOB transfers will generally only pay off if the list of Target nodes is extensive, and the effectiveness of each addressing strategy will vary heavily between different deployments and the size of the chunks.

Transfer timeout If a Target node fails to respond to an acknowledged message within the BLOB Transfer Client's time limit, the Target node is dropped from the transfer. The application can reduce the chances of this by giving the BLOB Transfer Client extra time through the context structure. The extra time may be set in 10-second increments, up to 182 hours, in addition to the base time of 20 seconds. The wait time scales automatically with the transfer TTL.

Note that the BLOB Transfer Client only moves forward with the transfer in following cases:

- All Target nodes have responded.
- A node has been removed from the list of Target nodes.
- The BLOB Transfer Client times out.

Increasing the wait time will increase this delay.

BLOB transfer capabilities retrieval It is generally recommended to retrieve BLOB transfer capabilities before starting a transfer. The procedure populates the transfer capabilities from all Target nodes with the most liberal set of parameters that allows all Target nodes to participate in the transfer. Any Target nodes that fail to respond, or respond with incompatible transfer parameters, will be dropped.

Target nodes are prioritized according to their order in the list of Target nodes. If a Target node is found to be incompatible with any of the previous Target nodes, for instance by reporting a non-overlapping block size range, it will be dropped. Lost Target nodes will be reported through the *lost_target* callback.

The end of the procedure is signalled through the *caps* callback, and the resulting capabilities can be used to determine the block and chunk sizes required for the BLOB transfer.

BLOB transfer The BLOB transfer is started by calling *bt_mesh_blob_cli_send()* function, which (in addition to the aforementioned transfer inputs) requires a set of transfer parameters and a BLOB stream instance. The transfer parameters include the 64-bit BLOB ID, the BLOB size,

the transfer mode, the block size in logarithmic representation and the chunk size. The BLOB ID is application defined, but must match the BLOB ID the BLOB Transfer Servers have been started with.

The transfer runs until it either completes successfully for at least one Target node, or it is cancelled. The end of the transfer is communicated to the application through the *end* callback. Lost Target nodes will be reported through the *lost_target* callback.

API reference

group `bt_mesh_blob_cli`

Defines

`BT_MESH_MODEL_BLOB_CLI`(_cli)

BLOB Transfer Client model Composition Data entry.

Parameters

- `_cli` – Pointer to a *Bluetooth Mesh BLOB Transfer Client model API* instance.

Enums

enum `bt_mesh_blob_cli_state`

BLOB Transfer Client state.

Values:

enumerator `BT_MESH_BLOB_CLI_STATE_NONE`

No transfer is active.

enumerator `BT_MESH_BLOB_CLI_STATE_CAPS_GET`

Retrieving transfer capabilities.

enumerator `BT_MESH_BLOB_CLI_STATE_START`

Sending transfer start.

enumerator `BT_MESH_BLOB_CLI_STATE_BLOCK_START`

Sending block start.

enumerator `BT_MESH_BLOB_CLI_STATE_BLOCK_SEND`

Sending block chunks.

enumerator `BT_MESH_BLOB_CLI_STATE_BLOCK_CHECK`

Checking block status.

enumerator `BT_MESH_BLOB_CLI_STATE_XFER_CHECK`

Checking transfer status.

enumerator BT_MESH_BLOB_CLI_STATE_CANCEL

Cancelling transfer.

enumerator BT_MESH_BLOB_CLI_STATE_SUSPENDED

Transfer is suspended.

enumerator BT_MESH_BLOB_CLI_STATE_XFER_PROGRESS_GET

Checking transfer progress.

Functions

int **bt_mesh_blob_cli_caps_get**(struct *bt_mesh_blob_cli* *cli, const struct *bt_mesh_blob_cli_inputs* *inputs)

Retrieve transfer capabilities for a list of Target nodes.

Queries the availability and capabilities of all Target nodes, producing a cumulative set of transfer capabilities for the Target nodes, and returning it through the *bt_mesh_blob_cli_cb::caps* callback.

Retrieving the capabilities may take several seconds, depending on the number of Target nodes and mesh network performance. The end of the procedure is indicated through the *bt_mesh_blob_cli_cb::caps* callback.

This procedure is not required, but strongly recommended as a preparation for a transfer to maximize performance and the chances of success.

Parameters

- **cli** – BLOB Transfer Client instance.
- **inputs** – Statically allocated BLOB Transfer Client transfer inputs.

Returns

0 on success, or (negative) error code otherwise.

int **bt_mesh_blob_cli_send**(struct *bt_mesh_blob_cli* *cli, const struct *bt_mesh_blob_cli_inputs* *inputs, const struct *bt_mesh_blob_xfer* *xfer, const struct *bt_mesh_blob_io* *io)

Perform a BLOB transfer.

Starts sending the transfer to the Target nodes. Only Target nodes with a `status` of *BT_MESH_BLOB_SUCCESS* will be considered.

The transfer will keep going either until all Target nodes have been dropped, or the full BLOB has been sent.

The BLOB transfer may take several minutes, depending on the number of Target nodes, size of the BLOB and mesh network performance. The end of the transfer is indicated through the *bt_mesh_blob_cli_cb::end* callback.

A Client only supports one transfer at the time.

Parameters

- **cli** – BLOB Transfer Client instance.
- **inputs** – Statically allocated BLOB Transfer Client transfer inputs.
- **xfer** – Statically allocated transfer parameters.
- **io** – BLOB stream to read the transfer from.

Returns
0 on success, or (negative) error code otherwise.

int **bt_mesh_blob_cli_suspend**(struct *bt_mesh_blob_cli* *cli)

Suspend the active transfer.

Parameters

- **cli** – BLOB Transfer Client instance.

Returns
0 on success, or (negative) error code otherwise.

int **bt_mesh_blob_cli_resume**(struct *bt_mesh_blob_cli* *cli)

Resume the suspended transfer.

Parameters

- **cli** – BLOB Transfer Client instance.

Returns
0 on success, or (negative) error code otherwise.

void **bt_mesh_blob_cli_cancel**(struct *bt_mesh_blob_cli* *cli)

Cancel an ongoing transfer.

Parameters

- **cli** – BLOB Transfer Client instance.

int **bt_mesh_blob_cli_xfer_progress_get**(struct *bt_mesh_blob_cli* *cli, const struct *bt_mesh_blob_cli_inputs* *inputs)

Get the progress of BLOB transfer.

This function can only be used if the BLOB Transfer Client is currently not performing a BLOB transfer. To get progress of the active BLOB transfer, use the *bt_mesh_blob_cli_xfer_progress_active_get* function.

Parameters

- **cli** – BLOB Transfer Client instance.
- **inputs** – Statically allocated BLOB Transfer Client transfer inputs.

Returns
0 on success, or (negative) error code otherwise.

uint8_t **bt_mesh_blob_cli_xfer_progress_active_get**(struct *bt_mesh_blob_cli* *cli)

Get the current progress of the active transfer in percent.

Parameters

- **cli** – BLOB Transfer Client instance.

Returns
The current transfer progress, or 0 if no transfer is active.

bool **bt_mesh_blob_cli_is_busy**(struct *bt_mesh_blob_cli* *cli)

Get the current state of the BLOB Transfer Client.

Parameters

- **cli** – BLOB Transfer Client instance.

Returns
true if the BLOB Transfer Client is currently participating in a transfer or retrieving the capabilities and false otherwise.

struct **bt_mesh_blob_target_pull**

#include <blob_cli.h> Target node's Pull mode (Pull BLOB Transfer Mode) context used while sending chunks to the Target node.

Public Members

int64_t **block_report_timestamp**

Timestamp when the Block Report Timeout Timer expires for this Target node.

uint8_t **missing**[*DIV_ROUND_UP*(*CONFIG_BT_MESH_BLOB_CHUNK_COUNT_MAX*, 8)]

Missing chunks reported by this Target node.

struct **bt_mesh_blob_target**

#include <blob_cli.h> BLOB Transfer Client Target node.

Public Members

sys_snode_t **n**

Linked list node.

uint16_t **addr**

Target node address.

struct *bt_mesh_blob_target_pull* ***pull**

Target node's Pull mode context.

Needs to be initialized when sending a BLOB in Pull mode.

uint8_t **status**

BLOB transfer status, see *bt_mesh_blob_status*.

struct **bt_mesh_blob_xfer_info**

#include <blob_cli.h> BLOB transfer information.

If `phase` is *BT_MESH_BLOB_XFER_PHASE_INACTIVE*, the fields below `phase` are not initialized. If `phase` is *BT_MESH_BLOB_XFER_PHASE_WAITING_FOR_START*, the fields below `id` are not initialized.

Public Members

enum *bt_mesh_blob_status* **status**

BLOB transfer status.

enum *bt_mesh_blob_xfer_mode* **mode**

BLOB transfer mode.

enum *bt_mesh_blob_xfer_phase* **phase**

BLOB transfer phase.

uint64_t **id**

BLOB ID.

uint32_t **size**

BLOB size in octets.

uint8_t **block_size_log**

Logarithmic representation of the block size.

uint16_t **mtu_size**

MTU size in octets.

const uint8_t ***missing_blocks**

Bit field indicating blocks that were not received.

struct **bt_mesh_blob_cli_inputs**

#include <*blob_cli.h*> BLOB Transfer Client transfer inputs.

Public Members

sys_slist_t **targets**

Linked list of Target nodes.

Each node should point to *bt_mesh_blob_target::n*.

uint16_t **app_idx**

AppKey index to send with.

uint16_t **group**

Group address destination for the BLOB transfer, or *BT_MESH_ADDR_UNASSIGNED* to send every message to each Target node individually.

uint8_t **ttl**

Time to live value of BLOB transfer messages.

uint16_t **timeout_base**

Additional response time for the Target nodes, in 10-second increments.

The extra time can be used to give the Target nodes more time to respond to messages from the Client. The actual timeout will be calculated according to the following formula:

```
*  timeout = 20 seconds + (10 seconds * timeout_base) + (100 ms * TTL)
*
```

If a Target node fails to respond to a message from the Client within the configured transfer timeout, the Target node is dropped.

struct **bt_mesh_blob_cli_caps**

#include <blob_cli.h> Transfer capabilities of a Target node.

Public Members

size_t **max_size**

Max BLOB size.

uint8_t **min_block_size_log**

Logarithmic representation of the minimum block size.

uint8_t **max_block_size_log**

Logarithmic representation of the maximum block size.

uint16_t **max_chunks**

Max number of chunks per block.

uint16_t **max_chunk_size**

Max chunk size.

uint16_t **mtu_size**

Max MTU size.

enum *bt_mesh_blob_xfer_mode* **modes**

Supported transfer modes.

struct **bt_mesh_blob_cli_cb**

#include <blob_cli.h> Event handler callbacks for the BLOB Transfer Client model.

All handlers are optional.

Public Members

void (***caps**)(struct *bt_mesh_blob_cli* *cli, const struct *bt_mesh_blob_cli_caps* *caps)

Capabilities retrieval completion callback.

Called when the capabilities retrieval procedure completes, indicating that a common set of acceptable transfer parameters have been established for the given list of Target nodes. All compatible Target nodes have status code *BT_MESH_BLOB_SUCCESS*.

Param cli
BLOB Transfer Client instance.

Param caps
Safe transfer capabilities if the transfer capabilities of at least one Target node has satisfied the Client, or NULL otherwise.

void (***lost_target**)(struct *bt_mesh_blob_cli* *cli, struct *bt_mesh_blob_target* *target, enum *bt_mesh_blob_status* reason)

Target node loss callback.

Called whenever a Target node has been lost due to some error in the transfer. Losing a Target node is not considered a fatal error for the Client until all Target nodes have been lost.

Param cli
BLOB Transfer Client instance.

Param target
Target node that was lost.

Param reason
Reason for the Target node loss.

void (*`suspended`)(struct *bt_mesh_blob_cli* *cli)

Transfer is suspended.

Called when the transfer is suspended due to response timeout from all Target nodes.

Param cli
BLOB Transfer Client instance.

void (*`end`)(struct *bt_mesh_blob_cli* *cli, const struct *bt_mesh_blob_xfer* *xfer, bool success)

Transfer end callback.

Called when the transfer ends.

Param cli
BLOB Transfer Client instance.

Param xfer
Completed transfer.

Param success
Status of the transfer. Is `true` if at least one Target node received the whole transfer.

void (*`xfer_progress`)(struct *bt_mesh_blob_cli* *cli, struct *bt_mesh_blob_target* *target, const struct *bt_mesh_blob_xfer_info* *info)

Transfer progress callback.

The content of `info` is invalidated upon exit from the callback. Therefore it needs to be copied if it is planned to be used later.

Param cli
BLOB Transfer Client instance.

Param target
Target node that responded to the request.

Param info
BLOB transfer information.

void (*`xfer_progress_complete`)(struct *bt_mesh_blob_cli* *cli)

End of Get Transfer Progress procedure.

Called when all Target nodes have responded or the procedure timed-out.

Param cli
BLOB Transfer Client instance.

struct `bt_mesh_blob_cli`

#include <*blob_cli.h*> BLOB Transfer Client model instance.

Public Members

const struct *bt_mesh_blob_cli_cb* *cb

Event handler callbacks.

The BLOB Transfer Client is instantiated on the sender node, and the BLOB Transfer Server is instantiated on the receiver nodes.

Concepts The BLOB transfer protocol introduces several new concepts to implement the BLOB transfer.

BLOBs BLOBs are binary objects up to 4 GB in size, that can contain any data the application would like to transfer through the mesh network. The BLOBs are continuous data objects, divided into blocks and chunks to make the transfers reliable and easy to process. No limitations are put on the contents or structure of the BLOB, and applications are free to define any encoding or compression they'd like on the data itself.

The BLOB transfer protocol does not provide any built-in integrity checks, encryption or authentication of the BLOB data. However, the underlying encryption of the Bluetooth Mesh protocol provides data integrity checks and protects the contents of the BLOB from third parties using network and application level encryption.

Blocks The binary objects are divided into blocks, typically from a few hundred to several thousand bytes in size. Each block is transmitted separately, and the BLOB Transfer Client ensures that all BLOB Transfer Servers have received the full block before moving on to the next. The block size is determined by the transfer's `block_size_log` parameter, and is the same for all blocks in the transfer except the last, which may be smaller. For a BLOB stored in flash memory, the block size is typically a multiple of the flash page size of the Target devices.

Chunks Each block is divided into chunks. A chunk is the smallest data unit in the BLOB transfer, and must fit inside a single Bluetooth Mesh access message excluding the opcode (379 bytes or less). The mechanism for transferring chunks depends on the transfer mode.

When operating in Push BLOB Transfer Mode, the chunks are sent as unacknowledged packets from the BLOB Transfer Client to all targeted BLOB Transfer Servers. Once all chunks in a block have been sent, the BLOB Transfer Client asks each BLOB Transfer Server if they're missing any chunks, and resends them. This is repeated until all BLOB Transfer Servers have received all chunks, or the BLOB Transfer Client gives up.

When operating in Pull BLOB Transfer Mode, the BLOB Transfer Server will request a small number of chunks from the BLOB Transfer Client at a time, and wait for the BLOB Transfer Client to send them before requesting more chunks. This repeats until all chunks have been transferred, or the BLOB Transfer Server gives up.

Read more about the transfer modes in *Transfer modes* section.

BLOB streams In the BLOB Transfer models' APIs, the BLOB data handling is separated from the high-level transfer handling. This split allows reuse of different BLOB storage and transfer strategies for different applications. While the high level transfer is controlled directly by the application, the BLOB data itself is accessed through a *BLOB stream*.

The BLOB stream is comparable to a standard library file stream. Through opening, closing, reading and writing, the BLOB Transfer model gets full access to the BLOB data, whether it's kept in flash, RAM, or on a peripheral. The BLOB stream is opened with an access mode (read or write) before it's used, and the BLOB Transfer models will move around inside the BLOB's data in blocks and chunks, using the BLOB stream as an interface.

Interaction Before the BLOB is read or written, the stream is opened by calling its *open* callback. When used with a BLOB Transfer Server, the BLOB stream is always opened in write mode, and when used with a BLOB Transfer Client, it's always opened in read mode.

For each block in the BLOB, the BLOB Transfer model starts by calling *block_start*. Then, depending on the access mode, the BLOB stream's *wr* or *rd* callback is called repeatedly to move data to or from the BLOB. When the model is done processing the block, it calls *block_end*. When the transfer is complete, the BLOB stream is closed by calling *close*.

Implementations The application may implement their own BLOB stream, or use the implementations provided by Zephyr:

BLOB Flash The BLOB Flash Readers and Writers implement BLOB reading to and writing from flash partitions defined in the *flash map*.

BLOB Flash Reader The BLOB Flash Reader interacts with the BLOB Transfer Client to read BLOB data directly from flash. It must be initialized by calling `bt_mesh_blob_flash_rd_init()` before being passed to the BLOB Transfer Client. Each BLOB Flash Reader only supports one transfer at the time.

BLOB Flash Writer The BLOB Flash Writer interacts with the BLOB Transfer Server to write BLOB data directly to flash. It must be initialized by calling `bt_mesh_blob_flash_rd_init()` before being passed to the BLOB Transfer Server. Each BLOB Flash Writer only supports one transfer at the time, and requires a block size that is a multiple of the flash page size. If a transfer is started with a block size lower than the flash page size, the transfer will be rejected.

The BLOB Flash Writer copies chunk data into a buffer to accommodate chunks that are unaligned with the flash write block size. The buffer data is padded with `0xff` if either the start or length of the chunk is unaligned.

API Reference

group `bt_mesh_blob_io_flash`

Functions

int **bt_mesh_blob_io_flash_init**(struct *bt_mesh_blob_io_flash* *flash, uint8_t area_id, off_t offset)

Initialize a flash stream.

Parameters

- `flash` – Flash stream.
- `area_id` – Flash partition identifier. See *flash_area_open*.
- `offset` – Offset into the flash area, in bytes.

Returns

0 on success or (negative) error code otherwise.

struct **bt_mesh_blob_io_flash**

#include <blob_io_flash.h> BLOB flash stream.

Public Members

uint8_t **area_id**

Flash area ID to write the BLOB to.

enum *bt_mesh_blob_io_mode* **mode**

Active stream mode.

off_t **offset**

Offset into the flash area to place the BLOB at (in bytes).

Transfer capabilities Each BLOB Transfer Server may have different transfer capabilities. The transfer capabilities of each device are controlled through the following configuration options:

- CONFIG_BT_MESH_BLOB_SIZE_MAX
- CONFIG_BT_MESH_BLOB_BLOCK_SIZE_MIN
- CONFIG_BT_MESH_BLOB_BLOCK_SIZE_MAX
- CONFIG_BT_MESH_BLOB_CHUNK_COUNT_MAX

The CONFIG_BT_MESH_BLOB_CHUNK_COUNT_MAX option is also used by the BLOB Transfer Client and affects memory consumption by the BLOB Transfer Client model structure.

To ensure that the transfer can be received by as many servers as possible, the BLOB Transfer Client can retrieve the capabilities of each BLOB Transfer Server before starting the transfer. The client will transfer the BLOB with the highest possible block and chunk size.

Transfer modes BLOBs can be transferred using two transfer modes, Push BLOB Transfer Mode and Pull BLOB Transfer Mode. In most cases, the transfer should be conducted in Push BLOB Transfer Mode.

In Push BLOB Transfer Mode, the send rate is controlled by the BLOB Transfer Client, which will push all the chunks of each block without any high level flow control. Push BLOB Transfer Mode supports any number of Target nodes, and should be the default transfer mode.

In Pull BLOB Transfer Mode, the BLOB Transfer Server will "pull" the chunks from the BLOB Transfer Client at its own rate. Pull BLOB Transfer Mode can be conducted with multiple Target nodes, and is intended for transferring BLOBs to Target nodes acting as *Low Power Node*. When operating in Pull BLOB Transfer Mode, the BLOB Transfer Server will request chunks from the BLOB Transfer Client in small batches, and wait for them all to arrive before requesting more chunks. This process is repeated until the BLOB Transfer Server has received all chunks in a block. Then, the BLOB Transfer Client starts the next block, and the BLOB Transfer Server requests all chunks of that block.

Transfer timeout The timeout of the BLOB transfer is based on a Timeout Base value. Both client and server use the same Timeout Base value, but they calculate timeout differently.

The BLOB Transfer Server uses the following formula to calculate the BLOB transfer timeout:

```
10 * (Timeout Base + 1) seconds
```

For the BLOB Transfer Client, the following formula is used:

```
(10000 * (Timeout Base + 2)) + (100 * TTL) milliseconds
```

where TTL is time to live value set in the transfer.

API reference This section contains types and defines common to the BLOB Transfer models.

group `bt_mesh_blob`

Defines

`CONFIG_BT_MESH_BLOB_CHUNK_COUNT_MAX`

Enums

enum `bt_mesh_blob_xfer_mode`

BLOB transfer mode.

Values:

enumerator `BT_MESH_BLOB_XFER_MODE_NONE`

No valid transfer mode.

enumerator `BT_MESH_BLOB_XFER_MODE_PUSH`

Push mode (Push BLOB Transfer Mode).

enumerator `BT_MESH_BLOB_XFER_MODE_PULL`

Pull mode (Pull BLOB Transfer Mode).

enumerator `BT_MESH_BLOB_XFER_MODE_ALL`

Both modes are valid.

enum `bt_mesh_blob_xfer_phase`

Transfer phase.

Values:

enumerator `BT_MESH_BLOB_XFER_PHASE_INACTIVE`

The BLOB Transfer Server is awaiting configuration.

enumerator `BT_MESH_BLOB_XFER_PHASE_WAITING_FOR_START`

The BLOB Transfer Server is ready to receive a BLOB transfer.

enumerator `BT_MESH_BLOB_XFER_PHASE_WAITING_FOR_BLOCK`

The BLOB Transfer Server is waiting for the next block of data.

enumerator `BT_MESH_BLOB_XFER_PHASE_WAITING_FOR_CHUNK`

The BLOB Transfer Server is waiting for the next chunk of data.

enumerator `BT_MESH_BLOB_XFER_PHASE_COMPLETE`

The BLOB was transferred successfully.

enumerator BT_MESH_BLOB_XFER_PHASE_SUSPENDED

The BLOB transfer is paused.

enum bt_mesh_blob_status

BLOB model status codes.

Values:

enumerator BT_MESH_BLOB_SUCCESS

The message was processed successfully.

enumerator BT_MESH_BLOB_ERR_INVALID_BLOCK_NUM

The Block Number field value is not within the range of blocks being transferred.

enumerator BT_MESH_BLOB_ERR_INVALID_BLOCK_SIZE

The block size is smaller than the size indicated by the Min Block Size Log state or is larger than the size indicated by the Max Block Size Log state.

enumerator BT_MESH_BLOB_ERR_INVALID_CHUNK_SIZE

The chunk size exceeds the size indicated by the Max Chunk Size state, or the number of chunks exceeds the number specified by the Max Total Chunks state.

enumerator BT_MESH_BLOB_ERR_WRONG_PHASE

The operation cannot be performed while the server is in the current phase.

enumerator BT_MESH_BLOB_ERR_INVALID_PARAM

A parameter value in the message cannot be accepted.

enumerator BT_MESH_BLOB_ERR_WRONG_BLOB_ID

The message contains a BLOB ID value that is not expected.

enumerator BT_MESH_BLOB_ERR_BLOB_TOO_LARGE

There is not enough space available in memory to receive the BLOB.

enumerator BT_MESH_BLOB_ERR_UNSUPPORTED_MODE

The transfer mode is not supported by the BLOB Transfer Server model.

enumerator BT_MESH_BLOB_ERR_INTERNAL

An internal error occurred on the node.

enumerator BT_MESH_BLOB_ERR_INFO_UNAVAILABLE

The requested information cannot be provided while the server is in the current phase.

enum bt_mesh_blob_io_mode

BLOB stream interaction mode.

Values:

enumerator BT_MESH_BLOB_READ

Read data from the stream.

enumerator **BT_MESH_BLOB_WRITE**
Write data to the stream.

struct **bt_mesh_blob_block**
#include <*blob.h*> BLOB transfer data block.

Public Members

size_t **size**
Block size in bytes.

off_t **offset**
Offset in bytes from the start of the BLOB.

uint16_t **number**
Block number.

uint16_t **chunk_count**
Number of chunks in block.

uint8_t **missing**[*DIV_ROUND_UP*(0, 8)]
Bitmap of missing chunks.

struct **bt_mesh_blob_chunk**
#include <*blob.h*> BLOB data chunk.

Public Members

off_t **offset**
Offset of the chunk data from the start of the block.

size_t **size**
Chunk data size.

uint8_t ***data**
Chunk data.

struct **bt_mesh_blob_xfer**
#include <*blob.h*> BLOB transfer.

Public Members

uint64_t **id**
BLOB ID.

size_t **size**

Total BLOB size in bytes.

enum *bt_mesh_blob_xfer_mode* **mode**

BLOB transfer mode.

uint16_t **chunk_size**

Base chunk size.

May be smaller for the last chunk.

struct **bt_mesh_blob_io**

#include <*blob.h*> BLOB stream.

Public Members

int (***open**)(const struct *bt_mesh_blob_io* *io, const struct *bt_mesh_blob_xfer* *xfer, enum *bt_mesh_blob_io_mode* mode)

Open callback.

Called when the reader is opened for reading.

Param io
BLOB stream.
Param xfer
BLOB transfer.
Param mode
Direction of the stream (read/write).
Return
0 on success, or (negative) error code otherwise.

void (***close**)(const struct *bt_mesh_blob_io* *io, const struct *bt_mesh_blob_xfer* *xfer)

Close callback.

Called when the reader is closed.

Param io
BLOB stream.
Param xfer
BLOB transfer.

int (***block_start**)(const struct *bt_mesh_blob_io* *io, const struct *bt_mesh_blob_xfer* *xfer, const struct *bt_mesh_blob_block* *block)

Block start callback.

Called when a new block is opened for sending. Each block is only sent once, and are always sent in increasing order. The data chunks inside a single block may be requested out of order and multiple times.

Param io
BLOB stream.
Param xfer
BLOB transfer.
Param block
Block that was started.

void (*block_end)(const struct *bt_mesh_blob_io* *io, const struct *bt_mesh_blob_xfer* *xfer, const struct *bt_mesh_blob_block* *block)

Block end callback.

Called when the current block has been transmitted in full. No data from this block will be requested again, and the application data associated with this block may be discarded.

Param io
BLOB stream.

Param xfer
BLOB transfer.

Param block
Block that finished sending.

int (*wr)(const struct *bt_mesh_blob_io* *io, const struct *bt_mesh_blob_xfer* *xfer, const struct *bt_mesh_blob_block* *block, const struct *bt_mesh_blob_chunk* *chunk)

Chunk data write callback.

Used by the BLOB Transfer Server on incoming data.

Each block is divided into chunks of data. This callback is called when a new chunk of data is received. Chunks may be received in any order within their block.

If the callback returns successfully, this chunk will be marked as received, and will not be received again unless the block is restarted due to a transfer suspension. If the callback returns a non-zero value, the chunk remains unreceived, and the BLOB Transfer Client will attempt to resend it later.

Note that the Client will only perform a limited number of attempts at delivering a chunk before dropping a Target node from the transfer. The number of retries performed by the Client is implementation specific.

Param io
BLOB stream.

Param xfer
BLOB transfer.

Param block
Block the chunk is part of.

Param chunk
Received chunk.

Return
0 on success, or (negative) error code otherwise.

int (*rd)(const struct *bt_mesh_blob_io* *io, const struct *bt_mesh_blob_xfer* *xfer, const struct *bt_mesh_blob_block* *block, const struct *bt_mesh_blob_chunk* *chunk)

Chunk data read callback.

Used by the BLOB Transfer Client to fetch outgoing data.

The Client calls the chunk data request callback to populate a chunk message going out to the Target nodes. The data request callback may be called out of order and multiple times for each offset, and cannot be used as an indication of progress.

Returning a non-zero status code on the chunk data request callback results in termination of the transfer.

Param io
BLOB stream.

Param xfer
BLOB transfer.

Param block
Block the chunk is part of.

Param chunk
Chunk to get the data of. The buffer pointer to by the `data` member should be filled by the callback.
Return
0 on success, or (negative) error code otherwise.

Device Firmware Update (DFU) Bluetooth Mesh supports the distribution of firmware images across a mesh network. The Bluetooth mesh DFU subsystem implements the Bluetooth Mesh Device Firmware Update Model specification version 1.0.

Bluetooth Mesh DFU implements a distribution mechanism for firmware images, and does not put any restrictions on the size, format or usage of the images. The primary design goal of the subsystem is to provide the qualifiable parts of the Bluetooth Mesh DFU specification, and leave the usage, firmware validation and deployment to the application.

The DFU specification is implemented in the Zephyr Bluetooth Mesh DFU subsystem as three separate models:

Firmware Update Server The Firmware Update Server model implements the Target node functionality of the *Device Firmware Update (DFU)* subsystem. It extends the *BLOB Transfer Server*, which it uses to receive the firmware image binary from the Distributor node.

Together with the extended BLOB Transfer Server model, the Firmware Update Server model implements all the required functionality for receiving firmware updates over the mesh network, but does not provide any functionality for storing, applying or verifying the images.

Firmware images The Firmware Update Server holds a list of all the updatable firmware images on the device. The full list shall be passed to the server through the `_imgs` parameter in *`BT_MESH_DFU_SRV_INIT`*, and must be populated before the Bluetooth Mesh subsystem is started. Each firmware image in the image list must be independently updatable, and should have its own firmware ID.

For instance, a device with an upgradable bootloader, an application and a peripheral chip with firmware update capabilities could have three entries in the firmware image list, each with their own separate firmware ID.

Receiving transfers The Firmware Update Server model uses a BLOB Transfer Server model on the same element to transfer the binary image. The interaction between the Firmware Update Server, BLOB Transfer Server and application is described below:

Transfer check The transfer check is an optional pre-transfer check the application can perform on incoming firmware image metadata. The Firmware Update Server performs the transfer check by calling the *`check`* callback.

The result of the transfer check is a pass/fail status return and the expected *`bt_mesh_dfu_effect`*. The DFU effect return parameter will be communicated back to the Distributor, and should indicate what effect the firmware update will have on the mesh state of the device.

Composition Data and Models Metadata If the transfer will cause the device to change its Composition Data or become unprovisioned, this should be communicated through the effect parameter of the metadata check.

When the transfer will cause the Composition Data to change, and the *Remote Provisioning Server* is supported, the Composition Data of the new firmware image will be represented by Composition Data Pages 128, 129, and 130. The Models Metadata of the new firmware image will be

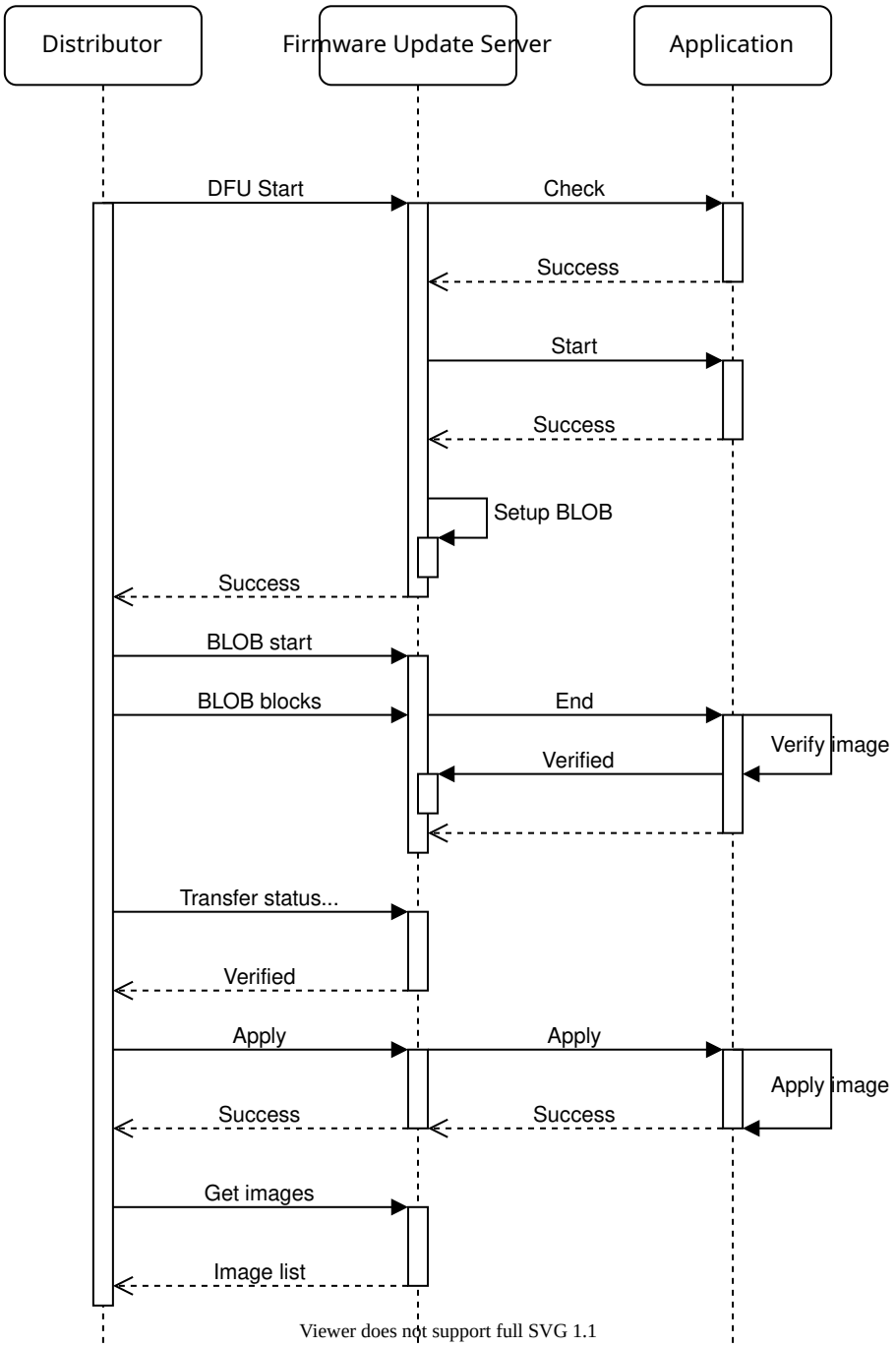


Fig. 9: Bluetooth Mesh Firmware Update Server transfer

represented by Models Metadata Page 128. Composition Data Pages 0, 1 and 2, and Models Metadata Page 0, will represent the Composition Data and the Models Metadata of the old firmware image until the device is reprovisioned with Node Provisioning Protocol Interface (NPPI) procedures using the *Remote Provisioning Client*.

The application must call functions `bt_mesh_comp_change_prepare()` and `bt_mesh_models_metadata_change_prepare()` to store the existing Composition Data and Models Metadata pages before booting into the firmware with the updated Composition Data and Models Metadata. The old Composition Data will then be loaded into Composition Data Pages 0, 1 and 2, while the Composition Data in the new firmware will be loaded into Composition Data Pages 128, 129 and 130. The Models Metadata for the old image will be loaded into Models Metadata Page 0, and the Models Metadata for the new image will be loaded into Models Metadata Page 128.

Limitation:

- It is not possible to change the Composition Data of the device and keep the device provisioned and working with the old firmware after the new firmware image is applied.

Start The Start procedure prepares the application for the incoming transfer. It'll contain information about which image is being updated, as well as the update metadata.

The Firmware Update Server `start` callback must return a pointer to the BLOB Writer the BLOB Transfer Server will send the BLOB to.

BLOB transfer After the setup stage, the Firmware Update Server prepares the BLOB Transfer Server for the incoming transfer. The entire firmware image is transferred to the BLOB Transfer Server, which passes the image to its assigned BLOB Writer.

At the end of the BLOB transfer, the Firmware Update Server calls its `end` callback.

Image verification After the BLOB transfer has finished, the application should verify the image in any way it can to ensure that it is ready for being applied. Once the image has been verified, the application calls `bt_mesh_dfu_srv_verified()`.

If the image can't be verified, the application calls `bt_mesh_dfu_srv_rejected()`.

Applying the image Finally, if the image was verified, the Distributor may instruct the Firmware Update Server to apply the transfer. This is communicated to the application through the `apply` callback. The application should swap the image and start running with the new firmware. The firmware image table should be updated to reflect the new firmware ID of the updated image.

When the transfer applies to the mesh application itself, the device might have to reboot as part of the swap. This restart can be performed from inside the apply callback, or done asynchronously. After booting up with the new firmware, the firmware image table should be updated before the Bluetooth Mesh subsystem is started.

The Distributor will read out the firmware image table to confirm that the transfer was successfully applied. If the metadata check indicated that the device would become unprovisioned, the Target node is not required to respond to this check.

API reference

group `bt_mesh_dfu_srv`

API for the Bluetooth Mesh Firmware Update Server model.

Defines

BT_MESH_DFU_SRV_INIT(_handlers, _imgs, _img_count)

Initialization parameters for *Firmware Update Server model*.

Parameters

- **_handlers** – DFU handler function structure.
- **_imgs** – List of *bt_mesh_dfu_img* managed by this Server.
- **_img_count** – Number of DFU images managed by this Server.

BT_MESH_MODEL_DFU_SRV(_srv)

Firmware Update Server model entry.

Parameters

- **_srv** – Pointer to a *Firmware Update Server model* instance.

Functions

void **bt_mesh_dfu_srv_verified**(struct *bt_mesh_dfu_srv* *srv)

Accept the received DFU transfer.

Should be called at the end of a successful DFU transfer.

If the DFU transfer completes successfully, the application should verify the image validity (including any image authentication or integrity checks), and call this function if the image is ready to be applied.

Parameters

- **srv** – Firmware Update Server instance.

void **bt_mesh_dfu_srv_rejected**(struct *bt_mesh_dfu_srv* *srv)

Reject the received DFU transfer.

Should be called at the end of a successful DFU transfer.

If the DFU transfer completes successfully, the application should verify the image validity (including any image authentication or integrity checks), and call this function if one of the checks fail.

Parameters

- **srv** – Firmware Update Server instance.

void **bt_mesh_dfu_srv_cancel**(struct *bt_mesh_dfu_srv* *srv)

Cancel the ongoing DFU transfer.

Parameters

- **srv** – Firmware Update Server instance.

void **bt_mesh_dfu_srv_applied**(struct *bt_mesh_dfu_srv* *srv)

Confirm that the received DFU transfer was applied.

Should be called as a result of the *bt_mesh_dfu_srv_cb::apply* callback.

Parameters

- **srv** – Firmware Update Server instance.

bool **bt_mesh_dfu_srv_is_busy**(const struct *bt_mesh_dfu_srv* *srv)

Check if the Firmware Update Server is busy processing a transfer.

Parameters

- **srv** – Firmware Update Server instance.

Returns

true if a DFU procedure is in progress, false otherwise.

uint8_t **bt_mesh_dfu_srv_progress**(const struct *bt_mesh_dfu_srv* *srv)

Get the progress of the current DFU procedure, in percent.

Parameters

- **srv** – Firmware Update Server instance.

Returns

The current transfer progress in percent.

struct **bt_mesh_dfu_srv_cb**

#include <dfu_srv.h> Firmware Update Server event callbacks.

Public Members

int (***check**)(struct *bt_mesh_dfu_srv* *srv, const struct *bt_mesh_dfu_img* *img, struct *net_buf_simple* *metadata, enum *bt_mesh_dfu_effect* *effect)

Transfer check callback.

The transfer check can be used to validate the incoming transfer before it starts. The contents of the metadata is implementation specific, and should contain all the information the application needs to determine whether this image should be accepted, and what the effect of the transfer would be.

If applying the image will have an effect on the provisioning state of the mesh stack, this can be communicated through the `effect` return parameter.

The metadata check can be performed both as part of starting a new transfer and as a separate procedure.

This handler is optional.

Param srv
Firmware Update Server instance.

Param img
DFU image the metadata check is performed on.

Param metadata
Image metadata.

Param effect
Return parameter for the image effect on the provisioning state of the mesh stack.

Return
0 on success, or (negative) error code otherwise.

int (***start**)(struct *bt_mesh_dfu_srv* *srv, const struct *bt_mesh_dfu_img* *img, struct *net_buf_simple* *metadata, const struct *bt_mesh_blob_io* **io)

Transfer start callback.

Called when the Firmware Update Server is ready to start a new DFU transfer. The application must provide an initialized BLOB stream to be used during the DFU transfer.

The following error codes are treated specially, and should be used to communicate these issues:

- `-ENOMEM`: The device cannot fit this image.
- `-EBUSY`: The application is temporarily unable to accept the transfer.
- `-EALREADY`: The device has already received and verified this image, and there's no need to transfer it again. The Firmware Update model will skip the transfer phase, and mark the image as verified.

This handler is mandatory.

Param srv
Firmware Update Server instance.

Param img
DFU image being updated.

Param metadata
Image metadata.

Param io
BLOB stream return parameter. Must be set to a valid BLOB stream by the callback.

Return
0 on success, or (negative) error code otherwise. Return codes `-ENOMEM`, `-EBUSY` `-EALREADY` will be passed to the updater, other error codes are reported as internal errors.

void (*end)(struct *bt_mesh_dfu_srv* *srv, const struct *bt_mesh_dfu_img* *img, bool success)

Transfer end callback.

This handler is optional.

If the transfer is successful, the application should verify the firmware image, and call either *bt_mesh_dfu_srv_verified* or *bt_mesh_dfu_srv_rejected* depending on the outcome.

If the transfer fails, the Firmware Update Server will be available for new transfers immediately after this function returns.

Param srv
Firmware Update Server instance.

Param img
DFU image that failed the update.

Param success
Whether the DFU transfer was successful.

int (*recover)(struct *bt_mesh_dfu_srv* *srv, const struct *bt_mesh_dfu_img* *img, const struct *bt_mesh_blob_io* **io)

Transfer recovery callback.

If the device reboots in the middle of a transfer, the Firmware Update Server calls this function when the Bluetooth Mesh subsystem is started.

This callback is optional, but transfers will not be recovered after a reboot without it.

Param srv
Firmware Update Server instance.

Param img
DFU image being updated.

Param io
BLOB stream return parameter. Must be set to a valid BLOB stream by the callback.

Return
0 on success, or (negative) error code to abandon the transfer.

int (*apply)(struct *bt_mesh_dfu_srv* *srv, const struct *bt_mesh_dfu_img* *img)

Transfer apply callback.

Called after a transfer has been validated, and the updater sends an apply message to the Target nodes.

This handler is optional.

Param srv
Firmware Update Server instance.

Param img
DFU image that should be applied.

Return
0 on success, or (negative) error code otherwise.

struct bt_mesh_dfu_srv

#include <dfu_srv.h> Firmware Update Server instance.

Should be initialized with *BT_MESH_DFU_SRV_INIT*.

Public Members

struct *bt_mesh_blob_srv* blob

Underlying BLOB Transfer Server.

const struct *bt_mesh_dfu_srv_cb* *cb

Callback structure.

const struct *bt_mesh_dfu_img* *imgs

List of updatable images.

size_t img_count

Number of updatable images.

Firmware Update Client The Firmware Update Client is responsible for distributing firmware updates through the mesh network. The Firmware Update Client uses the *BLOB Transfer Client* as a transport for its transfers.

API reference

group bt_mesh_dfu_cli

API for the Bluetooth Mesh Firmware Update Client model.

Defines

BT_MESH_DFU_CLI_INIT(_handlers)

Initialization parameters for the *Firmware Uppdate Client model*.

See also

bt_mesh_dfu_cli_cb.

Parameters

- **_handlers** – Handler callback structure.

BT_MESH_MODEL_DFU_CLI(_cli)

Firmware Update Client model Composition Data entry.

Parameters

- **_cli** – Pointer to a *Firmware Uppdate Client model* instance.

Typedefs

typedef enum *bt_mesh_dfu_iter* (***bt_mesh_dfu_img_cb_t**)(struct *bt_mesh_dfu_cli* *cli, struct *bt_mesh_msg_ctx* *ctx, uint8_t idx, uint8_t total, const struct *bt_mesh_dfu_img* *img, void *cb_data)

DFU image callback.

The image callback is called for every DFU image on the Target node when calling *bt_mesh_dfu_cli_imgs_get*.

Param cli
Firmware Update Client model instance.

Param ctx
Message context of the received message.

Param idx
Image index.

Param total
Total number of images on the Target node.

Param img
Image information for the given image index.

Param cb_data
Callback data.

Retval BT_MESH_DFU_ITER_STOP
Stop iterating through the image list and return from *bt_mesh_dfu_cli_imgs_get*.

Retval BT_MESH_DFU_ITER_CONTINUE
Continue iterating through the image list if any images remain.

Functions

int **bt_mesh_dfu_cli_send**(struct *bt_mesh_dfu_cli* *cli, const struct *bt_mesh_blob_cli_inputs* *inputs, const struct *bt_mesh_blob_io* *io, const struct *bt_mesh_dfu_cli_xfer* *xfer)

Start distributing a DFU.

Starts distribution of the firmware in the given slot to the list of DFU Target nodes in `ctx`. The transfer runs in the background, and its end is signalled through the *bt_mesh_dfu_cli_cb::ended* callback.

> **Note**
>
> The BLOB Transfer Client transfer inputs `targets` list must point to a list of *bt_mesh_dfu_target* nodes.

Parameters

- **cli** – Firmware Update Client model instance.
- **inputs** – BLOB Transfer Client transfer inputs.
- **io** – BLOB stream to read BLOB from.
- **xfer** – Firmware Update Client transfer parameters.

Returns
0 on success, or (negative) error code otherwise.

int **bt_mesh_dfu_cli_suspend**(struct *bt_mesh_dfu_cli* *cli)

Suspend a DFU transfer.

Parameters

- **cli** – Firmware Update Client instance.

Returns
0 on success, or (negative) error code otherwise.

int **bt_mesh_dfu_cli_resume**(struct *bt_mesh_dfu_cli* *cli)

Resume the suspended transfer.

Parameters

- **cli** – Firmware Update Client instance.

Returns
0 on success, or (negative) error code otherwise.

int **bt_mesh_dfu_cli_cancel**(struct *bt_mesh_dfu_cli* *cli, struct *bt_mesh_msg_ctx* *ctx)

Cancel a DFU transfer.

Will cancel the ongoing DFU transfer, or the transfer on a specific Target node if `ctx` is valid.

Parameters

- **cli** – Firmware Update Client model instance.
- **ctx** – Message context, or NULL to cancel the ongoing DFU transfer.

Returns
0 on success, or (negative) error code otherwise.

int **bt_mesh_dfu_cli_apply**(struct *bt_mesh_dfu_cli* *cli)

Apply the completed DFU transfer.

A transfer can only be applied after it has ended successfully. The Firmware Update Client's `applied` callback is called at the end of the apply procedure.

Parameters

- **cli** – Firmware Update Client model instance.

Returns
0 on success, or (negative) error code otherwise.

int **bt_mesh_dfu_cli_confirm**(struct *bt_mesh_dfu_cli* *cli)

Confirm that the active transfer has been applied on the Target nodes.

A transfer can only be confirmed after it has been applied. The Firmware Update Client's `confirmed` callback is called at the end of the confirm procedure.

Target nodes that have reported the effect as *BT_MESH_DFU_EFFECT_UNPROV* are expected to not respond to the query, and will fail if they do.

Parameters

- **cli** – Firmware Update Client model instance.

Returns
0 on success, or (negative) error code otherwise.

uint8_t **bt_mesh_dfu_cli_progress**(struct *bt_mesh_dfu_cli* *cli)

Get progress as a percentage of completion.

Parameters

- **cli** – Firmware Update Client model instance.

Returns
The progress of the current transfer in percent, or 0 if no transfer is active.

bool **bt_mesh_dfu_cli_is_busy**(struct *bt_mesh_dfu_cli* *cli)

Check whether a DFU transfer is in progress.

Parameters

- **cli** – Firmware Update Client model instance.

Returns
true if the BLOB Transfer Client is currently participating in a transfer, false otherwise.

int **bt_mesh_dfu_cli_imgs_get**(struct *bt_mesh_dfu_cli* *cli, struct *bt_mesh_msg_ctx* *ctx, *bt_mesh_dfu_img_cb_t* cb, void *cb_data, uint8_t max_count)

Perform a DFU image list request.

Requests the full list of DFU images on a Target node, and iterates through them, calling the cb for every image.

The DFU image list request can be used to determine which image index the Target node holds its different firmwares in.

Waits for a response until the procedure timeout expires.

Parameters

- **cli** – Firmware Update Client model instance.
- **ctx** – Message context.
- **cb** – Callback to call for each image index.
- **cb_data** – Callback data to pass to cb.
- **max_count** – Max number of images to return.

Returns
0 on success, or (negative) error code otherwise.

int **bt_mesh_dfu_cli_metadata_check**(struct *bt_mesh_dfu_cli* *cli, struct *bt_mesh_msg_ctx* *ctx, uint8_t img_idx, const struct *bt_mesh_dfu_slot* *slot, struct *bt_mesh_dfu_metadata_status* *rsp)

Perform a metadata check for the given DFU image slot.

The metadata check procedure allows the Firmware Update Client to check if a Target node will accept a transfer of this DFU image slot, and what the effect would be.

Waits for a response until the procedure timeout expires.

Parameters

- **cli** – Firmware Update Client model instance.
- **ctx** – Message context.
- **img_idx** – Target node's image index to check.
- **slot** – DFU image slot to check for.
- **rsp** – Metadata status response buffer.

Returns

0 on success, or (negative) error code otherwise.

int **bt_mesh_dfu_cli_status_get**(struct *bt_mesh_dfu_cli* *cli, struct *bt_mesh_msg_ctx* *ctx, struct *bt_mesh_dfu_target_status* *rsp)

Get the status of a Target node.

Parameters

- **cli** – Firmware Update Client model instance.
- **ctx** – Message context.
- **rsp** – Response data buffer.

Returns

0 on success, or (negative) error code otherwise.

int32_t **bt_mesh_dfu_cli_timeout_get**(void)

Get the current procedure timeout value.

Returns

The configured procedure timeout.

void **bt_mesh_dfu_cli_timeout_set**(int32_t timeout)

Set the procedure timeout value.

Parameters

- **timeout** – The new procedure timeout.

struct **bt_mesh_dfu_target**

#include <dfu_cli.h> DFU Target node.

Public Members

struct *bt_mesh_blob_target* **blob**

BLOB Target node.

uint8_t **img_idx**

Image index on the Target node.

uint8_t **effect**
Expected DFU effect, see *bt_mesh_dfu_effect*.

uint8_t **status**
Current DFU status, see *bt_mesh_dfu_status*.

uint8_t **phase**
Current DFU phase, see *bt_mesh_dfu_phase*.

struct **bt_mesh_dfu_metadata_status**
#include <dfu_cli.h> Metadata status response.

Public Members

uint8_t **idx**
Image index.

enum *bt_mesh_dfu_status* **status**
Status code.

enum *bt_mesh_dfu_effect* **effect**
Effect of transfer.

struct **bt_mesh_dfu_target_status**
#include <dfu_cli.h> DFU Target node status parameters.

Public Members

enum *bt_mesh_dfu_status* **status**
Status of the previous operation.

enum *bt_mesh_dfu_phase* **phase**
Phase of the current DFU transfer.

enum *bt_mesh_dfu_effect* **effect**
The effect the update will have on the Target device's state.

uint64_t **blob_id**
BLOB ID used in the transfer.

uint8_t **img_idx**
Image index to transfer.

uint8_t **ttl**
TTL used in the transfer.

uint16_t **timeout_base**

Additional response time for the Target nodes, in 10-second increments.

The extra time can be used to give the Target nodes more time to respond to messages from the Client. The actual timeout will be calculated according to the following formula:

```
*  timeout = 20 seconds + (10 seconds * timeout_base) + (100 ms * TTL)
*
```

If a Target node fails to respond to a message from the Client within the configured transfer timeout, the Target node is dropped.

struct **bt_mesh_dfu_cli_cb**

#include <dfu_cli.h> Firmware Update Client event callbacks.

Public Members

void (***suspended**)(struct *bt_mesh_dfu_cli* *cli)

BLOB transfer is suspended.

Called when the BLOB transfer is suspended due to response timeout from all Target nodes.

Param cli
Firmware Update Client model instance.

void (***ended**)(struct *bt_mesh_dfu_cli* *cli, enum *bt_mesh_dfu_status* reason)

DFU ended.

Called when the DFU transfer ends, either because all Target nodes were lost or because the transfer was completed successfully.

Param cli
Firmware Update Client model instance.

Param reason
Reason for ending.

void (***applied**)(struct *bt_mesh_dfu_cli* *cli)

DFU transfer applied on all active Target nodes.

Called at the end of the apply procedure started by *bt_mesh_dfu_cli_apply*.

Param cli
Firmware Update Client model instance.

void (***confirmed**)(struct *bt_mesh_dfu_cli* *cli)

DFU transfer confirmed on all active Target nodes.

Called at the end of the apply procedure started by *bt_mesh_dfu_cli_confirm*.

Param cli
Firmware Update Client model instance.

void (***lost_target**)(struct *bt_mesh_dfu_cli* *cli, struct *bt_mesh_dfu_target* *target)

DFU Target node was lost.

A DFU Target node was dropped from the receivers list. The Target node's `status` is set to reflect the reason for the failure.

Param cli
Firmware Update Client model instance.
Param target
DFU Target node that was lost.

struct **bt_mesh_dfu_cli**

#include <*dfu_cli.h*> Firmware Update Client model instance.

Should be initialized with *BT_MESH_DFU_CLI_INIT*.

Public Members

const struct *bt_mesh_dfu_cli_cb* ***cb**

Callback structure.

struct *bt_mesh_blob_cli* **blob**

Underlying BLOB Transfer Client.

struct **bt_mesh_dfu_cli_xfer_blob_params**

#include <*dfu_cli.h*> BLOB parameters for Firmware Update Client transfer:

Public Members

uint16_t **chunk_size**

Base chunk size.

May be smaller for the last chunk.

struct **bt_mesh_dfu_cli_xfer**

#include <*dfu_cli.h*> Firmware Update Client transfer parameters:

Public Members

uint64_t **blob_id**

BLOB ID to use for this transfer, or 0 to set it randomly.

const struct *bt_mesh_dfu_slot* ***slot**

DFU image slot to transfer.

enum *bt_mesh_blob_xfer_mode* **mode**

Transfer mode (Push (Push BLOB Transfer Mode) or Pull (Pull BLOB Transfer Mode))

const struct *bt_mesh_dfu_cli_xfer_blob_params* ***blob_params**

BLOB parameters to be used for the transfer, or NULL to retrieve Target nodes' capabilities before sending a firmware.

Firmware Distribution Server The Firmware Distribution Server model implements the Distributor role for the *Device Firmware Update (DFU)* subsystem. It extends the *BLOB Transfer Server*, which it uses to receive the firmware image binary from the Initiator node. It also instantiates a *Firmware Update Client*, which it uses to distribute firmware updates throughout the mesh network.

> **Note**
>
> Currently, the Firmware Distribution Server supports out-of-band (OOB) retrieval of firmware images over SMP service only.

The Firmware Distribution Server does not have an API of its own, but relies on a Firmware Distribution Client model on a different device to give it information and trigger image distribution and upload.

Firmware slots The Firmware Distribution Server is capable of storing multiple firmware images for distribution. Each slot contains a separate firmware image with metadata, and can be distributed to other mesh nodes in the network in any order. The contents, format and size of the firmware images are vendor specific, and may contain data from other vendors. The application should never attempt to execute or modify them.

The slots are managed remotely by a Firmware Distribution Client, which can both upload new slots and delete old ones. The application is notified of changes to the slots through the Firmware Distribution Server's callbacks (`bt_mesh_fd_srv_cb`). While the metadata for each firmware slot is stored internally, the application must provide a *BLOB streams* for reading and writing the firmware image.

API reference

group `bt_mesh_dfd_srv`

API for the Firmware Distribution Server model.

Defines

`CONFIG_BT_MESH_DFD_SRV_TARGETS_MAX`

`CONFIG_BT_MESH_DFD_SRV_SLOT_MAX_SIZE`

`CONFIG_BT_MESH_DFD_SRV_SLOT_SPACE`

`BT_MESH_DFD_SRV_INIT(_cb)`

Initialization parameters for the *Firmware Distribution Server model*.

Parameters

- `_cb` – **[in]** Pointer to a *bt_mesh_dfd_srv_cb* instance.

`BT_MESH_MODEL_DFD_SRV(_srv)`

Firmware Distribution Server model Composition Data entry.

Parameters

- `_srv` – Pointer to a *Firmware Distribution Server model* instance.

struct **bt_mesh_dfd_srv_cb**

#include <dfd_srv.h> Firmware Distribution Server callbacks:

Public Members

int (***recv**)(struct *bt_mesh_dfd_srv* *srv, const struct *bt_mesh_dfu_slot* *slot, const struct *bt_mesh_blob_io* **io)

Slot receive callback.

Called at the start of an upload procedure. The callback must fill `io` with a pointer to a writable BLOB stream for the Firmware Distribution Server to write the firmware image to.

Param srv
Firmware Distribution Server model instance.

Param slot
DFU image slot being received.

Param io
BLOB stream response pointer.

Return
0 on success, or (negative) error code otherwise.

void (***del**)(struct *bt_mesh_dfd_srv* *srv, const struct *bt_mesh_dfu_slot* *slot)

Slot delete callback.

Called when the Firmware Distribution Server is about to delete a DFU image slot. All allocated data associated with the firmware slot should be deleted.

Param srv
Firmware Update Server instance.

Param slot
DFU image slot being deleted.

int (***send**)(struct *bt_mesh_dfd_srv* *srv, const struct *bt_mesh_dfu_slot* *slot, const struct *bt_mesh_blob_io* **io)

Slot send callback.

Called at the start of a distribution procedure. The callback must fill `io` with a pointer to a readable BLOB stream for the Firmware Distribution Server to read the firmware image from.

Param srv
Firmware Distribution Server model instance.

Param slot
DFU image slot being sent.

Param io
BLOB stream response pointer.

Return
0 on success, or (negative) error code otherwise.

void (***phase**)(struct *bt_mesh_dfd_srv* *srv, enum *bt_mesh_dfd_phase* phase)

Phase change callback (Optional).

Called whenever the phase of the Firmware Distribution Server changes.

Param srv
Firmware Distribution Server model instance.

Param phase
New Firmware Distribution phase.

struct **bt_mesh_dfd_srv**

#include <dfd_srv.h> Firmware Distribution Server instance.

Overview

DFU roles The Bluetooth Mesh DFU subsystem defines three different roles the mesh nodes have to assume in the distribution of firmware images:

Target node
Target node is the receiver and user of the transferred firmware images. All its functionality is implemented by the *Firmware Update Server* model. A transfer may be targeting any number of Target nodes, and they will all be updated concurrently.

Distributor
The Distributor role serves two purposes in the DFU process. First, it's acting as the Target node in the Upload Firmware procedure, then it distributes the uploaded image to other Target nodes as the Distributor. The Distributor does not select the parameters of the transfer, but relies on an Initiator to give it a list of Target nodes and transfer parameters. The Distributor functionality is implemented in two models, *Firmware Distribution Server* and *Firmware Update Client*. The *Firmware Distribution Server* is responsible for communicating with the Initiator, and the *Firmware Update Client* is responsible for distributing the image to the Target nodes.

Initiator
The Initiator role is typically implemented by the same device that implements the Bluetooth Mesh *Provisioner* and *Configurator* roles. The Initiator needs a full overview of the potential Target nodes and their firmware, and will control (and initiate) all firmware updates. The Initiator role is not implemented in the Zephyr Bluetooth Mesh DFU subsystem.

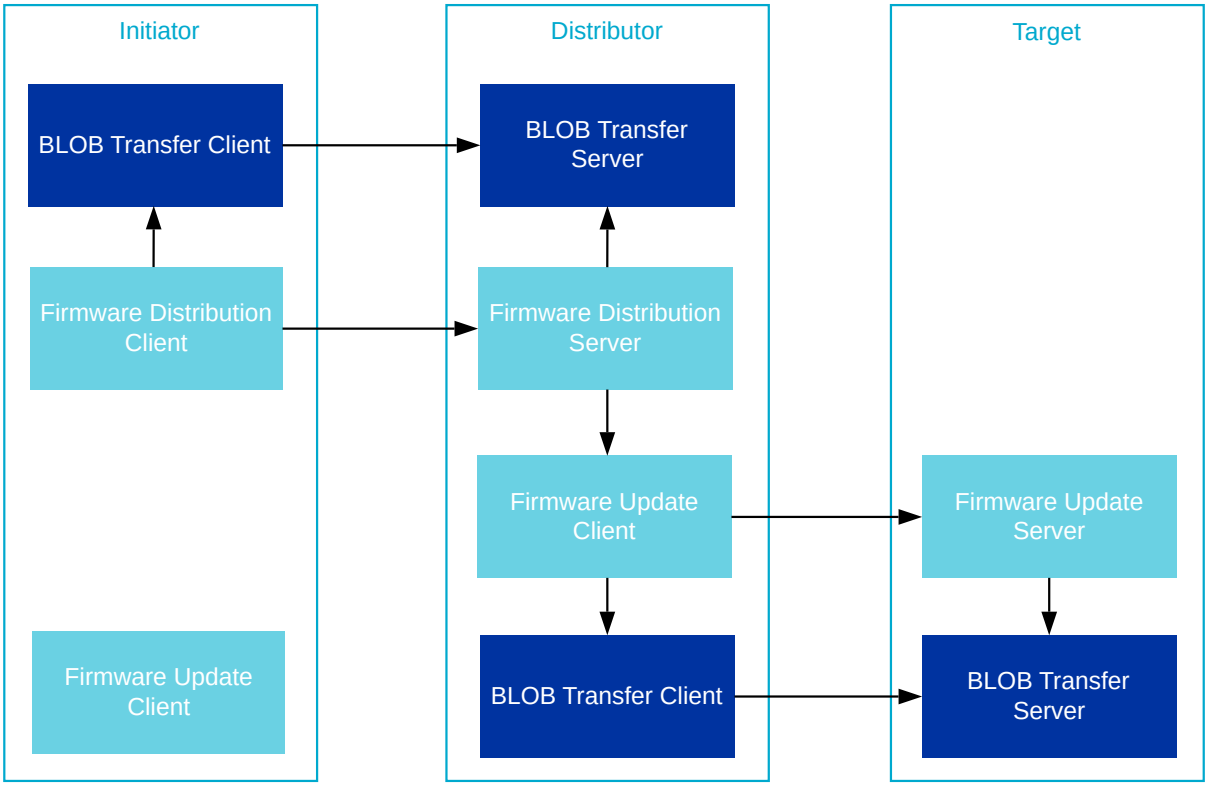


Fig. 10: DFU roles and the associated Bluetooth Mesh models

Bluetooth Mesh applications may combine the DFU roles in any way they'd like, and even take on multiple instances of the same role by instantiating the models on separate elements. For instance, the Distributor and Initiator role can be combined by instantiating the *Firmware Update Client* on the Initiator node and calling its API directly.

It's also possible to combine the Initiator and Distributor devices into a single device, and replace the Firmware Distribution Server model with a proprietary mechanism that will access the Firmware Update Client model directly, e.g. over a serial protocol.

Note

All DFU models instantiate one or more *BLOB Transfer models*, and may need to be spread over multiple elements for certain role combinations.

Stages The Bluetooth Mesh DFU process is designed to act in three stages:

Upload stage
: First, the image is uploaded to a Distributor in a mesh network by an external entity, such as a phone or gateway (the Initiator). During the Upload stage, the Initiator transfers the firmware image and all its metadata to the Distributor node inside the mesh network. The Distributor stores the firmware image and its metadata persistently, and awaits further instructions from the Initiator. The time required to complete the upload process depends on the size of the image. After the upload completes, the Initiator can disconnect from the network during the much more time-consuming Distribution stage. Once the firmware has been uploaded to the Distributor, the Initiator may trigger the Distribution stage at any time.

Firmware Capability Check stage (optional)
: Before starting the Distribution stage, the Initiator may optionally check if Target nodes can accept the new firmware. Nodes that do not respond, or respond that they can't receive the new firmware, are excluded from the firmware distribution process.

Distribution stage
: Before the firmware image can be distributed, the Initiator transfers the list of Target nodes and their designated firmware image index to the Distributor. Next, it tells the Distributor to start the firmware distributon process, which runs in the background while the Initiator and the mesh network perform other duties. Once the firmware image has been transferred to the Target nodes, the Distributor may ask them to apply the firmware image immediately and report back with their status and new firmware IDs.

Firmware images All updatable parts of a mesh node's firmware should be represented as a firmware image. Each Target node holds a list of firmware images, each of which should be independently updatable and identifiable.

Firmware images are represented as a BLOB (the firmware itself) with the following additional information attached to it:

Firmware ID
: The firmware ID is used to identify a firmware image. The Initiator node may ask the Target nodes for a list of its current firmware IDs to determine whether a newer version of the firmware is available. The format of the firmware ID is vendor specific, but generally, it should include enough information for an Initiator node with knowledge of the format to determine the type of image as well as its version. The firmware ID is optional, and its max length is determined by `CONFIG_BT_MESH_DFU_FWID_MAXLEN`.

Firmware metadata
: The firmware metadata is used by the Target node to determine whether it should accept an incoming firmware update, and what the effect of the update would be. The metadata format is vendor specific, and should contain all information the Target node needs to verify

the image, as well as any preparation the Target node has to make before the image is applied. Typical metadata information can be image signatures, changes to the node's Composition Data and the format of the BLOB. The Target node may perform a metadata check before accepting incoming transfers to determine whether the transfer should be started. The firmware metadata can be discarded by the Target node after the metadata check, as other nodes will never request the metadata from the Target node. The firmware metadata is optional, and its maximum length is determined by `CONFIG_BT_MESH_DFU_METADATA_MAXLEN`.

The Bluetooth Mesh DFU subsystem in Zephyr provides its own metadata format (*`bt_mesh_dfu_metadata`*) together with a set of related functions that can be used by an end product. The support for it is enabled using the `CONFIG_BT_MESH_DFU_METADATA` option. The format of the metadata is presented in the table below.

Field	Size (Bytes)	Description
New firmware version	8 B	1 B: Major version 1 B: Minor version 2 B: Revision 4 B: Build number
New firmware size	3 B	Size in bytes for a new firmware
New firmware core type	1 B	Bit field: Bit 0: Application core Bit 1: Network core Bit 2: Applications specific BLOB. Other bits: RFU
Hash of incoming composition data	4 B (Optional)	Lower 4 octets of AES-CMAC (app-specific-key, composition data). This field is present, if Bit 0 is set in the New firmware core type field.
New number of elements	2 B (Optional)	Number of elements on the node after firmware is applied. This field is present, if Bit 0 is set in the New firmware core type field.
Application-specific data for new firmware	<variable> (Optional)	Application-specific data to allow application to execute some vendor-specific behaviors using this data before it can respond with a status message.

Note

The AES-CMAC algorithm serves as a hashing function with a fixed key and is not used for encryption in Bluetooth Mesh DFU metadata. The resulting hash is not secure since the key is known.

Firmware URI

The firmware URI gives the Initiator information about where firmware updates for the image can be found. The URI points to an online resource the Initiator can interact with to get new versions of the firmware. This allows Initiators to perform updates for any node in the mesh network by interacting with the web server pointed to in the URI. The URI must point to a resource using the `http` or `https` schemes, and the targeted web server must behave according to the Firmware Check Over HTTPS procedure defined by the specification. The firmware URI is optional, and its max length is determined by `CONFIG_BT_MESH_DFU_URI_MAXLEN`.

Note

The out-of-band distribution mechanism is not supported.

Firmware effect A new image may have the Composition Data Page 0 different from the one allocated on a Target node. This may have an effect on the provisioning data of the node and

how the Distributor finalizes the DFU. Depending on the availability of the Remote Provisioning Server model on the old and new image, the device may either boot up unprovisioned after applying the new firmware or require to be re-provisioned. The complete list of available options is defined in *bt_mesh_dfu_effect*:

BT_MESH_DFU_EFFECT_NONE
: The device stays provisioned after the new firmware is programmed. This effect is chosen if the composition data of the new firmware doesn't change.

BT_MESH_DFU_EFFECT_COMP_CHANGE_NO_RPR
: This effect is chosen when the composition data changes and the device doesn't support the remote provisioning. The new composition data takes place only after re-provisioning.

BT_MESH_DFU_EFFECT_COMP_CHANGE
: This effect is chosen when the composition data changes and the device supports the remote provisioning. In this case, the device stays provisioned and the new composition data takes place after re-provisioning using the Remote Provisioning models.

BT_MESH_DFU_EFFECT_UNPROV
: This effect is chosen if the composition data in the new firmware changes, the device doesn't support the remote provisioning, and the new composition data takes effect after applying the firmware.

When the Target node receives the Firmware Update Firmware Metadata Check message, the Firmware Update Server model calls the *bt_mesh_dfu_srv_cb.check* callback, the application can then process the metadata and provide the effect value. If the effect is *BT_MESH_DFU_EFFECT_COMP_CHANGE*, the application must call functions *bt_mesh_comp_change_prepare()* and *bt_mesh_models_metadata_change_prepare()* to prepare the Composition Data Page and Models Metadata Page contents before applying the new firmware image. See *Composition Data and Models Metadata* for more information.

DFU procedures The DFU protocol is implemented as a set of procedures that must be performed in a certain order.

The Initiator controls the Upload stage of the DFU protocol, and all Distributor side handling of the upload subprocedures is implemented in the *Firmware Distribution Server*.

The Distribution stage is controlled by the Distributor, as implemented by the *Firmware Update Client*. The Target node implements all handling of these procedures in the *Firmware Update Server*, and notifies the application through a set of callbacks.

Uploading the firmware The Upload Firmware procedure uses the *BLOB Transfer models* to transfer the firmware image from the Initiator to the Distributor. The Upload Firmware procedure works in two steps:

1. The Initiator generates a BLOB ID, and sends it to the Distributor's Firmware Distribution Server along with the firmware information and other input parameters of the BLOB transfer. The Firmware Distribution Server stores the information, and prepares its BLOB Transfer Server for the incoming transfer before it responds with a status message to the Initiator.
2. The Initiator's BLOB Transfer Client model transfers the firmware image to the Distributor's BLOB Transfer Server, which stores the image in a predetermined flash partition.

When the BLOB transfer finishes, the firmware image is ready for distribution. The Initiator may upload several firmware images to the Distributor, and ask it to distribute them in any order or at any time. Additional procedures are available for querying and deleting firmware images from the Distributor.

The following Distributor's capabilities related to firmware images can be configured using the configuration options:

- `CONFIG_BT_MESH_DFU_SLOT_CNT`: Amount of image slots available on the device.

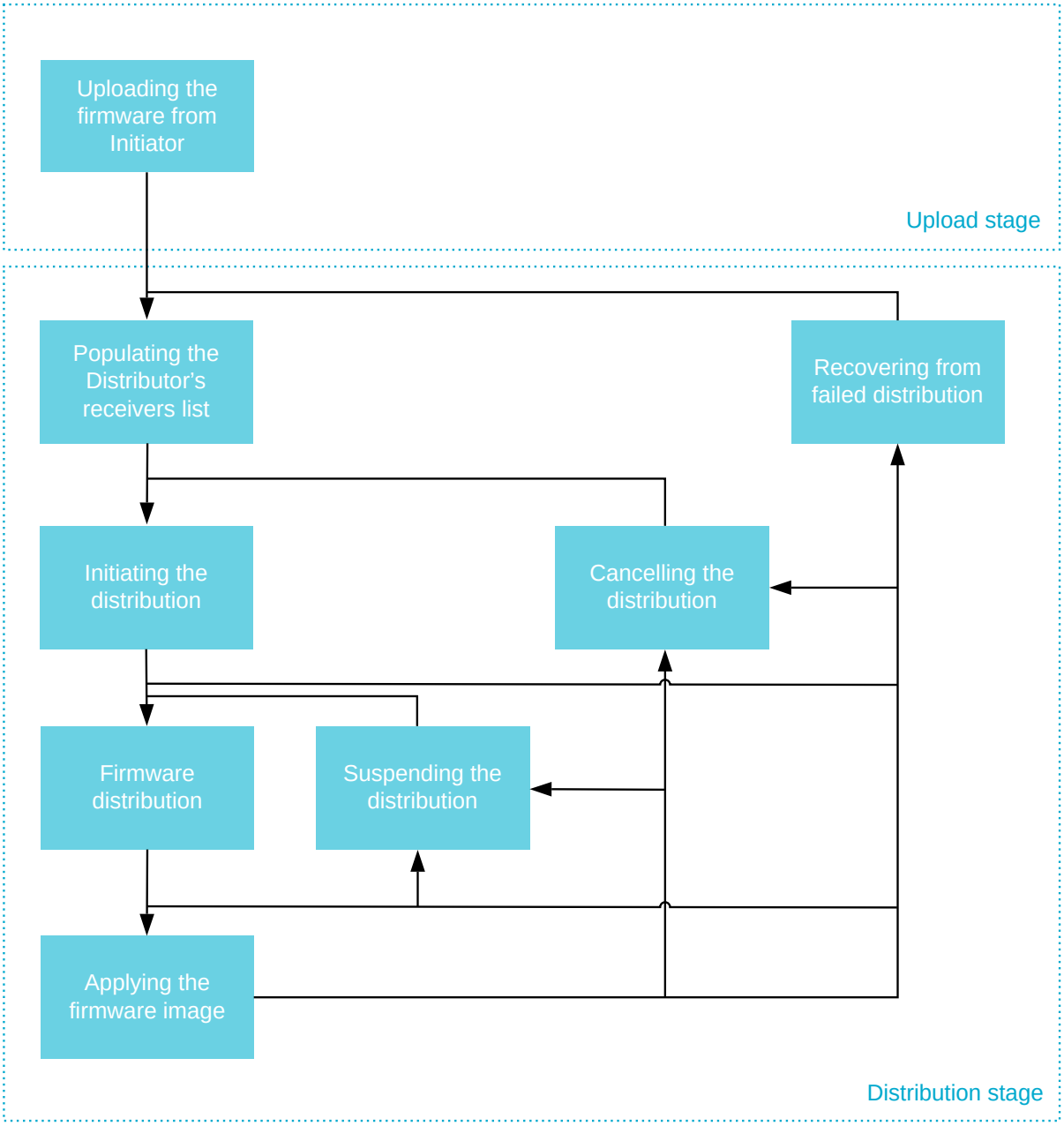


Fig. 11: DFU stages and procedures as seen from the Distributor

- `CONFIG_BT_MESH_DFD_SRV_SLOT_MAX_SIZE`: Maximum allowed size for each image.
- `CONFIG_BT_MESH_DFD_SRV_SLOT_SPACE`: Available space for all images.

Populating the Distributor's receivers list Before the Distributor can start distributing the firmware image, it needs a list of Target nodes to send the image to. The Initiator gets the full list of Target nodes either by querying the potential targets directly, or through some external authority. The Initiator uses this information to populate the Distributor's receivers list with the address and relevant firmware image index of each Target node. The Initiator may send one or more Firmware Distribution Receivers Add messages to build the Distributor's receivers list, and a Firmware Distribution Receivers Delete All message to clear it.

The maximum number of receivers that can be added to the Distributor is configured through the `CONFIG_BT_MESH_DFD_SRV_TARGETS_MAX` configuration option.

Initiating the distribution Once the Distributor has stored a firmware image and received a list of Target nodes, the Initiator may initiate the distribution procedure. The BLOB transfer parameters for the distribution are passed to the Distributor along with an update policy. The update policy decides whether the Distributor should request that the firmware is applied on the Target nodes or not. The Distributor stores the transfer parameters and starts distributing the firmware image to its list of Target nodes.

Firmware distribution The Distributor's Firmware Update Client model uses its BLOB Transfer Client model's broadcast subsystem to communicate with all Target nodes. The firmware distribution is performed with the following steps:

1. The Distributor's Firmware Update Client model generates a BLOB ID and sends it to each Target node's Firmware Update Server model, along with the other BLOB transfer parameters, the Target node firmware image index and the firmware image metadata. Each Target node performs a metadata check and prepares their BLOB Transfer Server model for the transfer, before sending a status response to the Firmware Update Client, indicating if the firmware update will have any effect on the Bluetooth Mesh state of the node.
2. The Distributor's BLOB Transfer Client model transfers the firmware image to all Target nodes.
3. Once the BLOB transfer has been received, the Target nodes' applications verify that the firmware is valid by performing checks such as signature verification or image checksums against the image metadata.
4. The Distributor's Firmware Update Client model queries all Target nodes to ensure that they've all verified the firmware image.

If the distribution procedure completed with at least one Target node reporting that the image has been received and verified, the distribution procedure is considered successful.

Note

The firmware distribution procedure only fails if *all* Target nodes are lost. It is up to the Initiator to request a list of failed Target nodes from the Distributor and initiate additional attempts to update the lost Target nodes after the current attempt is finished.

Suspending the distribution The Initiator can also request the Distributor to suspend the firmware distribution. In this case, the Distributor will stop sending any messages to Target nodes. When the firmware distribution is resumed, the Distributor will continue sending the firmware from the last successfully transferred block.

Applying the firmware image If the Initiator requested it, the Distributor can initiate the Apply Firmware on Target Node procedure on all Target nodes that successfully received and verified the firmware image. The Apply Firmware on Target Node procedure takes no parameters, and to avoid ambiguity, it should be performed before a new transfer is initiated. The Apply Firmware on Target Node procedure consists of the following steps:

1. The Distributor's Firmware Update Client model instructs all Target nodes that have verified the firmware image to apply it. The Target nodes' Firmware Update Server models respond with a status message before calling their application's `apply` callback.
2. The Target node's application performs any preparations needed before applying the transfer, such as storing a snapshot of the Composition Data or clearing its configuration.
3. The Target node's application swaps the current firmware with the new image and updates its firmware image list with the new firmware ID.
4. The Distributor's Firmware Update Client model requests the full list of firmware images from each Target node, and scans through the list to make sure that the new firmware ID has replaced the old.

Note

During the metadata check in the distribution procedure, the Target node may have reported that it will become unprovisioned after the firmware image is applied. In this case, the Distributor's Firmware Update Client model will send a request for the full firmware image list, and expect no response.

Cancelling the distribution The firmware distribution can be cancelled at any time by the Initiator. In this case, the Distributor starts the cancelling procedure by sending a cancelling message to all Target nodes. The Distributor waits for the response from all Target nodes. Once all Target nodes have replied, or the request has timed out, the distribution procedure is cancelled. After this the distribution procedure can be started again from the `Firmware distribution` section.

API reference This section lists the types common to the Device Firmware Update mesh models.

group `bt_mesh_dfd`

Enums

enum `bt_mesh_dfd_status`

Firmware distribution status.

Values:

enumerator `BT_MESH_DFD_SUCCESS`

The message was processed successfully.

enumerator `BT_MESH_DFD_ERR_INSUFFICIENT_RESOURCES`

Insufficient resources on the node.

enumerator BT_MESH_DFD_ERR_WRONG_PHASE
The operation cannot be performed while the Server is in the current phase.

enumerator BT_MESH_DFD_ERR_INTERNAL
An internal error occurred on the node.

enumerator BT_MESH_DFD_ERR_FW_NOT_FOUND
The requested firmware image is not stored on the Distributor.

enumerator BT_MESH_DFD_ERR_INVALID_APPKEY_INDEX
The AppKey identified by the AppKey Index is not known to the node.

enumerator BT_MESH_DFD_ERR_RECEIVERS_LIST_EMPTY
There are no Target nodes in the Distribution Receivers List state.

enumerator BT_MESH_DFD_ERR_BUSY_WITH_DISTRIBUTION
Another firmware image distribution is in progress.

enumerator BT_MESH_DFD_ERR_BUSY_WITH_UPLOAD
Another upload is in progress.

enumerator BT_MESH_DFD_ERR_URI_NOT_SUPPORTED
The URI scheme name indicated by the Update URI is not supported.

enumerator BT_MESH_DFD_ERR_URI_MALFORMED
The format of the Update URI is invalid.

enumerator BT_MESH_DFD_ERR_URI_UNREACHABLE
The URI is currently unreachable.

enumerator BT_MESH_DFD_ERR_NEW_FW_NOT_AVAILABLE
The Check Firmware OOB procedure did not find any new firmware.

enumerator BT_MESH_DFD_ERR_SUSPEND_FAILED
The suspension of the Distribute Firmware procedure failed.

enum bt_mesh_dfd_phase
Firmware distribution phases.
Values:

enumerator BT_MESH_DFD_PHASE_IDLE
No firmware distribution is in progress.

enumerator BT_MESH_DFD_PHASE_TRANSFER_ACTIVE
Firmware distribution is in progress.

enumerator BT_MESH_DFD_PHASE_TRANSFER_SUCCESS
The Transfer BLOB procedure has completed successfully.

enumerator BT_MESH_DFD_PHASE_APPLYING_UPDATE
The Apply Firmware on Target Nodes procedure is being executed.

enumerator BT_MESH_DFD_PHASE_COMPLETED
The Distribute Firmware procedure has completed successfully.

enumerator BT_MESH_DFD_PHASE_FAILED
The Distribute Firmware procedure has failed.

enumerator BT_MESH_DFD_PHASE_CANCELING_UPDATE
The Cancel Firmware Update procedure is being executed.

enumerator BT_MESH_DFD_PHASE_TRANSFER_SUSPENDED
The Transfer BLOB procedure is suspended.

enum bt_mesh_dfd_upload_phase
Firmware upload phases.
Values:

enumerator BT_MESH_DFD_UPLOAD_PHASE_IDLE
No firmware upload is in progress.

enumerator BT_MESH_DFD_UPLOAD_PHASE_TRANSFER_ACTIVE
The Store Firmware procedure is being executed.

enumerator BT_MESH_DFD_UPLOAD_PHASE_TRANSFER_ERROR
The Store Firmware procedure or Store Firmware OOB procedure failed.

enumerator BT_MESH_DFD_UPLOAD_PHASE_TRANSFER_SUCCESS
The Store Firmware procedure or the Store Firmware OOB procedure completed successfully.

group bt_mesh_dfu

Defines

CONFIG_BT_MESH_DFU_FWID_MAXLEN

CONFIG_BT_MESH_DFU_METADATA_MAXLEN

CONFIG_BT_MESH_DFU_URI_MAXLEN

CONFIG_BT_MESH_DFU_SLOT_CNT

Enums

enum bt_mesh_dfu_phase

DFU transfer phase.

Values:

enumerator BT_MESH_DFU_PHASE_IDLE

Ready to start a Receive Firmware procedure.

enumerator BT_MESH_DFU_PHASE_TRANSFER_ERR

The Transfer BLOB procedure failed.

enumerator BT_MESH_DFU_PHASE_TRANSFER_ACTIVE

The Receive Firmware procedure is being executed.

enumerator BT_MESH_DFU_PHASE_VERIFY

The Verify Firmware procedure is being executed.

enumerator BT_MESH_DFU_PHASE_VERIFY_OK

The Verify Firmware procedure completed successfully.

enumerator BT_MESH_DFU_PHASE_VERIFY_FAIL

The Verify Firmware procedure failed.

enumerator BT_MESH_DFU_PHASE_APPLYING

The Apply New Firmware procedure is being executed.

enumerator BT_MESH_DFU_PHASE_TRANSFER_CANCELED

Firmware transfer has been canceled.

enumerator BT_MESH_DFU_PHASE_APPLY_SUCCESS

Firmware applying succeeded.

enumerator BT_MESH_DFU_PHASE_APPLY_FAIL

Firmware applying failed.

enumerator BT_MESH_DFU_PHASE_UNKNOWN

Phase of a node was not yet retrieved.

enum bt_mesh_dfu_status

DFU status.

Values:

enumerator BT_MESH_DFU_SUCCESS

The message was processed successfully.

enumerator BT_MESH_DFU_ERR_RESOURCES

Insufficient resources on the node.

enumerator BT_MESH_DFU_ERR_WRONG_PHASE

The operation cannot be performed while the Server is in the current phase.

enumerator BT_MESH_DFU_ERR_INTERNAL

An internal error occurred on the node.

enumerator BT_MESH_DFU_ERR_FW_IDX

The message contains a firmware index value that is not expected.

enumerator BT_MESH_DFU_ERR_METADATA

The metadata check failed.

enumerator BT_MESH_DFU_ERR_TEMPORARILY_UNAVAILABLE

The Server cannot start a firmware update.

enumerator BT_MESH_DFU_ERR_BLOB_XFER_BUSY

Another BLOB transfer is in progress.

enum bt_mesh_dfu_effect

Expected effect of a DFU transfer.

Values:

enumerator BT_MESH_DFU_EFFECT_NONE

No changes to node Composition Data.

enumerator BT_MESH_DFU_EFFECT_COMP_CHANGE_NO_RPR

Node Composition Data changed and the node does not support remote provisioning.

enumerator BT_MESH_DFU_EFFECT_COMP_CHANGE

Node Composition Data changed, and remote provisioning is supported.

The node supports remote provisioning and Composition Data Page 0x80. Page 0x80 contains different Composition Data than Page 0x0.

enumerator BT_MESH_DFU_EFFECT_UNPROV

Node will be unprovisioned after the update.

enum bt_mesh_dfu_iter

Action for DFU iteration callbacks.

Values:

enumerator BT_MESH_DFU_ITER_STOP

Stop iterating.

enumerator BT_MESH_DFU_ITER_CONTINUE

Continue iterating.

struct **bt_mesh_dfu_img**

#include <*dfu.h*> DFU image instance.

Each DFU image represents a single updatable firmware image.

Public Members

const void ***fwid**

Firmware ID.

size_t **fwid_len**

Length of the firmware ID.

const char ***uri**

Update URI, or NULL.

struct **bt_mesh_dfu_slot**

#include <*dfu.h*> DFU image slot for DFU distribution.

Public Members

size_t **size**

Size of the firmware in bytes.

size_t **fwid_len**

Length of the firmware ID.

size_t **metadata_len**

Length of the metadata.

uint8_t **fwid**[0]

Firmware ID.

uint8_t **metadata**[0]

Metadata.

group **bt_mesh_dfu_metadata**

Common types and functions for the Bluetooth Mesh DFU metadata.

Enums

enum **bt_mesh_dfu_metadata_fw_core_type**

Firmware core type.

Values:

enumerator **BT_MESH_DFU_FW_CORE_TYPE_APP** = *BIT*(0)

Application core.

enumerator **BT_MESH_DFU_FW_CORE_TYPE_NETWORK** = *BIT*(1)

Network core.

enumerator **BT_MESH_DFU_FW_CORE_TYPE_APP_SPECIFIC_BLOB** = *BIT*(2)

Application-specific BLOB.

Functions

int **bt_mesh_dfu_metadata_decode**(struct *net_buf_simple* *buf, struct *bt_mesh_dfu_metadata* *metadata)

Decode a firmware metadata from a network buffer.

Parameters

- **buf** – Buffer containing a raw metadata to be decoded.
- **metadata** – Pointer to a metadata structure to be filled.

Returns

0 on success, or (negative) error code otherwise.

int **bt_mesh_dfu_metadata_encode**(const struct *bt_mesh_dfu_metadata* *metadata, struct *net_buf_simple* *buf)

Encode a firmware metadata into a network buffer.

Parameters

- **metadata** – Firmware metadata to be encoded.
- **buf** – Buffer to store the encoded metadata.

Returns

0 on success, or (negative) error code otherwise.

int **bt_mesh_dfu_metadata_comp_hash_get**(struct *net_buf_simple* *buf, uint8_t *key, uint32_t *hash)

Compute hash of the Composition Data state.

The format of the Composition Data is defined in MshPRTv1.1: 4.2.2.1.

Parameters

- **buf** – Pointer to buffer holding Composition Data.
- **key** – 128-bit key to be used in the hash computation.
- **hash** – Pointer to a memory location to which the hash will be stored.

Returns

0 on success, or (negative) error code otherwise.

int **bt_mesh_dfu_metadata_comp_hash_local_get**(uint8_t *key, uint32_t *hash)

Compute hash of the Composition Data Page 0 of this device.

Parameters

- **key** – 128-bit key to be used in the hash computation.
- **hash** – Pointer to a memory location to which the hash will be stored.

Returns
0 on success, or (negative) error code otherwise.

struct **bt_mesh_dfu_metadata_fw_ver**

#include <dfu_metadata.h> Firmware version.

Public Members

uint8_t **major**

Firmware major version.

uint8_t **minor**

Firmware minor version.

uint16_t **revision**

Firmware revision.

uint32_t **build_num**

Firmware build number.

struct **bt_mesh_dfu_metadata**

#include <dfu_metadata.h> Firmware metadata.

Public Members

struct *bt_mesh_dfu_metadata_fw_ver* **fw_ver**

New firmware version.

uint32_t **fw_size**

New firmware size.

enum *bt_mesh_dfu_metadata_fw_core_type* **fw_core_type**

New firmware core type.

uint32_t **comp_hash**

Hash of incoming Composition Data.

uint16_t **elems**

New number of node elements.

uint8_t ***user_data**

Application-specific data for new firmware.

This field is optional.

uint32_t **user_data_len**

Length of the application-specific field.

Message The Bluetooth Mesh message provides set of structures, macros and functions used for preparing message buffers, managing message and acknowledged message contexts.

API reference

group `bt_mesh_msg`

Message.

Defines

`BT_MESH_MIC_SHORT`

Length of a short Mesh MIC.

`BT_MESH_MIC_LONG`

Length of a long Mesh MIC.

`BT_MESH_MODEL_OP_LEN`(_op)

Helper to determine the length of an opcode.

Parameters

- **_op** – Opcode.

`BT_MESH_MODEL_BUF_LEN`(_op, _payload_len)

Helper for model message buffer length.

Returns the length of a Mesh model message buffer, including the opcode length and a short MIC.

Parameters

- **_op** – Opcode of the message.
- **_payload_len** – Length of the model payload.

`BT_MESH_MODEL_BUF_LEN_LONG_MIC`(_op, _payload_len)

Helper for model message buffer length.

Returns the length of a Mesh model message buffer, including the opcode length and a long MIC.

Parameters

- **_op** – Opcode of the message.
- **_payload_len** – Length of the model payload.

`BT_MESH_MODEL_BUF_DEFINE`(_buf, _op, _payload_len)

Define a Mesh model message buffer using *NET_BUF_SIMPLE_DEFINE*.

Parameters

- **_buf** – Buffer name.
- **_op** – Opcode of the message.
- **_payload_len** – Length of the model message payload.

`BT_MESH_MSG_CTX_INIT`(net_key_idx, app_key_idx, dst, ttl)

Helper for *bt_mesh_msg_ctx* structure initialization.

> **Note**
>
> If `dst` is a Virtual Address, Label UUID shall be initialized separately.

Parameters

- **net_key_idx** – NetKey Index of the subnet to send the message on. Only used if `app_key_idx` points to devkey.
- **app_key_idx** – AppKey Index to encrypt the message with.
- **dst** – Remote addr.
- **ttl** – Time To Live.

`BT_MESH_MSG_CTX_INIT_APP`(app_key_idx, dst)

Helper for *bt_mesh_msg_ctx* structure initialization secured with Application Key.

Parameters

- **app_key_idx** – AppKey Index to encrypt the message with.
- **dst** – Remote addr.

`BT_MESH_MSG_CTX_INIT_DEV`(net_key_idx, dst)

Helper for *bt_mesh_msg_ctx* structure initialization secured with Device Key of a remote device.

Parameters

- **net_key_idx** – NetKey Index of the subnet to send the message on.
- **dst** – Remote addr.

`BT_MESH_MSG_CTX_INIT_PUB`(pub)

Helper for *bt_mesh_msg_ctx* structure initialization using Model Publication context.

Parameters

- **pub** – Pointer to a model publication context.

Functions

void **bt_mesh_model_msg_init**(struct *net_buf_simple* *msg, uint32_t opcode)

Initialize a model message.

Clears the message buffer contents, and encodes the given opcode. The message buffer will be ready for filling in payload data.

Parameters

- **msg** – Message buffer.
- **opcode** – Opcode to encode.

static inline void **bt_mesh_msg_ack_ctx_init**(struct *bt_mesh_msg_ack_ctx* *ack)

Initialize an acknowledged message context.

Initializes semaphore used for synchronization between *bt_mesh_msg_ack_ctx_wait* and *bt_mesh_msg_ack_ctx_rx* calls. Call this function before using *bt_mesh_msg_ack_ctx*.

Parameters

- **ack** – Acknowledged message context to initialize.

static inline void **bt_mesh_msg_ack_ctx_reset**(struct *bt_mesh_msg_ack_ctx* *ack)

Reset the synchronization semaphore in an acknowledged message context.

This function aborts call to *bt_mesh_msg_ack_ctx_wait*.

Parameters

- **ack** – Acknowledged message context to be reset.

void **bt_mesh_msg_ack_ctx_clear**(struct *bt_mesh_msg_ack_ctx* *ack)

Clear parameters of an acknowledged message context.

This function clears the opcode, remote address and user data set by *bt_mesh_msg_ack_ctx_prepare*.

Parameters

- **ack** – Acknowledged message context to be cleared.

int **bt_mesh_msg_ack_ctx_prepare**(struct *bt_mesh_msg_ack_ctx* *ack, uint32_t op, uint16_t dst, void *user_data)

Prepare an acknowledged message context for the incoming message to wait.

This function sets the opcode, remote address of the incoming message and stores the user data. Use this function before calling *bt_mesh_msg_ack_ctx_wait*.

Parameters

- **ack** – Acknowledged message context to prepare.
- **op** – The message OpCode.
- **dst** – Destination address of the message.
- **user_data** – User data for the acknowledged message context.

Returns

0 on success, or (negative) error code on failure.

static inline bool **bt_mesh_msg_ack_ctx_busy**(struct *bt_mesh_msg_ack_ctx* *ack)

Check if the acknowledged message context is initialized with an opcode.

Parameters

- **ack** – Acknowledged message context.

Returns

true if the acknowledged message context is initialized with an opcode, false otherwise.

int **bt_mesh_msg_ack_ctx_wait**(struct *bt_mesh_msg_ack_ctx* *ack, *k_timeout_t* timeout)

Wait for a message acknowledge.

This function blocks execution until *bt_mesh_msg_ack_ctx_rx* is called or by timeout.

Parameters

- **ack** – Acknowledged message context of the message to wait for.
- **timeout** – Wait timeout.

Returns

0 on success, or (negative) error code on failure.

static inline void **bt_mesh_msg_ack_ctx_rx**(struct *bt_mesh_msg_ack_ctx* *ack)

Mark a message as acknowledged.

This function unblocks call to *bt_mesh_msg_ack_ctx_wait*.

Parameters

- **ack** – Context of a message to be acknowledged.

bool **bt_mesh_msg_ack_ctx_match**(const struct *bt_mesh_msg_ack_ctx* *ack, uint32_t op, uint16_t addr, void **user_data)

Check if an opcode and address of a message matches the expected one.

Parameters

- **ack** – Acknowledged message context to be checked.
- **op** – OpCode of the incoming message.
- **addr** – Source address of the incoming message.
- **user_data** – If not NULL, returns a user data stored in the acknowledged message context by *bt_mesh_msg_ack_ctx_prepare*.

Returns

true if the incoming message matches the expected one, false otherwise.

struct **bt_mesh_msg_ctx**

#include <msg.h> Message sending context.

Public Members

uint16_t **net_idx**

NetKey Index of the subnet to send the message on.

uint16_t **app_idx**

AppKey Index to encrypt the message with.

uint16_t **addr**

Remote address.

uint16_t **recv_dst**

Destination address of a received message.

Not used for sending.

const uint8_t ***uuid**

Label UUID if Remote address is Virtual address, or NULL otherwise.

int8_t **recv_rssi**

RSSI of received packet.

Not used for sending.

uint8_t **recv_ttl**

Received TTL value.

Not used for sending.

bool **send_rel**

Force sending reliably by using segment acknowledgment.

bool **rnd_delay**

Send message with a random delay according to the Access layer transmitting rules.

uint8_t **send_ttl**

TTL, or BT_MESH_TTL_DEFAULT for default TTL.

struct **bt_mesh_msg_ack_ctx**

#include <msg.h> Acknowledged message context for tracking the status of model messages pending a response.

Public Members

struct k_sem **sem**

Sync semaphore.

uint32_t **op**

Opcode we're waiting for.

uint16_t **dst**

Address of the node that should respond.

void ***user_data**

User specific parameter.

Segmentation and reassembly (SAR) Segmentation and reassembly (SAR) provides a way of handling larger upper transport layer messages in a mesh network, with a purpose of enhancing the Bluetooth Mesh throughput. The segmentation and reassembly mechanism is used by the lower transport layer.

The lower transport layer defines how the upper transport layer PDUs are segmented and reassembled into multiple Lower Transport PDUs, and sends them to the lower transport layer on a peer device. If the Upper Transport PDU fits, it is sent in a single Lower Transport PDU. For longer packets, which do not fit into a single Lower Transport PDU, the lower transport layer performs segmentation, splitting the Upper Transport PDU into multiple segments.

The lower transport layer on the receiving device reassembles the segments into a single Upper Transport PDU before passing it up the stack. Delivery of a segmented message is acknowledged by the lower transport layer of the receiving node, while an unsegmented message delivery is not acknowledged. However, an Upper Transport PDU that fits into one Lower Transport PDU can also be sent as a single-segment segmented message when acknowledgment by the lower transport layer is required. Set the `send rel` flag (see *bt_mesh_msg_ctx*) to use the reliable message transmission and acknowledge single-segment segmented messages.

The transport layer is able to transport up to 32 segments with its SAR mechanism, with a maximum message (PDU) size of 384 octets. To configure message size for the Bluetooth Mesh stack, use the following Kconfig options:

- `CONFIG_BT_MESH_RX_SEG_MAX` to set the maximum number of segments in an incoming message.
- `CONFIG_BT_MESH_TX_SEG_MAX` to set the maximum number of segments in an outgoing message.

The Kconfig options CONFIG_BT_MESH_TX_SEG_MSG_COUNT and CONFIG_BT_MESH_RX_SEG_MSG_COUNT define how many outgoing and incoming segmented messages can be processed simultaneously. When more than one segmented message is sent to the same destination, the messages are queued and sent one at a time.

Incoming and outgoing segmented messages share the same pool for allocation of their segments. This pool size is configured through the CONFIG_BT_MESH_SEG_BUFS Kconfig option. Both incoming and outgoing messages allocate segments at the start of the transaction. The outgoing segmented message releases its segments one by one as soon as they are acknowledged by the receiver, while the incoming message releases the segments first after the message is fully received. Keep this in mind when defining the size of the buffers.

SAR does not impose extra overhead on the access layer payload per segment.

Segmentation and reassembly (SAR) Configuration models With Bluetooth Mesh Protocol Specification version 1.1, it became possible to configure SAR behavior, such as intervals, timers and retransmission counters, over a mesh network using SAR Configuration models:

- *SAR Configuration Client*
- *SAR Configuration Server*

The following SAR behavior applies regardless of the presence of a SAR Configuration Server on a node.

Transmission of segments is separated by a segment transmission interval (see the *SAR Segment Interval Step* state). Other configurable time intervals and delays available for the segmentation and reassembly are:

- Interval between unicast retransmissions (see the states *SAR Unicast Retransmissions Interval Step* and *SAR Unicast Retransmissions Interval Increment*).
- Interval between multicast retransmissions (see the *SAR Multicast Retransmissions Interval Step* state).
- Segment reception interval (see the *SAR Receiver Segment Interval Step* state).
- Acknowledgment delay increment (see the *SAR Acknowledgment Delay Increment* state).

When the last segment marked as unacknowledged is transmitted, the lower transport layer starts a retransmissions timer. The initial value of the SAR Unicast Retransmissions timer depends on the value of the TTL field of the message. If the TTL field value is greater than 0, the initial value for the timer is set according to the following formula:

$$unicast\ retransmissions\ interval\ step + unicast\ retransmissions\ interval\ increment \times (TTL - 1)$$

If the TTL field value is 0, the initial value of the timer is set to the unicast retransmissions interval step.

The initial value of the SAR Multicast Retransmissions timer is set to the multicast retransmissions interval.

When the lower transport layer receives a message segment, it starts a SAR Discard timer. The discard timer tells how long the lower transport layer waits before discarding the segmented message the segment belongs to. The initial value of the SAR Discard timer is the discard timeout value indicated by the *SAR Discard Timeout* state.

SAR Acknowledgment timer holds the time before a Segment Acknowledgment message is sent for a received segment. The initial value of the SAR Acknowledgment timer is calculated using the following formula:

$$min(SegN + 0.5, acknowledgment\ delay\ increment) \times segment\ reception\ interval$$

The SegN field value identifies the total number of segments the Upper Transport PDU is segmented into.

Four counters are related to SAR behavior:

- Two unicast retransmissions counts (see *SAR Unicast Retransmissions Count* state and *SAR Unicast Retransmissions Without Progress Count* state)
- Multicast retransmissions count (see *SAR Multicast Retransmissions Count* state)
- Acknowledgment retransmissions count (see *SAR Acknowledgment Retransmissions Count* state)

If the number of segments in the transmission is higher than the value of the *SAR Segments Threshold* state, Segment Acknowledgment messages are retransmitted using the value of the *SAR Acknowledgment Retransmissions Count* state.

SAR states There are two states defined related to segmentation and reassembly:

- SAR Transmitter state
- SAR Receiver state

The SAR Transmitter state is a composite state that controls the number and timing of transmissions of segmented messages. It includes the following states:

- SAR Segment Interval Step
- SAR Unicast Retransmissions Count
- SAR Unicast Retransmissions Without Progress Count
- SAR Unicast Retransmissions Interval Step
- SAR Unicast Retransmissions Interval Increment
- SAR Multicast Retransmissions Count
- SAR Multicast Retransmissions Interval Step

The SAR Receiver state is a composite state that controls the number and timing of Segment Acknowledgment transmissions and the discarding of reassembly of a segmented message. It includes the following states:

- SAR Segments Threshold
- SAR Discard Timeout
- SAR Acknowledgment Delay Increment
- SAR Acknowledgment Retransmissions Count
- SAR Receiver Segment Interval Step

SAR Segment Interval Step SAR Segment Interval Step state holds a value that controls the interval between transmissions of segments of a segmented message. The interval is measured in milliseconds.

Use the `CONFIG_BT_MESH_SAR_TX_SEG_INT_STEP` Kconfig option to set the default value. Segment transmission interval is then calculated using the following formula:

$$(\texttt{CONFIG\_BT\_MESH\_SAR\_TX\_SEG\_INT\_STEP} + 1) \times 10\,\text{ms}$$

SAR Unicast Retransmissions Count SAR Unicast Retransmissions Count holds a value that defines the maximum number of retransmissions of a segmented message to a unicast destination. Use the `CONFIG_BT_MESH_SAR_TX_UNICAST_RETRANS_COUNT` Kconfig option to set the default value for this state.

SAR Unicast Retransmissions Without Progress Count This state holds a value that defines the maximum number of retransmissions of a segmented message to a unicast address that will be sent if no acknowledgment was received during the timeout, or if an acknowledgment with already confirmed segments was received. Use the Kconfig option CONFIG_BT_MESH_SAR_TX_UNICAST_RETRANS_WITHOUT_PROG_COUNT to set the maximum number of retransmissions.

SAR Unicast Retransmissions Interval Step The value of this state controls the interval step used for delaying the retransmissions of unacknowledged segments of a segmented message to a unicast address. The interval step is measured in milliseconds.

Use the CONFIG_BT_MESH_SAR_TX_UNICAST_RETRANS_INT_STEP Kconfig option to set the default value. This value is then used to calculate the interval step using the following formula:

$$(\texttt{CONFIG\_BT\_MESH\_SAR\_TX\_UNICAST\_RETRANS\_INT\_STEP} + 1) \times 25 \text{ ms}$$

SAR Unicast Retransmissions Interval Increment SAR Unicast Retransmissions Interval Increment holds a value that controls the interval increment used for delaying the retransmissions of unacknowledged segments of a segmented message to a unicast address. The increment is measured in milliseconds.

Use the Kconfig option CONFIG_BT_MESH_SAR_TX_UNICAST_RETRANS_INT_INC to set the default value. The Kconfig option value is used to calculate the increment using the following formula:

$$(\texttt{CONFIG\_BT\_MESH\_SAR\_TX\_UNICAST\_RETRANS\_INT\_INC} + 1) \times 25 \text{ ms}$$

SAR Multicast Retransmissions Count The state holds a value that controls the total number of retransmissions of a segmented message to a multicast address. Use the Kconfig option CONFIG_BT_MESH_SAR_TX_MULTICAST_RETRANS_COUNT to set the total number of retransmissions.

SAR Multicast Retransmissions Interval Step This state holds a value that controls the interval between retransmissions of all segments in a segmented message to a multicast address. The interval is measured in milliseconds.

Use the Kconfig option CONFIG_BT_MESH_SAR_TX_MULTICAST_RETRANS_INT to set the default value that is used to calculate the interval using the following formula:

$$(\texttt{CONFIG\_BT\_MESH\_SAR\_TX\_MULTICAST\_RETRANS\_INT} + 1) \times 25 \text{ ms}$$

SAR Discard Timeout The value of this state defines the time in seconds that the lower transport layer waits after receiving segments of a segmented message before discarding that segmented message. Use the Kconfig option CONFIG_BT_MESH_SAR_RX_DISCARD_TIMEOUT to set the default value. The discard timeout will be calculated using the following formula:

$$(\texttt{CONFIG\_BT\_MESH\_SAR\_RX\_DISCARD\_TIMEOUT} + 1) \times 5 \text{ seconds}$$

SAR Acknowledgment Delay Increment This state holds a value that controls the delay increment of an interval used for delaying the transmission of an acknowledgment message after receiving a new segment. The increment is measured in segments.

Use the Kconfig option CONFIG_BT_MESH_SAR_RX_ACK_DELAY_INC to set the default value. The increment value is calculated to be CONFIG_BT_MESH_SAR_RX_ACK_DELAY_INC $+ 1.5$.

SAR Segments Threshold SAR Segments Threshold state holds a value that defines a threshold in number of segments of a segmented message for acknowledgment retransmissions. Use the Kconfig option CONFIG_BT_MESH_SAR_RX_SEG_THRESHOLD to set the threshold.

When the number of segments of a segmented message is above this threshold, the stack will additionally retransmit every acknowledgment message the number of times given by the value of CONFIG_BT_MESH_SAR_RX_ACK_RETRANS_COUNT.

SAR Acknowledgment Retransmissions Count The SAR Acknowledgment Retransmissions Count state controls the number of retransmissions of Segment Acknowledgment messages sent by the lower transport layer. It gives the total number of retranmissions of an acknowledgment message that the stack will additionally send when the size of segments in a segmented message is above the CONFIG_BT_MESH_SAR_RX_SEG_THRESHOLD value.

Use the Kconfig option CONFIG_BT_MESH_SAR_RX_ACK_RETRANS_COUNT to set the default value for this state. The maximum number of transmissions of a Segment Acknowledgment message is CONFIG_BT_MESH_SAR_RX_ACK_RETRANS_COUNT $+ 1$.

SAR Receiver Segment Interval Step The SAR Receiver Segment Interval Step defines the segments reception interval step used for delaying the transmission of an acknowledgment message after receiving a new segment. The interval is measured in milliseconds.

Use the Kconfig option CONFIG_BT_MESH_SAR_RX_SEG_INT_STEP to set the default value and calculate the interval using the following formula:

$$(\texttt{CONFIG\_BT\_MESH\_SAR\_RX\_SEG\_INT\_STEP} + 1) \times 10 \text{ ms}$$

Provisioning Provisioning is the process of adding devices to a mesh network. It requires two devices operating in the following roles:

- The *provisioner* represents the network owner, and is responsible for adding new nodes to the mesh network.
- The *provisionee* is the device that gets added to the network through the Provisioning process. Before the provisioning process starts, the provisionee is an *unprovisioned device*.

The Provisioning module in the Zephyr Bluetooth Mesh stack supports both the Advertising and GATT Provisioning bearers for the provisionee role, as well as the Advertising Provisioning bearer for the provisioner role.

The Provisioning process All Bluetooth Mesh nodes must be provisioned before they can participate in a Bluetooth Mesh network. The Provisioning API provides all the functionality necessary for a device to become a provisioned mesh node. Provisioning is a five-step process, involving the following steps:

- Beaconing
- Invitation
- Public key exchange
- Authentication
- Provisioning data transfer

Beaconing To start the provisioning process, the unprovisioned device must first start broadcasting the Unprovisioned Beacon. This makes it visible to nearby provisioners, which can initiate the provisioning. To indicate that the device needs to be provisioned, call *bt_mesh_prov_enable()*. The device starts broadcasting the Unprovisioned Beacon with the device UUID and the OOB information field, as specified in the prov parameter passed to *bt_mesh_init()*. Additionally, a Uniform Resource Identifier (URI) may be specified, which can point the provisioner to the location of some Out Of Band information, such as the device's public key or an authentication value database. The URI is advertised in a separate beacon, with a URI hash included in the unprovisioned beacon, to tie the two together.

Uniform Resource Identifier The Uniform Resource Identifier shall follow the format specified in the Bluetooth Core Specification Supplement. The URI must start with a URI scheme, encoded as a single utf-8 data point, or the special none scheme, encoded as 0x01. The available schemes are listed on the Bluetooth website.

Examples of encoded URIs:

Table 28: URI encoding examples

URI	Encoded
http://example.com	\x16//example.com
https://www.zephyrproject.org/	\x17//www.zephyrproject.org/
just a string	\x01just a string

Provisioning invitation The provisioner initiates the Provisioning process by sending a Provisioning invitation. The invitations prompts the provisionee to call attention to itself using the Health Server *Attention state*, if available.

The Unprovisioned device automatically responds to the invite by presenting a list of its capabilities, including the supported Out of Band Authentication methods and algorithms.

Public key exchange Before the provisioning process can begin, the provisioner and the unprovisioned device exchange public keys, either in-band or Out of Band (OOB).

In-band public key exchange is a part of the provisioning process and always supported by the unprovisioned device and provisioner.

If the application wants to support public key exchange via OOB, it needs to provide public and private keys to the mesh stack. The unprovisioned device will reflect this in its capabilities. The provisioner obtains the public key via any available OOB mechanism (e.g. the device may advertise a packet containing the public key or it can be encoded in a QR code printed on the device packaging). Note that even if the unprovisioned device has specified the public key for the Out of Band exchange, the provisioner may choose to exchange the public key in-band if it can't retrieve the public key via OOB mechanism. In this case, a new key pair will be generated by the mesh stack for each Provisioning process.

To enable support of OOB public key on the unprovisioned device side, CONFIG_BT_MESH_PROV_OOB_PUBLIC_KEY needs to be enabled. The application must provide public and private keys before the Provisioning process is started by initializing pointers to *bt_mesh_prov.public_key_be* and *bt_mesh_prov.private_key_be*. The keys needs to be provided in big-endian bytes order.

To provide the device's public key obtained via OOB, call *bt_mesh_prov_remote_pub_key_set()* on the provisioner side.

Authentication After the initial exchange, the provisioner selects an Out of Band (OOB) Authentication method. This allows the user to confirm that the device the provisioner connected to is actually the device they intended, and not a malicious third party.

The Provisioning API supports the following authentication methods for the provisionee:

- **Static OOB:** An authentication value is assigned to the device in production, which the provisioner can query in some application specific way.
- **Input OOB:** The user inputs the authentication value. The available input actions are listed in *bt_mesh_input_action_t*.
- **Output OOB:** Show the user the authentication value. The available output actions are listed in *bt_mesh_output_action_t*.

The application must provide callbacks for the supported authentication methods in *bt_mesh_prov*, as well as enabling the supported actions in *bt_mesh_prov.output_actions* and *bt_mesh_prov.input_actions*.

When an Output OOB action is selected, the authentication value should be presented to the user when the output callback is called, and remain until the *bt_mesh_prov.input_complete* or *bt_mesh_prov.complete* callback is called. If the action is `blink`, `beep` or `vibrate`, the sequence should be repeated after a delay of three seconds or more.

When an Input OOB action is selected, the user should be prompted when the application receives the *bt_mesh_prov.input* callback. The user response should be fed back to the Provisioning API through *bt_mesh_input_string()* or *bt_mesh_input_number()*. If no user response is recorded within 60 seconds, the Provisioning process is aborted.

If Provisionee wants to mandate OOB authentication, it is mandatory to use the BT_MESH_ECDH_P256_HMAC_SHA256_AES_CCM algorithm.

Data transfer After the device has been successfully authenticated, the provisioner transfers the Provisioning data:

- Unicast address
- A network key
- IV index
- Network flags
  - Key refresh
  - IV update

Additionally, a device key is generated for the node. All this data is stored by the mesh stack, and the provisioning *bt_mesh_prov.complete* callback gets called.

Provisioning security Depending on the choice of public key exchange mechanism and authentication method, the provisioning process can be secure or insecure.

On May 24th 2021, ANSSI disclosed a set of vulnerabilities in the Bluetooth Mesh provisioning protocol that showcased how the low entropy provided by the Blink, Vibrate, Push, Twist and Input/Output numeric OOB methods could be exploited in impersonation and MITM attacks. In response, the Bluetooth SIG has reclassified these OOB methods as insecure in the Bluetooth Mesh Profile Specification v1.0.1 erratum 16350, as AuthValue may be brute forced in real time. To ensure secure provisioning, applications should use a static OOB value and OOB public key transfer.

API reference

group `bt_mesh_prov`

Provisioning.

Enums

Available authentication algorithms.

Values:

enumerator `BT_MESH_PROV_AUTH_CMAC_AES128_AES_CCM`

enumerator `BT_MESH_PROV_AUTH_HMAC_SHA256_AES_CCM`

OOB Type field values.

Values:

enumerator `BT_MESH_STATIC_OOB_AVAILABLE` = *BIT*(0)

Static OOB information available.

enumerator `BT_MESH_OOB_AUTH_REQUIRED` = *BIT*(1)

OOB authentication required.

enum `bt_mesh_output_action_t`

Available Provisioning output authentication actions.

Values:

enumerator `BT_MESH_NO_OUTPUT` = 0

enumerator `BT_MESH_BLINK` = *BIT*(0)

Blink.

enumerator `BT_MESH_BEEP` = *BIT*(1)

Beep.

enumerator `BT_MESH_VIBRATE` = *BIT*(2)

Vibrate.

enumerator `BT_MESH_DISPLAY_NUMBER` = *BIT*(3)

Output numeric.

enumerator `BT_MESH_DISPLAY_STRING` = *BIT*(4)

Output alphanumeric.

enum **bt_mesh_input_action_t**

Available Provisioning input authentication actions.

Values:

enumerator **BT_MESH_NO_INPUT** = 0

enumerator **BT_MESH_PUSH** = *BIT*(0)

Push.

enumerator **BT_MESH_TWIST** = *BIT*(1)

Twist.

enumerator **BT_MESH_ENTER_NUMBER** = *BIT*(2)

Input number.

enumerator **BT_MESH_ENTER_STRING** = *BIT*(3)

Input alphanumeric.

enum **bt_mesh_prov_bearer_t**

Available Provisioning bearers.

Values:

enumerator **BT_MESH_PROV_ADV** = *BIT*(0)

PB-ADV bearer.

enumerator **BT_MESH_PROV_GATT** = *BIT*(1)

PB-GATT bearer.

enumerator **BT_MESH_PROV_REMOTE** = *BIT*(2)

PB-Remote bearer.

enum **bt_mesh_prov_oob_info_t**

Out of Band information location.

Values:

enumerator **BT_MESH_PROV_OOB_OTHER** = *BIT*(0)

Other.

enumerator **BT_MESH_PROV_OOB_URI** = *BIT*(1)

Electronic / URI.

enumerator **BT_MESH_PROV_OOB_2D_CODE** = *BIT*(2)

2D machine-readable code

enumerator **BT_MESH_PROV_OOB_BAR_CODE** = *BIT*(3)

Bar Code.

enumerator BT_MESH_PROV_OOB_NFC = *BIT*(4)
Near Field Communication (NFC)

enumerator BT_MESH_PROV_OOB_NUMBER = *BIT*(5)
Number.

enumerator BT_MESH_PROV_OOB_STRING = *BIT*(6)
String.

enumerator BT_MESH_PROV_OOB_CERTIFICATE = *BIT*(7)
Support for certificate-based provisioning.

enumerator BT_MESH_PROV_OOB_RECORDS = *BIT*(8)
Support for provisioning records.

enumerator BT_MESH_PROV_OOB_ON_BOX = *BIT*(11)
On box.

enumerator BT_MESH_PROV_OOB_IN_BOX = *BIT*(12)
Inside box.

enumerator BT_MESH_PROV_OOB_ON_PAPER = *BIT*(13)
On piece of paper.

enumerator BT_MESH_PROV_OOB_IN_MANUAL = *BIT*(14)
Inside manual.

enumerator BT_MESH_PROV_OOB_ON_DEV = *BIT*(15)
On device.

Functions

int **bt_mesh_input_string**(const char *str)
Provide provisioning input OOB string.

This is intended to be called after the *bt_mesh_prov* input callback has been called with BT_MESH_ENTER_STRING as the action.

Parameters

- **str** – String.

Returns
Zero on success or (negative) error code otherwise.

int **bt_mesh_input_number**(uint32_t num)
Provide provisioning input OOB number.

This is intended to be called after the *bt_mesh_prov* input callback has been called with BT_MESH_ENTER_NUMBER as the action.

Parameters

- **num** – Number.

Returns
Zero on success or (negative) error code otherwise.

int **bt_mesh_prov_remote_pub_key_set**(const uint8_t public_key[64])

Provide Device public key.

Parameters

- **public_key** – Device public key in big-endian.

Returns
Zero on success or (negative) error code otherwise.

int **bt_mesh_auth_method_set_input**(*bt_mesh_input_action_t* action, uint8_t size)

Use Input OOB authentication.

Provisioner only.

Instruct the unprovisioned device to use the specified Input OOB authentication action. When using *BT_MESH_PUSH*, *BT_MESH_TWIST* or *BT_MESH_ENTER_NUMBER*, the *bt_mesh_prov::output_number* callback is called with a random number that has to be entered on the unprovisioned device.

When using *BT_MESH_ENTER_STRING*, the *bt_mesh_prov::output_string* callback is called with a random string that has to be entered on the unprovisioned device.

Parameters

- **action** – Authentication action used by the unprovisioned device.
- **size** – Authentication size.

Returns
Zero on success or (negative) error code otherwise.

int **bt_mesh_auth_method_set_output**(*bt_mesh_output_action_t* action, uint8_t size)

Use Output OOB authentication.

Provisioner only.

Instruct the unprovisioned device to use the specified Output OOB authentication action. The *bt_mesh_prov::input* callback will be called.

When using *BT_MESH_BLINK*, *BT_MESH_BEEP*, *BT_MESH_VIBRATE* or *BT_MESH_DISPLAY_NUMBER*, and the application has to call *bt_mesh_input_number* with the random number indicated by the unprovisioned device.

When using *BT_MESH_DISPLAY_STRING*, the application has to call *bt_mesh_input_string* with the random string displayed by the unprovisioned device.

Parameters

- **action** – Authentication action used by the unprovisioned device.
- **size** – Authentication size.

Returns
Zero on success or (negative) error code otherwise.

int **bt_mesh_auth_method_set_static**(const uint8_t *static_val, uint8_t size)

Use static OOB authentication.

Provisioner only.

Instruct the unprovisioned device to use static OOB authentication, and use the given static authentication value when provisioning.

Parameters

- `static_val` – Static OOB value.
- `size` – Static OOB value size.

Returns
Zero on success or (negative) error code otherwise.

int `bt_mesh_auth_method_set_none`(void)

Don't use OOB authentication.

Provisioner only.

Don't use any authentication when provisioning new devices. This is the default behavior.

> **Warning**
>
> Not using any authentication exposes the mesh network to impersonation attacks, where attackers can pretend to be the unprovisioned device to gain access to the network. Authentication is strongly encouraged.

Returns
Zero on success or (negative) error code otherwise.

int `bt_mesh_prov_enable`(*bt_mesh_prov_bearer_t* bearers)

Enable specific provisioning bearers.

Enable one or more provisioning bearers.

Parameters

- `bearers` – Bit-wise or of provisioning bearers.

Returns
Zero on success or (negative) error code otherwise.

int `bt_mesh_prov_disable`(*bt_mesh_prov_bearer_t* bearers)

Disable specific provisioning bearers.

Disable one or more provisioning bearers.

Parameters

- `bearers` – Bit-wise or of provisioning bearers.

Returns
Zero on success or (negative) error code otherwise.

int `bt_mesh_provision`(const uint8_t net_key[16], uint16_t net_idx, uint8_t flags, uint32_t iv_index, uint16_t addr, const uint8_t dev_key[16])

Provision the local Mesh Node.

This API should normally not be used directly by the application. The only exception is for testing purposes where manual provisioning is desired without an actual external provisioner.

Parameters

- `net_key` – Network Key
- `net_idx` – Network Key Index
- `flags` – Provisioning Flags
- `iv_index` – IV Index
- `addr` – Primary element address

- **dev_key** – Device Key

Returns
Zero on success or (negative) error code otherwise.

int **bt_mesh_provision_adv**(const uint8_t uuid[16], uint16_t net_idx, uint16_t addr, uint8_t attention_duration)

Provision a Mesh Node using PB-ADV.

Parameters

- **uuid** – UUID
- **net_idx** – Network Key Index
- **addr** – Address to assign to remote device. If addr is 0, the lowest available address will be chosen.
- **attention_duration** – The attention duration to be send to remote device

Returns
Zero on success or (negative) error code otherwise.

int **bt_mesh_provision_gatt**(const uint8_t uuid[16], uint16_t net_idx, uint16_t addr, uint8_t attention_duration)

Provision a Mesh Node using PB-GATT.

Parameters

- **uuid** – UUID
- **net_idx** – Network Key Index
- **addr** – Address to assign to remote device. If addr is 0, the lowest available address will be chosen.
- **attention_duration** – The attention duration to be send to remote device

Returns
Zero on success or (negative) error code otherwise.

int **bt_mesh_provision_remote**(struct *bt_mesh_rpr_cli* *cli, const struct bt_mesh_rpr_node *srv, const uint8_t uuid[16], uint16_t net_idx, uint16_t addr)

Provision a Mesh Node using PB-Remote.

Parameters

- **cli** – Remote Provisioning Client Model to provision with.
- **srv** – Remote Provisioning Server that should be used to tunnel the provisioning.
- **uuid** – UUID of the unprovisioned node
- **net_idx** – Network Key Index to give to the unprovisioned node.
- **addr** – Address to assign to remote device. If addr is 0, the lowest available address will be chosen.

Returns
Zero on success or (negative) error code otherwise.

int **bt_mesh_reprovision_remote**(struct *bt_mesh_rpr_cli* *cli, struct bt_mesh_rpr_node *srv, uint16_t addr, bool comp_change)

Reprovision a Mesh Node using PB-Remote.

Reprovisioning can be used to change the device key, unicast address and composition data of another device. The reprovisioning procedure uses the same protocol as normal provisioning, with the same level of security.

There are three tiers of reprovisioning:

a. Refreshing the device key

b. Refreshing the device key and node address. Composition data may change, including the number of elements.

c. Refreshing the device key and composition data, in case the composition data of the target node changed due to a firmware update or a similar procedure.

The target node indicates that its composition data changed by instantiating its composition data page 128. If the number of elements have changed, it may be necessary to move the unicast address of the target node as well, to avoid overlapping addresses.

Note

Changing the unicast addresses of the target node requires changes to all nodes that publish directly to any of the target node's models.

Parameters

- `cli` – Remote Provisioning Client Model to provision on
- `srv` – Remote Provisioning Server to reprovision
- `addr` – Address to assign to remote device. If addr is 0, the lowest available address will be chosen.
- `comp_change` – The target node has indicated that its composition data has changed. Note that the target node will reject the update if this isn't true.

Returns

Zero on success or (negative) error code otherwise.

bool `bt_mesh_is_provisioned`(void)

Check if the local node has been provisioned.

This API can be used to check if the local node has been provisioned or not. It can e.g. be helpful to determine if there was a stored network in flash, i.e. if the network was restored after calling *settings_load()*.

Returns

True if the node is provisioned. False otherwise.

struct `bt_mesh_dev_capabilities`

#include <main.h> Device Capabilities.

Public Members

uint8_t `elem_count`

Number of elements supported by the device.

uint16_t `algorithms`

Supported algorithms and other capabilities.

uint8_t **pub_key_type**

Supported public key types.

uint8_t **oob_type**

Supported OOB Types.

bt_mesh_output_action_t **output_actions**

Supported Output OOB Actions.

bt_mesh_input_action_t **input_actions**

Supported Input OOB Actions.

uint8_t **output_size**

Maximum size of Output OOB supported.

uint8_t **input_size**

Maximum size in octets of Input OOB supported.

struct **bt_mesh_prov**

#include <main.h> Provisioning properties & capabilities.

Public Members

const uint8_t ***uuid**

The UUID that's used when advertising as unprovisioned.

const char ***uri**

Optional URI.

This will be advertised separately from the unprovisioned beacon, however the unprovisioned beacon will contain a hash of it so the two can be associated by the provisioner.

bt_mesh_prov_oob_info_t **oob_info**

Out of Band information field.

const uint8_t ***public_key_be**

Pointer to Public Key in big-endian for OOB public key type support.

Remember to enable `CONFIG_BT_MESH_PROV_OOB_PUBLIC_KEY` when initializing this parameter.

Must be used together with *bt_mesh_prov::private_key_be*.

const uint8_t ***private_key_be**

Pointer to Private Key in big-endian for OOB public key type support.

Remember to enable `CONFIG_BT_MESH_PROV_OOB_PUBLIC_KEY` when initializing this parameter.

Must be used together with *bt_mesh_prov::public_key_be*.

const uint8_t *static_val

Static OOB value.

uint8_t static_val_len

Static OOB value length.

uint8_t output_size

Maximum size of Output OOB supported.

uint16_t output_actions

Supported Output OOB Actions.

uint8_t input_size

Maximum size of Input OOB supported.

uint16_t input_actions

Supported Input OOB Actions.

void (*capabilities)(const struct *bt_mesh_dev_capabilities* *cap)

Provisioning Capabilities.

This callback notifies the application that the provisioning capabilities of the unprovisioned device has been received.

The application can consequently call bt_mesh_auth_method_set_<*> to select suitable provisioning oob authentication method.

When this callback returns, the provisioner will start authentication with the chosen method.

Param cap
capabilities supported by device.

int (*output_number)(*bt_mesh_output_action_t* act, uint32_t num)

Output of a number is requested.

This callback notifies the application that it should output the given number using the given action.

Param act
Action for outputting the number.

Param num
Number to be outputted.

Return
Zero on success or negative error code otherwise

int (*output_string)(const char *str)

Output of a string is requested.

This callback notifies the application that it should display the given string to the user.

Param str
String to be displayed.

Return
Zero on success or negative error code otherwise

int (*input)(*bt_mesh_input_action_t* act, uint8_t size)

Input is requested.

This callback notifies the application that it should request input from the user using the given action. The requested input will either be a string or a number, and the application needs to consequently call the *bt_mesh_input_string()* or *bt_mesh_input_number()* functions once the data has been acquired from the user.

Param act
Action for inputting data.

Param num
Maximum size of the inputted data.

Return
Zero on success or negative error code otherwise

void (*input_complete)(void)

The other device finished their OOB input.

This callback notifies the application that it should stop displaying its output OOB value, as the other party finished their OOB input.

void (*unprovisioned_beacon)(uint8_t uuid[16], *bt_mesh_prov_oob_info_t* oob_info, uint32_t *uri_hash)

Unprovisioned beacon has been received.

This callback notifies the application that an unprovisioned beacon has been received.

Param uuid
UUID

Param oob_info
OOB Information

Param uri_hash
Pointer to URI Hash value. NULL if no hash was present in the beacon.

void (*unprovisioned_beacon_gatt)(uint8_t uuid[16], *bt_mesh_prov_oob_info_t* oob_info)

PB-GATT Unprovisioned Advertising has been received.

This callback notifies the application that an PB-GATT unprovisioned Advertising has been received.

Param uuid
UUID

Param oob_info
OOB Information

void (*link_open)(*bt_mesh_prov_bearer_t* bearer)

Provisioning link has been opened.

This callback notifies the application that a provisioning link has been opened on the given provisioning bearer.

Param bearer
Provisioning bearer.

void (*link_close)(*bt_mesh_prov_bearer_t* bearer)

Provisioning link has been closed.

This callback notifies the application that a provisioning link has been closed on the given provisioning bearer.

Param bearer
Provisioning bearer.

void (*`complete`)(uint16_t net_idx, uint16_t addr)

Provisioning is complete.

This callback notifies the application that provisioning has been successfully completed, and that the local node has been assigned the specified NetKeyIndex and primary element address.

Param net_idx
NetKeyIndex given during provisioning.
Param addr
Primary element address.

void (*`reprovisioned`)(uint16_t addr)

Local node has been reprovisioned.

This callback notifies the application that reprovisioning has been successfully completed.

Param addr
New primary element address.

void (*`node_added`)(uint16_t net_idx, uint8_t uuid[16], uint16_t addr, uint8_t num_elem)

A new node has been added to the provisioning database.

This callback notifies the application that provisioning has been successfully completed, and that a node has been assigned the specified NetKeyIndex and primary element address.

Param net_idx
NetKeyIndex given during provisioning.
Param uuid
UUID of the added node
Param addr
Primary element address.
Param num_elem
Number of elements that this node has.

void (*`reset`)(void)

Node has been reset.

This callback notifies the application that the local node has been reset and needs to be provisioned again. The node will not automatically advertise as unprovisioned, rather the *bt_mesh_prov_enable()* API needs to be called to enable unprovisioned advertising on one or more provisioning bearers.

Proxy The Proxy feature allows legacy devices like phones to access the Bluetooth Mesh network through GATT. The Proxy feature is only compiled in if the `CONFIG_BT_MESH_GATT_PROXY` option is set. The Proxy feature state is controlled by the *Configuration Server*, and the initial value can be set with `bt_mesh_cfg_srv.gatt_proxy`.

Nodes with the Proxy feature enabled can advertise with Network Identity and Node Identity, which is controlled by the *Configuration Client*.

The GATT Proxy state indicates if the Proxy feature is supported.

Private Proxy A node supporting the Proxy feature and the *Private Beacon Server* model can advertise with Private Network Identity and Private Node Identity types, which is controlled by the *Private Beacon Client*. By advertising with this set of identification types, the node allows the legacy device to connect to the network over GATT while maintaining the privacy of the network.

The Private GATT Proxy state indicates whether the Private Proxy functionality is supported.

Proxy Solicitation In the case where both GATT Proxy and Private GATT Proxy states are disabled on a node, a legacy device cannot connect to it. A node supporting the *On-Demand Private Proxy Server* may however be solicited to advertise connectable advertising events without enabling the Private GATT Proxy state. To solicit the node, the legacy device can send a Solicitation PDU by calling the `bt_mesh_proxy_solicit()` function. To enable this feature, the device must to be compiled with the `CONFIG_BT_MESH_PROXY_SOLICITATION` option set.

Solicitation PDUs are non-mesh, non-connectable, undirected advertising messages containing Proxy Solicitation UUID, encrypted with the network key of the subnet that the legacy device wants to connect to. The PDU contains the source address of the legacy device and a sequence number. The sequence number is maintained by the legacy device and is incremented for every new Solicitation PDU sent.

Each node supporting the Solicitation PDU reception holds its own Solicitation Replay Protection List (SRPL). The SRPL protects the solicitation mechanism from replay attacks by storing solicitation sequence number (SSEQ) and solicitation source (SSRC) pairs of valid Solicitation PDUs processed by the node. The delay between updating the SRPL and storing the change to the persistent storage is defined by `CONFIG_BT_MESH_RPL_STORE_TIMEOUT`.

The Solicitation PDU RPL Configuration models, *Solicitation PDU RPL Configuration Client* and *Solicitation PDU RPL Configuration Server*, provide the functionality of saving and clearing SRPL entries. A node that supports the Solicitation PDU RPL Configuration Client model can clear a section of the SRPL on the target by calling the `bt_mesh_sol_pdu_rpl_clear()` function. Communication between the Solicitation PDU RPL Configuration Client and Server is encrypted using the application key, therefore, the Solicitation PDU RPL Configuration Client can be instantiated on any device in the network.

When the node receives the Solicitation PDU and successfully authenticates it, it will start advertising connectable advertisements with the Private Network Identity type. The duration of the advertisement can be configured by the On-Demand Private Proxy Client model.

API reference

group `bt_mesh_proxy`

Proxy.

Defines

`BT_MESH_PROXY_CB_DEFINE`(_name)

Register a callback structure for Proxy events.

Registers a structure with callback functions that gets called on various Proxy events.

Parameters

- `_name` – Name of callback structure.

Functions

int **bt_mesh_proxy_identity_enable**(void)

Enable advertising with Node Identity.

This API requires that GATT Proxy support has been enabled. Once called each subnet will start advertising using Node Identity for the next 60 seconds.

Returns

0 on success, or (negative) error code on failure.

int **bt_mesh_proxy_private_identity_enable**(void)

Enable advertising with Private Node Identity.

This API requires that GATT Proxy support has been enabled. Once called each subnet will start advertising using Private Node Identity for the next 60 seconds.

Returns

0 on success, or (negative) error code on failure.

int **bt_mesh_proxy_connect**(uint16_t net_idx)

Allow Proxy Client to auto connect to a network.

This API allows a proxy client to auto-connect a given network.

Parameters

- **net_idx** – Network Key Index

Returns

0 on success, or (negative) error code on failure.

int **bt_mesh_proxy_disconnect**(uint16_t net_idx)

Disallow Proxy Client to auto connect to a network.

This API disallows a proxy client to connect a given network.

Parameters

- **net_idx** – Network Key Index

Returns

0 on success, or (negative) error code on failure.

int **bt_mesh_proxy_solicit**(uint16_t net_idx)

Schedule advertising of Solicitation PDUs.

Once called, the device will schedule advertising Solicitation PDUs for the amount of time defined by `adv_int` * (`CONFIG_BT_MESH_SOL_ADV_XMIT` + 1), where `adv_int` is 20ms for Bluetooth v5.0 or higher, or 100ms otherwise.

If the number of advertised Solicitation PDUs reached 0xFFFFFF, the advertisements will no longer be started until the node is reprovisioned.

Parameters

- **net_idx** – Network Key Index

Returns

0 on success, or (negative) error code on failure.

struct **bt_mesh_proxy_cb**

#include <*proxy.h*> Callbacks for the Proxy feature.

Should be instantiated with *BT_MESH_PROXY_CB_DEFINE*.

Public Members

void (*identity_enabled)(uint16_t net_idx)

Started sending Node Identity beacons on the given subnet.

Param net_idx
Network index the Node Identity beacons are running on.

void (*identity_disabled)(uint16_t net_idx)

Stopped sending Node Identity beacons on the given subnet.

Param net_idx
Network index the Node Identity beacons were running on.

Heartbeat The Heartbeat feature provides functionality for monitoring Bluetooth Mesh nodes and determining the distance between nodes.

The Heartbeat feature is configured through the *Configuration Server* model.

Heartbeat messages Heartbeat messages are sent as transport control packets through the network, and are only encrypted with a network key. Heartbeat messages contain the original Time To Live (TTL) value used to send the message and a bitfield of the active features on the node. Through this, a receiving node can determine how many relays the message had to go through to arrive at the receiver, and what features the node supports.

Available Heartbeat feature flags:

- *BT_MESH_FEAT_RELAY*
- *BT_MESH_FEAT_PROXY*
- *BT_MESH_FEAT_FRIEND*
- *BT_MESH_FEAT_LOW_POWER*

Heartbeat publication Heartbeat publication is controlled through the Configuration models, and can be triggered in two ways:

Periodic publication
The node publishes a new Heartbeat message at regular intervals. The publication can be configured to stop after a certain number of messages, or continue indefinitely.

Triggered publication
The node publishes a new Heartbeat message every time a feature changes. The set of features that can trigger the publication is configurable.

The two publication types can be combined.

Heartbeat subscription A node can be configured to subscribe to Heartbeat messages from one node at the time. To receive a Heartbeat message, both the source and destination must match the configured subscription parameters.

Heartbeat subscription is always time limited, and throughout the subscription period, the node keeps track of the number of received Heartbeats as well as the minimum and maximum received hop count.

All Heartbeats received with the configured subscription parameters are passed to the `bt_mesh_hb_cb::recv` event handler.

When the Heartbeat subscription period ends, the `bt_mesh_hb_cb::sub_end` callback gets called.

API reference

group `bt_mesh_heartbeat`

Heartbeat.

Defines

`BT_MESH_HB_CB_DEFINE`(_name)

Register a callback structure for Heartbeat events.

Registers a callback structure that will be called whenever Heartbeat events occur

Parameters

- `_name` – Name of callback structure.

Functions

void `bt_mesh_hb_pub_get`(struct *bt_mesh_hb_pub* *get)

Get the current Heartbeat publication parameters.

Parameters

- `get` – Heartbeat publication parameters return buffer.

void `bt_mesh_hb_sub_get`(struct *bt_mesh_hb_sub* *get)

Get the current Heartbeat subscription parameters.

Parameters

- `get` – Heartbeat subscription parameters return buffer.

struct `bt_mesh_hb_pub`

#include <*heartbeat.h*> Heartbeat Publication parameters.

Public Members

uint16_t `dst`

Destination address.

uint16_t `count`

Remaining publish count.

uint8_t `ttl`

Time To Live value.

uint16_t `feat`

Bitmap of features that trigger a Heartbeat publication if they change.

Legal values are *BT_MESH_FEAT_RELAY*, *BT_MESH_FEAT_PROXY*, *BT_MESH_FEAT_FRIEND* and *BT_MESH_FEAT_LOW_POWER*.

uint16_t `net_idx`

Network index used for publishing.

uint32_t **period**

Publication period in seconds.

struct **bt_mesh_hb_sub**

#include <*heartbeat.h*> Heartbeat Subscription parameters.

Public Members

uint32_t **period**

Subscription period in seconds.

uint32_t **remaining**

Remaining subscription time in seconds.

uint16_t **src**

Source address to receive Heartbeats from.

uint16_t **dst**

Destination address to received Heartbeats on.

uint16_t **count**

The number of received Heartbeat messages so far.

uint8_t **min_hops**

Minimum hops in received messages, ie the shortest registered path from the publishing node to the subscribing node.

A Heartbeat received from an immediate neighbor has hop count = 1.

uint8_t **max_hops**

Maximum hops in received messages, ie the longest registered path from the publishing node to the subscribing node.

A Heartbeat received from an immediate neighbor has hop count = 1.

struct **bt_mesh_hb_cb**

#include <*heartbeat.h*> Heartbeat callback structure.

Public Members

void (***recv**)(const struct *bt_mesh_hb_sub* *sub, uint8_t hops, uint16_t feat)

Receive callback for heartbeats.

Gets called on every received Heartbeat that matches the current Heartbeat subscription parameters.

Param sub
Current Heartbeat subscription parameters.

Param hops
The number of hops the Heartbeat was received with.

Param feat
The feature set of the publishing node. The value is a bitmap of *BT_MESH_FEAT_RELAY*, *BT_MESH_FEAT_PROXY*, *BT_MESH_FEAT_FRIEND* and *BT_MESH_FEAT_LOW_POWER*.

void (*`sub_end`)(const struct *bt_mesh_hb_sub* *sub)

Subscription end callback for heartbeats.

Gets called when the subscription period ends, providing a summary of the received heartbeat messages.

Param sub
Current Heartbeat subscription parameters.

void (*`pub_sent`)(const struct *bt_mesh_hb_pub* *pub)

Publication sent callback for heartbeats.

Gets called when the heartbeat is successfully published.

Param pub
Current Heartbeat publication parameters.

Runtime Configuration The runtime configuration API allows applications to change their runtime configuration directly, without going through the Configuration models.

Bluetooth Mesh nodes should generally be configured by a central network configurator device with a *Configuration Client* model. Each mesh node instantiates a *Configuration Server* model that the Configuration Client can communicate with to change the node configuration. In some cases, the mesh node can't rely on the Configuration Client to detect or determine local constraints, such as low battery power or changes in topology. For these scenarios, this API can be used to change the configuration locally.

Note

Runtime configuration changes before the node is provisioned will not be stored in the *persistent storage*.

API reference

group `bt_mesh_cfg`

Runtime Configuration.

Defines

`BT_MESH_KR_NORMAL`

`BT_MESH_KR_PHASE_1`

`BT_MESH_KR_PHASE_2`

`BT_MESH_KR_PHASE_3`

`BT_MESH_RELAY_DISABLED`

BT_MESH_RELAY_ENABLED

BT_MESH_RELAY_NOT_SUPPORTED

BT_MESH_BEACON_DISABLED

BT_MESH_BEACON_ENABLED

BT_MESH_PRIV_BEACON_DISABLED

BT_MESH_PRIV_BEACON_ENABLED

BT_MESH_GATT_PROXY_DISABLED

BT_MESH_GATT_PROXY_ENABLED

BT_MESH_GATT_PROXY_NOT_SUPPORTED

BT_MESH_PRIV_GATT_PROXY_DISABLED

BT_MESH_PRIV_GATT_PROXY_ENABLED

BT_MESH_PRIV_GATT_PROXY_NOT_SUPPORTED

BT_MESH_FRIEND_DISABLED

BT_MESH_FRIEND_ENABLED

BT_MESH_FRIEND_NOT_SUPPORTED

BT_MESH_NODE_IDENTITY_STOPPED

BT_MESH_NODE_IDENTITY_RUNNING

BT_MESH_NODE_IDENTITY_NOT_SUPPORTED

Enums

enum **bt_mesh_feat_state**

Bluetooth Mesh feature states.

Values:

enumerator **BT_MESH_FEATURE_DISABLED**

Feature is supported, but disabled.

enumerator BT_MESH_FEATURE_ENABLED

Feature is supported and enabled.

enumerator BT_MESH_FEATURE_NOT_SUPPORTED

Feature is not supported, and cannot be enabled.

Functions

void **bt_mesh_beacon_set**(bool beacon)

Enable or disable sending of the Secure Network Beacon.

Parameters

- **beacon** – New Secure Network Beacon state.

bool **bt_mesh_beacon_enabled**(void)

Get the current Secure Network Beacon state.

Returns

Whether the Secure Network Beacon feature is enabled.

int **bt_mesh_priv_beacon_set**(enum *bt_mesh_feat_state* priv_beacon)

Enable or disable sending of the Mesh Private beacon.

Support for the Private beacon state must be enabled with CONFIG_BT_MESH_PRIV_BEACONS.

Parameters

- **priv_beacon** – New Mesh Private beacon state. Must be one of *BT_MESH_FEATURE_ENABLED* and *BT_MESH_FEATURE_DISABLED*.

Return values

- **0** – Successfully changed the Mesh Private beacon feature state.
- **-ENOTSUP** – The Mesh Private beacon feature is not supported.
- **-EINVAL** – Invalid parameter.
- **-EALREADY** – Already in the given state.

enum *bt_mesh_feat_state* **bt_mesh_priv_beacon_get**(void)

Get the current Mesh Private beacon state.

Returns

The Mesh Private beacon feature state.

void **bt_mesh_priv_beacon_update_interval_set**(uint8_t interval)

Set the current Mesh Private beacon update interval.

The Mesh Private beacon's randomization value is updated regularly to maintain the node's privacy. The update interval controls how often the beacon is updated, in 10 second increments.

Parameters

- **interval** – Private beacon update interval in 10 second steps, or 0 to update on every beacon transmission.

uint8_t **bt_mesh_priv_beacon_update_interval_get**(void)

Get the current Mesh Private beacon update interval.

The Mesh Private beacon's randomization value is updated regularly to maintain the node's privacy. The update interval controls how often the beacon is updated, in 10 second increments.

Returns

The Private beacon update interval in 10 second steps, or 0 if the beacon is updated every time it's transmitted.

int **bt_mesh_default_ttl_set**(uint8_t default_ttl)

Set the default TTL value.

The default TTL value is used when no explicit TTL value is set. Models will use the default TTL value when *bt_mesh_msg_ctx::send_ttl* is *BT_MESH_TTL_DEFAULT*.

Parameters

- **default_ttl** – The new default TTL value. Valid values are 0x00 and 0x02 to *BT_MESH_TTL_MAX*.

Return values

- **0** – Successfully set the default TTL value.
- **-EINVAL** – Invalid TTL value.

uint8_t **bt_mesh_default_ttl_get**(void)

Get the current default TTL value.

Returns

The current default TTL value.

int **bt_mesh_od_priv_proxy_get**(void)

Get the current Mesh On-Demand Private Proxy state.

Return values

- **0** – or positive value represents On-Demand Private Proxy feature state
- **-ENOTSUP** – The On-Demand Private Proxy feature is not supported.

int **bt_mesh_od_priv_proxy_set**(uint8_t on_demand_proxy)

Set state of Mesh On-Demand Private Proxy.

Support for the On-Demand Private Proxy state must be enabled with BT_MESH_OD_PRIV_PROXY_SRV.

Parameters

- **on_demand_proxy** – New Mesh On-Demand Private Proxy state. Value of 0x00 means that advertising with Private Network Identity cannot be enabled on demand. Values in range 0x01 - 0xFF set interval of this advertising after valid Solicitation PDU is received or client disconnects.

Return values

- **0** – Successfully changed the Mesh On-Demand Private Proxy feature state.
- **-ENOTSUP** – The On-Demand Private Proxy feature is not supported.
- **-EINVAL** – Invalid parameter.
- **-EALREADY** – Already in the given state.

void **bt_mesh_net_transmit_set**(uint8_t xmit)

Set the Network Transmit parameters.

The Network Transmit parameters determine the parameters local messages are transmitted with.

See also

BT_MESH_TRANSMIT

Parameters

- **xmit** – New Network Transmit parameters. Use *BT_MESH_TRANSMIT* for encoding.

uint8_t **bt_mesh_net_transmit_get**(void)

Get the current Network Transmit parameters.

The *BT_MESH_TRANSMIT_COUNT* and *BT_MESH_TRANSMIT_INT* macros can be used to decode the Network Transmit parameters.

Returns

The current Network Transmit parameters.

int **bt_mesh_relay_set**(enum *bt_mesh_feat_state* relay, uint8_t xmit)

Configure the Relay feature.

Enable or disable the Relay feature, and configure the parameters to transmit relayed messages with.

Support for the Relay feature must be enabled through the CONFIG_BT_MESH_RELAY configuration option.

See also

BT_MESH_TRANSMIT

Parameters

- **relay** – New Relay feature state. Must be one of *BT_MESH_FEATURE_ENABLED* and *BT_MESH_FEATURE_DISABLED*.
- **xmit** – New Relay retransmit parameters. Use *BT_MESH_TRANSMIT* for encoding.

Return values

- **0** – Successfully changed the Relay configuration.
- **-ENOTSUP** – The Relay feature is not supported.
- **-EINVAL** – Invalid parameter.
- **-EALREADY** – Already using the given parameters.

enum *bt_mesh_feat_state* **bt_mesh_relay_get**(void)

Get the current Relay feature state.

Returns

The Relay feature state.

uint8_t **bt_mesh_relay_retransmit_get**(void)

Get the current Relay Retransmit parameters.

The *BT_MESH_TRANSMIT_COUNT* and *BT_MESH_TRANSMIT_INT* macros can be used to decode the Relay Retransmit parameters.

Returns

The current Relay Retransmit parameters, or 0 if relay is not supported.

int **bt_mesh_gatt_proxy_set**(enum *bt_mesh_feat_state* gatt_proxy)

Enable or disable the GATT Proxy feature.

Support for the GATT Proxy feature must be enabled through the CONFIG_BT_MESH_GATT_PROXY configuration option.

Note

The GATT Proxy feature only controls a Proxy node's ability to relay messages to the mesh network. A node that supports GATT Proxy will still advertise Connectable Proxy beacons, even if the feature is disabled. The Proxy feature can only be fully disabled through compile time configuration.

Parameters

- **gatt_proxy** – New GATT Proxy state. Must be one of *BT_MESH_FEATURE_ENABLED* and *BT_MESH_FEATURE_DISABLED*.

Return values

- **0** – Successfully changed the GATT Proxy feature state.
- **-ENOTSUP** – The GATT Proxy feature is not supported.
- **-EINVAL** – Invalid parameter.
- **-EALREADY** – Already in the given state.

enum *bt_mesh_feat_state* **bt_mesh_gatt_proxy_get**(void)

Get the current GATT Proxy state.

Returns

The GATT Proxy feature state.

int **bt_mesh_priv_gatt_proxy_set**(enum *bt_mesh_feat_state* priv_gatt_proxy)

Enable or disable the Private GATT Proxy feature.

Support for the Private GATT Proxy feature must be enabled through the CONFIG_BT_MESH_PRIV_BEACONS and CONFIG_BT_MESH_GATT_PROXY configuration options.

Parameters

- **priv_gatt_proxy** – New Private GATT Proxy state. Must be one of *BT_MESH_FEATURE_ENABLED* and *BT_MESH_FEATURE_DISABLED*.

Return values

- **0** – Successfully changed the Private GATT Proxy feature state.
- **-ENOTSUP** – The Private GATT Proxy feature is not supported.
- **-EINVAL** – Invalid parameter.
- **-EALREADY** – Already in the given state.

enum *bt_mesh_feat_state* **bt_mesh_priv_gatt_proxy_get**(void)

Get the current Private GATT Proxy state.

Returns

The Private GATT Proxy feature state.

int **bt_mesh_friend_set**(enum *bt_mesh_feat_state* friendship)

Enable or disable the Friend feature.

Any active friendships will be terminated immediately if the Friend feature is disabled.

Support for the Friend feature must be enabled through the CONFIG_BT_MESH_FRIEND configuration option.

Parameters

- **friendship** – New Friend feature state. Must be one of *BT_MESH_FEATURE_ENABLED* and *BT_MESH_FEATURE_DISABLED*.

Return values

- **0** – Successfully changed the Friend feature state.
- **-ENOTSUP** – The Friend feature is not supported.
- **-EINVAL** – Invalid parameter.
- **-EALREADY** – Already in the given state.

enum *bt_mesh_feat_state* **bt_mesh_friend_get**(void)

Get the current Friend state.

Returns

The Friend feature state.

Frame statistic The frame statistic API allows monitoring the number of received frames over different interfaces, and the number of planned and succeeded transmission and relaying attempts.

The API helps the user to estimate the efficiency of the advertiser configuration parameters and the scanning ability of the device. The number of the monitored parameters can be easily extended by customer values.

An application can read out and clean up statistics at any time.

API reference

group **bt_mesh_stat**

Statistic.

Functions

void **bt_mesh_stat_get**(struct *bt_mesh_statistic* *st)

Get mesh frame handling statistic.

Parameters

- **st** – BLE mesh statistic.

void **bt_mesh_stat_reset**(void)

Reset mesh frame handling statistic.

struct **bt_mesh_statistic**

#include <*statistic.h*> The structure that keeps statistics of mesh frames handling.

Public Members

uint32_t **rx_adv**

All received frames passed basic validation and decryption.

Received frames over advertiser.

uint32_t **rx_loopback**

Received frames over loopback.

uint32_t **rx_proxy**

Received frames over proxy.

uint32_t **rx_uknown**

Received over unknown interface.

uint32_t **tx_adv_relay_planned**

Counter of frames that were initiated to relay over advertiser bearer.

uint32_t **tx_adv_relay_succeeded**

Counter of frames that succeeded relaying over advertiser bearer.

uint32_t **tx_local_planned**

Counter of frames that were initiated to send over advertiser bearer locally.

uint32_t **tx_local_succeeded**

Counter of frames that succeeded to send over advertiser bearer locally.

uint32_t **tx_friend_planned**

Counter of frames that were initiated to send over friend bearer.

uint32_t **tx_friend_succeeded**

Counter of frames that succeeded to send over friend bearer.

Bluetooth Mesh Shell The Bluetooth Mesh shell subsystem provides a set of Bluetooth Mesh shell commands for the *Shell* module. It allows for testing and exploring the Bluetooth Mesh API through an interactive interface, without having to write an application.

The Bluetooth Mesh shell interface provides access to most Bluetooth Mesh features, including provisioning, configuration, and message sending.

Prerequisites The Bluetooth Mesh shell subsystem depends on the application to create the composition data and do the mesh initialization.

Application The Bluetooth Mesh shell subsystem is most easily used through the Bluetooth Mesh shell application under `tests/bluetooth/mesh_shell`. See *Shell* for information on how to connect and interact with the Bluetooth Mesh shell application.

Basic usage The Bluetooth Mesh shell subsystem adds a single `mesh` command, which holds a set of sub-commands. Every time the device boots up, make sure to call `mesh init` before any of the other Bluetooth Mesh shell commands can be called:

```
uart:~$ mesh init
```

This is done to ensure that all available log will be printed to the shell output.

Provisioning The mesh node must be provisioned to become part of the network. This is only necessary the first time the device boots up, as the device will remember its provisioning data between reboots.

The simplest way to provision the device is through self-provisioning. To do this the user must provision the device with the default network key and address `0x0001`, execute:

```
uart:~$ mesh prov local 0 0x0001
```

Since all mesh nodes use the same values for the default network key, this can be done on multiple devices, as long as they're assigned non-overlapping unicast addresses. Alternatively, to provision the device into an existing network, the unprovisioned beacon can be enabled with `mesh prov pb-adv on` or `mesh prov pb-gatt on`. The beacons can be picked up by an external provisioner, which can provision the node into its network.

Once the mesh node is part of a network, its transmission parameters can be controlled by the general configuration commands:

- To set the destination address, call `mesh target dst <Addr>`.
- To set the network key index, call `mesh target net <NetKeyIdx>`.
- To set the application key index, call `mesh target app <AppKeyIdx>`.

By default, the transmission parameters are set to send messages to the provisioned address and network key.

Configuration By setting the destination address to the local unicast address (`0x0001` in the `mesh prov local` command above), we can perform self-configuration through any of the *Models* commands.

A good first step is to read out the node's own composition data:

```
uart:~$ mesh models cfg get-comp
```

This prints a list of the composition data of the node, including a list of its model IDs.

Next, since the device has no application keys by default, it's a good idea to add one:

```
uart:~$ mesh models cfg appkey add 0 0
```

Message sending With an application key added (see above), the mesh node's transition parameters are all valid, and the Bluetooth Mesh shell can send raw mesh messages through the network.

For example, to send a Generic OnOff Set message, call:

```
uart:~$ mesh test net-send 82020100
```

Note

All multibyte fields model messages are in little endian, except the opcode.

The message will be sent to the current destination address, using the current network and application key indexes. As the destination address points to the local unicast address by default, the device will only send packets to itself. To change the destination address to the All Nodes broadcast address, call:

```
uart:~$ mesh target dst 0xffff
```

With the destination address set to `0xffff`, any other mesh nodes in the network with the configured network and application keys will receive and process the messages we send.

Note

To change the configuration of the device, the destination address must be set back to the local unicast address before issuing any configuration commands.

Sending raw mesh packets is a good way to test model message handler implementations during development, as it can be done without having to implement the sending model. By default, only the reception of the model messages can be tested this way, as the Bluetooth Mesh shell only includes the foundation models. To receive a packet in the mesh node, you have to add a model with a valid opcode handler list to the composition data in `subsys/bluetooth/mesh/shell.c`, and print the incoming message to the shell in the handler callback.

Parameter formats The Bluetooth Mesh shell commands are parsed with a variety of formats:

Table 29: Parameter formats

Type	Description	Example
Integers	The default format unless something else is specified. Can be either decimal or hexadecimal.	`1234`, `0xabcd01234`
Hexstrings	For raw byte arrays, like UUIDs, key values and message payloads, the parameters should be formatted as an unbroken string of hexadecimal values without any prefix.	`deadbeef01234`
Booleans	Boolean values are denoted in the API documentation as `<val(off, on)>`.	`on`, `off`, `enabled`, `disabled`, `1`, `0`

Commands The Bluetooth Mesh shell implements a large set of commands. Some of the commands accept parameters, which are mentioned in brackets after the command name. For example, `mesh lpn set <value: off, on>`. Mandatory parameters are marked with angle brackets (e.g. `<NetKeyIdx>`), and optional parameters are marked with square brackets (e.g. `[DstAddr]`).

The Bluetooth Mesh shell commands are divided into the following groups:

- *General configuration*

- *Target*
- *Low Power Node*
- *Testing*
- *Provisioning*
- *Proxy*
- *Models*
- *Configuration database*
- *Frame statistic*

Note

Some commands depend on specific features being enabled in the compile time configuration of the application. Not all features are enabled by default. The list of available Bluetooth mesh shell commands can be shown in the shell by calling `mesh` without any arguments.

General configuration

`mesh init`

Initialize the mesh shell. This command must be run before any other mesh command.

`mesh reset-local`

Reset the local mesh node to its initial unprovisioned state. This command will also clear the Configuration Database (CDB) if present.

Target The target commands enables the user to monitor and set the target destination address, network index and application index for the shell. These parameters are used by several commands, like provisioning, Configuration Client, etc.

`mesh target dst [DstAddr]`

Get or set the message destination address. The destination address determines where mesh packets are sent with the shell, but has no effect on modules outside the shell's control.

- `DstAddr`: If present, sets the new 16-bit mesh destination address. If omitted, the current destination address is printed.

`mesh target net [NetKeyIdx]`

Get or set the message network index. The network index determines which network key is used to encrypt mesh packets that are sent with the shell, but has no effect on modules outside the shell's control. The network key must already be added to the device, either through provisioning or by a Configuration Client.

- `NetKeyIdx`: If present, sets the new network index. If omitted, the current network index is printed.

`mesh target app [AppKeyIdx]`

Get or set the message application index. The application index determines which application key is used to encrypt mesh packets that are sent with the shell, but has no effect on modules outside the shell's control. The application key must already be added to the device by a Configuration Client, and must be bound to the current network index.

- `AppKeyIdx`: If present, sets the new application index. If omitted, the current application index is printed.

Low Power Node

`mesh lpn set <Val(off, on)>`

Enable or disable Low Power operation. Once enabled, the device will turn off its radio and start polling for friend nodes.

- `Val`: Sets whether Low Power operation is enabled.

`mesh lpn poll`

Perform a poll to the friend node, to receive any pending messages. Only available when LPN is enabled.

Testing

`mesh test net-send <HexString>`

Send a raw mesh message with the current destination address, network and application index. The message opcode must be encoded manually.

- `HexString` Raw hexadecimal representation of the message to send.

`mesh test iv-update`

Force an IV update.

`mesh test iv-update-test <Val(off, on)>`

Set the IV update test mode. In test mode, the IV update timing requirements are bypassed.

- `Val`: Enable or disable the IV update test mode.

`mesh test rpl-clear`

Clear the replay protection list, forcing the node to forget all received messages.

Warning

Clearing the replay protection list breaks the security mechanisms of the mesh node, making it susceptible to message replay attacks. This should never be performed in a real deployment.

Health Server Test

`mesh test health-srv add-fault <FaultID>`

Register a new Health Server Fault for the Linux Foundation Company ID.

- `FaultID`: ID of the fault to register (`0x0001` to `0xFFFF`)

`mesh test health-srv del-fault [FaultID]`

Remove registered Health Server faults for the Linux Foundation Company ID.

- `FaultID`: If present, the given fault ID will be deleted. If omitted, all registered faults will be cleared.

Provisioning To allow a device to broadcast connectable unprovisioned beacons, the `CONFIG_BT_MESH_PROVISIONEE` configuration option must be enabled, along with the `CONFIG_BT_MESH_PB_GATT` option.

`mesh prov pb-gatt <Val(off, on)>`

Start or stop advertising a connectable unprovisioned beacon. The connectable unprovisioned beacon allows the mesh node to be discovered by nearby GATT based provisioners, and provisioned through the GATT bearer.

- `Val`: Enable or disable provisioning with GATT

To allow a device to broadcast unprovisioned beacons, the `CONFIG_BT_MESH_PROVISIONEE` configuration option must be enabled, along with the `CONFIG_BT_MESH_PB_ADV` option.

`mesh prov pb-adv <Val(off, on)>`

Start or stop advertising the unprovisioned beacon. The unprovisioned beacon allows the mesh node to be discovered by nearby advertising-based provisioners, and provisioned through the advertising bearer.

- `Val`: Enable or disable provisioning with advertiser

To allow a device to provision devices, the `CONFIG_BT_MESH_PROVISIONER` and `CONFIG_BT_MESH_PB_ADV` configuration options must be enabled.

`mesh prov remote-adv <UUID(1-16 hex)> <NetKeyIdx> <Addr> <AttDur(s)>`

Provision a nearby device into the mesh. The mesh node starts scanning for unprovisioned beacons with the given UUID. Once found, the unprovisioned device will be added to the mesh network with the given unicast address, and given the network key indicated by `NetKeyIdx`.

- `UUID`: UUID of the unprovisioned device. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest.
- `NetKeyIdx`: Index of the network key to pass to the device.
- `Addr`: First unicast address to assign to the unprovisioned device. The device will occupy as many addresses as it has elements, and all must be available.
- `AttDur`: The duration in seconds the unprovisioned device will identify itself for, if supported. See *Attention state* for details.

To allow a device to provision devices over GATT, the `CONFIG_BT_MESH_PROVISIONER` and `CONFIG_BT_MESH_PB_GATT_CLIENT` configuration options must be enabled.

`mesh prov remote-gatt <UUID(1-16 hex)> <NetKeyIdx> <Addr> <AttDur(s)>`

Provision a nearby device into the mesh. The mesh node starts scanning for connectable advertising for PB-GATT with the given UUID. Once found, the unprovisioned device will be added to the mesh network with the given unicast address, and given the network key indicated by `NetKeyIdx`.

- `UUID`: UUID of the unprovisioned device. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest.
- `NetKeyIdx`: Index of the network key to pass to the device.
- `Addr`: First unicast address to assign to the unprovisioned device. The device will occupy as many addresses as it has elements, and all must be available.
- `AttDur`: The duration in seconds the unprovisioned device will identify itself for, if supported. See *Attention state* for details.

`mesh prov uuid [UUID(1-16 hex)]`

Get or set the mesh node's UUID, used in the unprovisioned beacons.

- `UUID`: If present, new 128-bit UUID value. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest. If omitted, the current UUID will be printed. To enable this command, the `CONFIG_BT_MESH_SHELL_PROV_CTX_INSTANCE` option must be enabled.

`mesh prov input-num <Number>`

Input a numeric OOB authentication value. Only valid when prompted by the shell during provisioning. The input number must match the number presented by the other participant in the provisioning.

- `Number`: Decimal authentication number.

`mesh prov input-str <String>`

Input an alphanumeric OOB authentication value. Only valid when prompted by the shell during provisioning. The input string must match the string presented by the other participant in the provisioning.

- `String`: Unquoted alphanumeric authentication string.

`mesh prov static-oob [Val(1-32 hex)]`

Set or clear the static OOB authentication value. The static OOB authentication value must be set before provisioning starts to have any effect. The static OOB value must be same on both participants in the provisioning. To enable this command, the `CONFIG_BT_MESH_SHELL_PROV_CTX_INSTANCE` option must be enabled.

- `Val`: If present, indicates the new hexadecimal value of the static OOB. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest. If omitted, the static OOB value is cleared.

`mesh prov local <NetKeyIdx> <Addr> [IVI]`

Provision the mesh node itself. If the Configuration database is enabled, the network key must be created. Otherwise, the default key value is used.

- `NetKeyIdx`: Index of the network key to provision.

- `Addr`: First unicast address to assign to the device. The device will occupy as many addresses as it has elements, and all must be available.
- `IVI`: Indicates the current network IV index. Defaults to 0 if omitted.

`mesh prov beacon-listen <Val(off, on)>`

Enable or disable printing of incoming unprovisioned beacons. Allows a provisioner device to detect nearby unprovisioned devices and provision them. To enable this command, the `CONFIG_BT_MESH_SHELL_PROV_CTX_INSTANCE` option must be enabled.

- `Val`: Whether to enable the unprovisioned beacon printing.

`mesh prov remote-pub-key <PubKey>`

Provide Device public key.

- `PubKey` - Device public key in big-endian.

`mesh prov auth-method input <Action> <Size>`

From the provisioner device, instruct the unprovisioned device to use the specified Input OOB authentication action.

- `Action` - Input action. Allowed values:
  - `0` - No input action.
  - `1` - Push action set.
  - `2` - Twist action set.
  - `4` - Enter number action set.
  - `8` - Enter String action set.
- `Size` - Authentication size.

`mesh prov auth-method output <Action> <Size>`

From the provisioner device, instruct the unprovisioned device to use the specified Output OOB authentication action.

- `Action` - Output action. Allowed values:
  - `0` - No output action.
  - `1` - Blink action set.
  - `2` - Vibrate action set.
  - `4` - Display number action set.
  - `8` - Display String action set.
- `Size` - Authentication size.

`mesh prov auth-method static <Val(1-16 hex)>`

From the provisioner device, instruct the unprovisioned device to use static OOB authentication, and use the given static authentication value when provisioning.

- `Val` - Static OOB value. Providing a hex-string shorter than 32 bytes will populate the N most significant bytes of the array and zero-pad the rest.

`mesh prov auth-method none`

From the provisioner device, don't use any authentication when provisioning new devices. This is the default behavior.

Proxy The Proxy Server module is an optional mesh subsystem that can be enabled through the `CONFIG_BT_MESH_GATT_PROXY` configuration option.

`mesh proxy identity-enable`

Enable the Proxy Node Identity beacon, allowing Proxy devices to connect explicitly to this device. The beacon will run for 60 seconds before the node returns to normal Proxy beacons.

The Proxy Client module is an optional mesh subsystem that can be enabled through the `CONFIG_BT_MESH_PROXY_CLIENT` configuration option.

`mesh proxy connect <NetKeyIdx>`

Auto-Connect a nearby proxy server into the mesh.

- `NetKeyIdx`: Index of the network key to connect.

`mesh proxy disconnect <NetKeyIdx>`

Disconnect the existing proxy connection.

- `NetKeyIdx`: Index of the network key to disconnect.

`mesh proxy solicit <NetKeyIdx>`

Begin Proxy Solicitation of a subnet. Support of this feature can be enabled through the `CONFIG_BT_MESH_PROXY_SOLICITATION` configuration option.

- `NetKeyIdx`: Index of the network key to send Solicitation PDUs to.

Models

Configuration Client The Configuration Client model is an optional mesh subsystem that can be enabled through the `CONFIG_BT_MESH_CFG_CLI` configuration option. This is implemented as a separate module (`mesh models cfg`) inside the `mesh models` subcommand list. This module will work on any instance of the Configuration Client model if the mentioned shell configuration options is enabled, and as long as the Configuration Client model is present in the model composition of the application. This shell module can be used for configuring itself and other nodes in the mesh network.

The Configuration Client uses general message parameters set by `mesh target dst` and `mesh target net` to target specific nodes. When the Bluetooth Mesh shell node is provisioned, given that the `CONFIG_BT_MESH_SHELL_PROV_CTX_INSTANCE` option is enabled with the shell provisioning context initialized, the Configuration Client model targets itself by default. Similarly, when another node has been provisioned by the Bluetooth Mesh shell, the Configuration Client model targets the new node. In most common use-cases, the Configuration Client is depending on the provisioning features and the Configuration database to be fully functional. The Configuration Client always sends messages using the Device key bound to the destination address, so it will only be able to configure itself and the mesh nodes it provisioned. The following steps are an example of how you can set up a device to start using the Configuration Client commands:

- Initialize the client node (`mesh init`).
- Create the CDB (`mesh cdb create`).
- Provision the local device (`mesh prov local`).
- The shell module should now target itself.
- Monitor the composition data of the local node (`mesh models cfg get-comp`).
- Configure the local node as desired with the Configuration Client commands.
- Provision other devices (`mesh prov beacon-listen`) (`mesh prov remote-adv`) (`mesh prov remote-gatt`).
- The shell module should now target the newly added node.
- Monitor the newly provisioned nodes and their addresses (`mesh cdb show`).
- Monitor the composition data of the target device (`mesh models cfg get-comp`).
- Configure the node as desired with the Configuration Client commands.

`mesh models cfg target get`

Get the target Configuration server for the Configuration Client model.

`mesh models cfg help`

Print information for the Configuration Client shell module.

`mesh models cfg reset`

Reset the target device.

`mesh models cfg timeout [Timeout(s)]`

Get and set the Config Client model timeout used during message sending.

- `Timeout`: If present, set the Config Client model timeout in seconds. If omitted, the current timeout is printed.

`mesh models cfg get-comp [Page]`

Read a composition data page. The full composition data page will be printed. If the target does not have the given page, it will return the last page before it.

- `Page`: The composition data page to request. Defaults to 0 if omitted.

`mesh models cfg beacon [Val(off, on)]`

Get or set the network beacon transmission.

- `Val`: If present, enables or disables sending of the network beacon. If omitted, the current network beacon state is printed.

`mesh models cfg ttl [TTL]`

Get or set the default TTL value.

- `TTL`: If present, sets the new default TTL value. Legal TTL values are 0x00 and 0x02-0x7f. If omitted, the current default TTL value is printed.

`mesh models cfg friend [Val(off, on)]`

Get or set the Friend feature.

- `Val`: If present, enables or disables the Friend feature. If omitted, the current Friend feature state is printed:
  - `0x00`: The feature is supported, but disabled.
  - `0x01`: The feature is enabled.
  - `0x02`: The feature is not supported.

`mesh models cfg gatt-proxy [Val(off, on)]`

Get or set the GATT Proxy feature.

- `Val`: If present, enables or disables the GATT Proxy feature. If omitted, the current GATT Proxy feature state is printed:
  - `0x00`: The feature is supported, but disabled.
  - `0x01`: The feature is enabled.
  - `0x02`: The feature is not supported.

`mesh models cfg relay [<Val(off, on)> [<Count> [Int(ms)]]]`

Get or set the Relay feature and its parameters.

- `Val`: If present, enables or disables the Relay feature. If omitted, the current Relay feature state is printed:
  - `0x00`: The feature is supported, but disabled.
  - `0x01`: The feature is enabled.
  - `0x02`: The feature is not supported.
- `Count`: Sets the new relay retransmit count if `val` is `on`. Ignored if `val` is `off`. Legal retransmit count is 0-7. Defaults to `2` if omitted.
- `Int`: Sets the new relay retransmit interval in milliseconds if `val` is `on`. Legal interval range is 10-320 milliseconds. Ignored if `val` is `off`. Defaults to `20` if omitted.

`mesh models cfg node-id <NetKeyIdx> [Identity]`

Get or Set of current Node Identity state of a subnet.

- `NetKeyIdx`: The network key index to Get/Set.
- `Identity`: If present, sets the identity of Node Identity state.

`mesh models cfg polltimeout-get <LPNAddr>`

Get current value of the PollTimeout timer of the LPN within a Friend node.

- `LPNAddr` Address of Low Power node.

`mesh models cfg net-transmit-param [<Count> <Int(ms)>]`

Get or set the network transmit parameters.

- `Count`: Sets the number of additional network transmits for every sent message. Legal retransmit count is 0-7.
- `Int`: Sets the new network retransmit interval in milliseconds. Legal interval range is 10-320 milliseconds.

`mesh models cfg netkey add <NetKeyIdx> [Key(1-16 hex)]`

Add a network key to the target node. Adds the key to the Configuration Database if enabled.

- `NetKeyIdx`: The network key index to add.
- `Key`: If present, sets the key value as a 128-bit hexadecimal value. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest. Only valid if the key does not already exist in the Configuration Database. If omitted, the default key value is used.

`mesh models cfg netkey upd <NetKeyIdx> [Key(1-16 hex)]`

Update a network key to the target node.

- `NetKeyIdx`: The network key index to updated.
- `Key`: If present, sets the key value as a 128-bit hexadecimal value. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest. If omitted, the default key value is used.

`mesh models cfg netkey get`

Get a list of known network key indexes.

`mesh models cfg netkey del <NetKeyIdx>`

Delete a network key from the target node.

- `NetKeyIdx`: The network key index to delete.

`mesh models cfg appkey add <NetKeyIdx> <AppKeyIdx> [Key(1-16 hex)]`

Add an application key to the target node. Adds the key to the Configuration Database if enabled.

- `NetKeyIdx`: The network key index the application key is bound to.
- `AppKeyIdx`: The application key index to add.
- `Key`: If present, sets the key value as a 128-bit hexadecimal value. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest. Only valid if the key does not already exist in the Configuration Database. If omitted, the default key value is used.

`mesh models cfg appkey upd <NetKeyIdx> <AppKeyIdx> [Key(1-16 hex)]`

Update an application key to the target node.

- `NetKeyIdx`: The network key index the application key is bound to.
- `AppKeyIdx`: The application key index to update.
- `Key`: If present, sets the key value as a 128-bit hexadecimal value. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest. If omitted, the default key value is used.

`mesh models cfg appkey get <NetKeyIdx>`

Get a list of known application key indexes bound to the given network key index.

- `NetKeyIdx`: Network key indexes to get a list of application key indexes from.

`mesh models cfg appkey del <NetKeyIdx> <AppKeyIdx>`

Delete an application key from the target node.

- `NetKeyIdx`: The network key index the application key is bound to.
- `AppKeyIdx`: The application key index to delete.

`mesh models cfg model app-bind <Addr> <AppKeyIdx> <MID> [CID]`

Bind an application key to a model. Models can only encrypt and decrypt messages sent with application keys they are bound to.

- `Addr`: Address of the element the model is on.
- `AppKeyIdx`: The application key to bind to the model.
- `MID`: The model ID of the model to bind the key to.
- `CID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh models cfg model app-unbind <Addr> <AppKeyIdx> <MID> [CID]`

Unbind an application key from a model.

- `Addr`: Address of the element the model is on.
- `AppKeyIdx`: The application key to unbind from the model.
- `MID`: The model ID of the model to unbind the key from.
- `CID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh models cfg model app-get <ElemAddr> <MID> [CID]`

Get a list of application keys bound to a model.

- `ElemAddr`: Address of the element the model is on.
- `MID`: The model ID of the model to get the bound keys of.
- `CID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh models cfg model pub <Addr> <MID> [CID] [<PubAddr> <AppKeyIdx> <Cred(off, on)> <TTL> <PerRes> <PerSteps> <Count> <Int(ms)>]`

Get or set the publication parameters of a model. If all publication parameters are included, they become the new publication parameters of the model. If all publication parameters are omitted, print the current publication parameters of the model.

- `Addr`: Address of the element the model is on.
- `MID`: The model ID of the model to get the bound keys of.
- `CID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

Publication parameters:

- `PubAddr`: The destination address to publish to.
- `AppKeyIdx`: The application key index to publish with.
- `Cred`: Whether to publish with Friendship credentials when acting as a Low Power Node.
- `TTL`: TTL value to publish with (`0x00` to `0x07f`).
- `PerRes`: Resolution of the publication period steps:
  - `0x00`: The Step Resolution is 100 milliseconds
  - `0x01`: The Step Resolution is 1 second
  - `0x02`: The Step Resolution is 10 seconds
  - `0x03`: The Step Resolution is 10 minutes
- `PerSteps`: Number of publication period steps, or 0 to disable periodic publication.
- `Count`: Number of retransmission for each published message (`0` to 7).
- `Int` The interval between each retransmission, in milliseconds. Must be a multiple of 50.

`mesh models cfg model pub-va <Addr> <UUID(1-16 hex)> <AppKeyIdx> <Cred(off, on)> <TTL> <PerRes> <PerSteps> <Count> <Int(ms)> <MID> [CID]`

Set the publication parameters of a model.

- `Addr`: Address of the element the model is on.
- `MID`: The model ID of the model to get the bound keys of.
- `CID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

Publication parameters:

- `UUID`: The destination virtual address to publish to. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest.
- `AppKeyIdx`: The application key index to publish with.
- `Cred`: Whether to publish with Friendship credentials when acting as a Low Power Node.
- `TTL`: TTL value to publish with (`0x00` to `0x07f`).
- `PerRes`: Resolution of the publication period steps:
  - `0x00`: The Step Resolution is 100 milliseconds

  - `0x01`: The Step Resolution is 1 second
  - `0x02`: The Step Resolution is 10 seconds
  - `0x03`: The Step Resolution is 10 minutes
- `PerSteps`: Number of publication period steps, or 0 to disable periodic publication.
- `Count`: Number of retransmission for each published message (`0` to 7).
- `Int` The interval between each retransmission, in milliseconds. Must be a multiple of 50.

`mesh models cfg model sub-add <ElemAddr> <SubAddr> <MID> [CID]`

Subscription the model to a group address. Models only receive messages sent to their unicast address or a group or virtual address they subscribe to. Models may subscribe to multiple group and virtual addresses.

- `ElemAddr`: Address of the element the model is on.
- `SubAddr`: 16-bit group address the model should subscribe to (`0xc000` to `0xFEFF`).
- `MID`: The model ID of the model to add the subscription to.
- `CID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh models cfg model sub-del <ElemAddr> <SubAddr> <MID> [CID]`

Unsubscribe a model from a group address.

- `ElemAddr`: Address of the element the model is on.
- `SubAddr`: 16-bit group address the model should remove from its subscription list (`0xc000` to `0xFEFF`).
- `MID`: The model ID of the model to add the subscription to.
- `CID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh models cfg model sub-add-va <ElemAddr> <LabelUUID(1-16 hex)> <MID> [CID]`

Subscribe the model to a virtual address. Models only receive messages sent to their unicast address or a group or virtual address they subscribe to. Models may subscribe to multiple group and virtual addresses.

- `ElemAddr`: Address of the element the model is on.
- `LabelUUID`: 128-bit label UUID of the virtual address to subscribe to. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest.
- `MID`: The model ID of the model to add the subscription to.
- `CID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh models cfg model sub-del-va <ElemAddr> <LabelUUID(1-16 hex)> <MID> [CID]`

Unsubscribe a model from a virtual address.

- `ElemAddr`: Address of the element the model is on.
- `LabelUUID`: 128-bit label UUID of the virtual address to remove the subscription of. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest.
- `MID`: The model ID of the model to add the subscription to.
- `CID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh models cfg model sub-ow <ElemAddr> <SubAddr> <MID> [CID]`

Overwrite all model subscriptions with a single new group address.

- `ElemAddr`: Address of the element the model is on.
- `SubAddr`: 16-bit group address the model should added to the subscription list (`0xc000` to `0xFEFF`).
- `MID`: The model ID of the model to add the subscription to.
- `CID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh models cfg model sub-ow-va <ElemAddr> <LabelUUID(1-16 hex)> <MID> [CID]`

Overwrite all model subscriptions with a single new virtual address. Models only receive messages sent to their unicast address or a group or virtual address they subscribe to. Models may subscribe to multiple group and virtual addresses.

- `ElemAddr`: Address of the element the model is on.
- `LabelUUID`: 128-bit label UUID of the virtual address as the new Address to be added to the subscription list. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest.
- `MID`: The model ID of the model to add the subscription to.
- `CID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh models cfg model sub-del-all <ElemAddr> <MID> [CID]`

Remove all group and virtual address subscriptions from of a model.

- `ElemAddr`: Address of the element the model is on.
- `MID`: The model ID of the model to Unsubscribe all.
- `CID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh models cfg model sub-get <ElemAddr> <MID> [CID]`

Get a list of addresses the model subscribes to.

- `ElemAddr`: Address of the element the model is on.
- `MID`: The model ID of the model to get the subscription list of.

- `CID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh models cfg krp <NetKeyIdx> [Phase]`

Get or set the key refresh phase of a subnet.

- `NetKeyIdx`: The identified network key used to Get/Set the current Key Refresh Phase state.
- `Phase`: New Key Refresh Phase. Valid phases are:
  - `0x00`: Normal operation; Key Refresh procedure is not active
  - `0x01`: First phase of Key Refresh procedure
  - `0x02`: Second phase of Key Refresh procedure

`mesh models cfg hb-sub [<Src> <Dst> <Per>]`

Get or set the Heartbeat subscription parameters. A node only receives Heartbeat messages matching the Heartbeat subscription parameters. Sets the Heartbeat subscription parameters if present, or prints the current Heartbeat subscription parameters if called with no parameters.

- `Src`: Unicast source address to receive Heartbeat messages from.
- `Dst`: Destination address to receive Heartbeat messages on.
- `Per`: Logarithmic representation of the Heartbeat subscription period:
  - `0`: Heartbeat subscription will be disabled.
  - `1` to `17`: The node will subscribe to Heartbeat messages for $2^{(\text{period} - 1)}$ seconds.

`mesh models cfg hb-pub [<Dst> <Count> <Per> <TTL> <Features> <NetKeyIdx>]`

Get or set the Heartbeat publication parameters. Sets the Heartbeat publication parameters if present, or prints the current Heartbeat publication parameters if called with no parameters.

- `Dst`: Destination address to publish Heartbeat messages to.
- `Count`: Logarithmic representation of the number of Heartbeat messages to publish periodically:
  - `0`: Heartbeat messages are not published periodically.
  - `1` to `17`: The node will periodically publish $2^{(\text{count} - 1)}$ Heartbeat messages.
  - `255`: Heartbeat messages will be published periodically indefinitely.
- `Per`: Logarithmic representation of the Heartbeat publication period:
  - `0`: Heartbeat messages are not published periodically.
  - `1` to `17`: The node will publish Heartbeat messages every $2^{(\text{period} - 1)}$ seconds.
- `TTL`: The TTL value to publish Heartbeat messages with (`0x00` to `0x7f`).
- `Features`: Bitfield of features that should trigger a Heartbeat publication when changed:
  - `Bit 0`: Relay feature.
  - `Bit 1`: Proxy feature.
  - `Bit 2`: Friend feature.

  - Bit 3: Low Power feature.
- NetKeyIdx: Index of the network key to publish Heartbeat messages with.

Health Client The Health Client model is an optional mesh subsystem that can be enabled through the CONFIG_BT_MESH_HEALTH_CLI configuration option. This is implemented as a separate module (mesh models health) inside the mesh models subcommand list. This module will work on any instance of the Health Client model if the mentioned shell configuration options is enabled, and as long as one or more Health Client model(s) is present in the model composition of the application. This shell module can be used to trigger interaction between Health Clients and Servers on devices in a Mesh network.

By default, the module will choose the first Health Client instance in the model composition when using the Health Client commands. To choose a specific Health Client instance the user can utilize the commands mesh models health instance set and mesh models health instance get-all.

The Health Client may use the general messages parameters set by mesh target dst, mesh target net and mesh target app to target specific nodes. If the shell target destination address is set to zero, the targeted Health Client will attempt to publish messages using its configured publication parameters.

`mesh models health instance set <ElemIdx>`

Set the Health Client model instance to use.

- ElemIdx: Element index of Health Client model.

`mesh models health instance get-all`

Prints all available Health Client model instances on the device.

`mesh models health fault-get <CID>`

Get a list of registered faults for a Company ID.

- CID: Company ID to get faults for.

`mesh models health fault-clear <CID>`

Clear the list of faults for a Company ID.

- CID: Company ID to clear the faults for.

`mesh models health fault-clear-unack <CID>`

Clear the list of faults for a Company ID without requesting a response.

- CID: Company ID to clear the faults for.

`mesh models health fault-test <CID> <TestID>`

Invoke a self-test procedure, and show a list of triggered faults.

- CID: Company ID to perform self-tests for.
- TestID: Test to perform.

`mesh models health fault-test-unack <CID> <TestID>`

Invoke a self-test procedure without requesting a response.

- `CID`: Company ID to perform self-tests for.
- `TestID`: Test to perform.

`mesh models health period-get`

Get the current Health Server publish period divisor.

`mesh models health period-set <Divisor>`

Set the current Health Server publish period divisor. When a fault is detected, the Health Server will start publishing is fault status with a reduced interval. The reduced interval is determined by the Health Server publish period divisor: Fault publish period = Publish period / 2^{divisor}.

- `Divisor`: The new Health Server publish period divisor.

`mesh models health period-set-unack <Divisor>`

Set the current Health Server publish period divisor. When a fault is detected, the Health Server will start publishing is fault status with a reduced interval. The reduced interval is determined by the Health Server publish period divisor: Fault publish period = Publish period / 2^{divisor}.

- `Divisor`: The new Health Server publish period divisor.

`mesh models health attention-get`

Get the current Health Server attention state.

`mesh models health attention-set <Time(s)>`

Enable the Health Server attention state for some time.

- `Time`: Duration of the attention state, in seconds (`0` to `255`)

`mesh models health attention-set-unack <Time(s)>`

Enable the Health Server attention state for some time without requesting a response.

- `Time`: Duration of the attention state, in seconds (`0` to `255`)

Binary Large Object (BLOB) Transfer Client model The *BLOB Transfer Client* can be added to the mesh shell by enabling the `CONFIG_BT_MESH_BLOB_CLI` option, and disabling the `CONFIG_BT_MESH_DFU_CLI` option.

`mesh models blob cli target <Addr>`

Add a Target node for the next BLOB transfer.

- `Addr`: Unicast address of the Target node's BLOB Transfer Server model.

`mesh models blob cli bounds [<Group>]`

Get the total boundary parameters of all Target nodes.

- `Group`: Optional group address to use when communicating with Target nodes. If omitted, the BLOB Transfer Client will address each Target node individually.

`mesh models blob cli tx <Id> <Size> <BlockSizeLog> <ChunkSize> [<Group> [<Mode(push, pull)>]]`

Perform a BLOB transfer to Target nodes. The BLOB Transfer Client will send a dummy BLOB to all Target nodes, then post a message when the transfer is completed. Note that all Target nodes must first be configured to receive the transfer using the `mesh models blob srv rx` command.

- `Id`: 64-bit BLOB transfer ID.
- `Size`: Size of the BLOB in bytes.
- `BlockSizeLog`: Logarithmic representation of the BLOB's block size. The final block size will be `1 << block size log` bytes.
- `ChunkSize`: Chunk size in bytes.
- `Group`: Optional group address to use when communicating with Target nodes. If omitted or set to 0, the BLOB Transfer Client will address each Target node individually.
- `Mode`: BLOB transfer mode to use. Must be either `push` (Push BLOB Transfer Mode) or `pull` (Pull BLOB Transfer Mode). If omitted, `push` will be used by default.

`mesh models blob cli tx-cancel`

Cancel an ongoing BLOB transfer.

`mesh models blob cli tx-get [Group]`

Determine the progress of a previously running BLOB transfer. Can be used when not performing a BLOB transfer.

- `Group`: Optional group address to use when communicating with Target nodes. If omitted or set to 0, the BLOB Transfer Client will address each Target node individually.

`mesh models blob cli tx-suspend`

Suspend the ongoing BLOB transfer.

`mesh models blob cli tx-resume`

Resume the suspended BLOB transfer.

`mesh models blob cli instance-set <ElemIdx>`

Use the BLOB Transfer Client model instance on the specified element when using the other BLOB Transfer Client model commands.

- `ElemIdx`: The element on which to find the BLOB Transfer Client model instance to use.

`mesh models blob cli instance-get-all`

Get a list of all BLOB Transfer Client model instances on the node.

BLOB Transfer Server model The *BLOB Transfer Server* can be added to the mesh shell by enabling the `CONFIG_BT_MESH_BLOB_SRV` option. The BLOB Transfer Server model is capable of receiving any BLOB data, but the implementation in the mesh shell will discard the incoming data.

`mesh models blob srv rx <ID> [<TimeoutBase(10s steps)>]`

Prepare to receive a BLOB transfer.

- `ID`: 64-bit BLOB transfer ID to receive.
- `TimeoutBase`: Optional additional time to wait for client messages, in 10-second increments.

`mesh models blob srv rx-cancel`

Cancel an ongoing BLOB transfer.

`mesh models blob srv instance-set <ElemIdx>`

Use the BLOB Transfer Server model instance on the specified element when using the other BLOB Transfer Server model commands.

- `ElemIdx`: The element on which to find the BLOB Transfer Server model instance to use.

`mesh models blob srv instance-get-all`

Get a list of all BLOB Transfer Server model instances on the node.

Firmware Update Client model The Firmware Update Client model can be added to the mesh shell by enabling configuration options `CONFIG_BT_MESH_BLOB_CLI` and `CONFIG_BT_MESH_DFU_CLI`. The Firmware Update Client demonstrates the firmware update Distributor role by transferring a dummy firmware update to a set of Target nodes.

`mesh models dfu slot add <Size> <FwID> [<Metadata>]`

Add a virtual DFU image slot that can be transferred as a DFU image. The image slot will be assigned an image slot index, which is printed as a response, and can be used to reference the slot in other commands. To update the image slot, remove it using the `mesh models dfu slot del` shell command and then add it again.

- `Size`: DFU image slot size in bytes.
- `FwID`: Firmware ID, formatted as a hexstring.
- `Metadata`: Optional firmware metadata, formatted as a hexstring.

`mesh models dfu slot del <SlotIdx>`

Delete the DFU image slot at the given index.

- `SlotIdx`: Index of the slot to delete.

`mesh models dfu slot get <SlotIdx>`

Get all available information about a DFU image slot.

- `SlotIdx`: Index of the slot to get.

`mesh models dfu cli target <Addr> <ImgIdx>`

Add a Target node.

- `Addr`: Unicast address of the Target node.
- `ImgIdx`: Image index to address on the Target node.

`mesh models dfu cli target-state`

Check the DFU Target state of the device at the configured destination address.

`mesh models dfu cli target-imgs [<MaxCount>]`

Get a list of DFU images on the device at the configured destination address.

- `MaxCount`: Optional maximum number of images to return. If omitted, there's no limit on the number of returned images.

`mesh models dfu cli target-check <SlotIdx> <TargetImgIdx>`

Check whether the device at the configured destination address will accept a DFU transfer from the given DFU image slot to the Target node's DFU image at the given index, and what the effect would be.

- `SlotIdx`: Index of the local DFU image slot to check.
- `TargetImgIdx`: Index of the Target node's DFU image to check.

`mesh models dfu cli send <SlotIdx> [<Group>]`

Start a DFU transfer to all added Target nodes.

- `SlotIdx`: Index of the local DFU image slot to send.
- `Group`: Optional group address to use when communicating with the Target nodes. If omitted, the Firmware Update Client will address each Target node individually.

`mesh models dfu cli cancel [<Addr>]`

Cancel the DFU procedure at any state on a specific Target node or on all Target nodes. When a Target node address is provided, the Firmware Update Client model will try to cancel the DFU procedure on the provided Target node. Otherwise, the Firmware Update Client model will try to cancel the ongoing DFU procedure on all Target nodes.

- `Addr`: Optional unicast address of a Target node on which to cancel the DFU procedure.

`mesh models dfu cli apply`

Apply the most recent DFU transfer on all Target nodes. Can only be called after a DFU transfer is completed.

`mesh models dfu cli confirm`

Confirm that the most recent DFU transfer was successfully applied on all Target nodes. Can only be called after a DFU transfer is completed and applied.

`mesh models dfu cli suspend`

Suspend the ongoing DFU transfer.

`mesh models dfu cli resume`

Resume the suspended DFU transfer.

`mesh models dfu cli progress`

Check the progress of the current transfer.

`mesh models dfu cli instance-set <ElemIdx>`

Use the Firmware Update Client model instance on the specified element when using the other Firmware Update Client model commands.

- `ElemIdx`: The element on which to find the Firmware Update Client model instance to use.

`mesh models dfu cli instance-get-all`

Get a list of all Firmware Update Client model instances on the node.

Firmware Update Server model The Firmware Update Server model can be added to the mesh shell by enabling configuration options `CONFIG_BT_MESH_BLOB_SRV` and `CONFIG_BT_MESH_DFU_SRV`. The Firmware Update Server demonstrates the firmware update Target role by accepting any firmware update. The mesh shell Firmware Update Server will discard the incoming firmware data, but otherwise behave as a proper firmware update Target node.

`mesh models dfu srv applied`

Mark the most recent DFU transfer as applied. Can only be called after a DFU transfer is completed, and the Distributor has requested that the transfer is applied.

As the mesh shell Firmware Update Server doesn't actually apply the incoming firmware image, this command can be used to emulate an applied status, to notify the Distributor that the transfer was successful.

`mesh models dfu srv progress`

Check the progress of the current transfer.

`mesh models dfu srv rx-cancel`

Cancel incoming DFU transfer.

`mesh models dfu srv instance-set <ElemIdx>`

Use the Firmware Update Server model instance on the specified element when using the other Firmware Update Server model commands.

- `ElemIdx`: The element on which to find the Firmware Update Server model instance to use.

`mesh models dfu srv instance-get-all`

Get a list of all Firmware Update Server model instances on the node.

Firmware Distribution Server model The Firmware Distribution Server model commands can be added to the mesh shell by enabling the `CONFIG_BT_MESH_DFD_SRV` configuration option. The shell commands for this model mirror the messages sent to the server by a Firmware Distribution Client model. To use these commands, a Firmware Distribution Server must be instantiated by the application.

`mesh models dfd receivers-add <Addr>,<FwIdx>[;<Addr>,<FwIdx>]...`

Add receivers to the Firmware Distribution Server. Supply receivers as a list of comma-separated addr,fw_idx pairs, separated by semicolons, for example, `0x0001,0;0x0002,0;0x0004,1`. Do not use spaces in the receiver list. Repeated calls to this command will continue populating the receivers list until `mesh models dfd receivers-delete-all` is called.

- `Addr`: Address of the receiving node(s).
- `FwIdx`: Index of the firmware slot to send to `Addr`.

`mesh models dfd receivers-delete-all`

Delete all receivers from the server.

`mesh models dfd receivers-get <First> <Count>`

Get a list of info about firmware receivers.

- `First`: Index of the first receiver to get from the receiver list.
- `Count`: The number of receivers for which to get info.

`mesh models dfd capabilities-get`

Get the capabilities of the server.

`mesh models dfd get`

Get information about the current distribution state, phase and the transfer parameters.

`mesh models dfd start <AppKeyIdx> <SlotIdx> [<Group> [<PolicyApply> [<TTL> [<TimeoutBase> [<XferMode>]]]]]`

Start the firmware distribution.

- `AppKeyIdx`: Application index to use for sending. The common application key should be bound to the Firmware Update and BLOB Transfer models on the Distributor and Target nodes.
- `SlotIdx`: Index of the local image slot to send.
- `Group`: Optional group address to use when communicating with the Target nodes. If omitted, the Firmware Distribution Server will address each Target node individually. To keep addressing each Target node individually while changing other arguments, set this argument value to 0.
- `PolicyApply`: Optional field that corresponds to the update policy. Setting this to `true` will make the Firmware Distribution Server apply the image immediately after the transfer is completed.
- `TTL`: Optional. TTL value to use when sending. Defaults to configured default TTL.
- `TimeoutBase`: Optional additional value used to calculate timeout values in the firmware distribution process, in 10-second increments.. See *Transfer timeout* for information about how `timeout_base` is used to calculate the transfer timeout. Defaults to 0.
- `XferMode`: Optional BLOB transfer mode. 1 = Push mode (Push BLOB Transfer Mode), 2 = Pull mode (Pull BLOB Transfer Mode). Defaults to Push mode.

`mesh models dfd suspend`

Suspends the ongoing distribution.

`mesh models dfd cancel`

Cancel the ongoing distribution.

`mesh models dfd apply`

Apply the distributed firmware.

`mesh models dfd fw-get <FwID>`

Get information about the firmware image uploaded to the server.

- `FwID`: Firmware ID of the image to get.

`mesh models dfd fw-get-by-idx <Idx>`

Get information about the firmware image uploaded to the server in a specific slot.

- `Idx`: Index of the slot to get the image from.

`mesh models dfd fw-delete <FwID>`

Delete a firmware image from the server.

- `FwID`: Firmware ID of the image to delete.

`mesh models dfd fw-delete-all`

Delete all firmware images from the server.

`mesh models dfd instance-set <ElemIdx>`

Use the Firmware Distribution Server model instance on the specified element when using the other Firmware Distribution Server model commands.

- `ElemIdx`: The element on which to find the Firmware Distribution Server model instance to use.

`mesh models dfd instance-get-all`

Get a list of all Firmware Distribution Server model instances on the node.

DFU metadata The DFU metadata commands allow generating metadata that can be used by a Target node to check the firmware before accepting it. The commands are enabled through the `CONFIG_BT_MESH_DFU_METADATA` configuration option.

`mesh models dfu metadata comp-clear`

Clear the stored composition data to be used for the Target node.

`mesh models dfu metadata comp-add <CID> <ProductID> <VendorID> <Crpl> <Features>`

Create a header of the Composition Data Page 0.

- `CID`: Company identifier assigned by Bluetooth SIG.
- `ProductID`: Vendor-assigned product identifier.
- `VendorID`: Vendor-assigned version identifier.
- `Crpl`: The size of the replay protection list.
- `Features`: Features supported by the node in bit field format:
  - `0`: Relay.
  - `1`: Proxy.
  - `2`: Friend.
  - `3`: Low Power.

`mesh models dfu metadata comp-elem-add <Loc> <NumS> <NumV> {<SigMID>|<VndCID> <VndMID>}...`

Add element description of the Target node.

- `Loc`: Element location.
- `NumS`: Number of SIG models instantiated on the element.
- `NumV`: Number of vendor models instantiated on the element.
- `SigMID`: SIG Model ID.
- `VndCID`: Vendor model company identifier.
- `VndMID`: Vendor model identifier.

`mesh models dfu metadata comp-hash-get [<Key(16 hex)>]`

Generate a hash of the stored Composition Data to be used in metadata.

- `Key`: Optional 128-bit key to be used to generate the hash. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest.

`mesh models dfu metadata encode <Major> <Minor> <Rev> <BuildNum> <Size> <CoreType> <Hash> <Elems> [<UserData>]`

Encode metadata for the DFU.

- `Major`: Major version of the firmware.
- `Minor`: Minor version of the firmware.
- `Rev`: Revision number of the firmware.
- `BuildNum`: Build number.
- `Size`: Size of the signed bin file.
- `CoreType`: New firmware core type:
  - `1`: Application core.
  - `2`: Network core.
  - `4`: Applications specific BLOB.
- `Hash`: Hash of the composition data generated using `mesh models dfu metadata comp-hash-get` command.
- `Elems`: Number of elements on the new firmware.
- `UserData`: User data supplied with the metadata.

Segmentation and Reassembly (SAR) Configuration Client The SAR Configuration client is an optional mesh model that can be enabled through the `CONFIG_BT_MESH_SAR_CFG_CLI` configuration option. The SAR Configuration Client model is used to support the functionality of configuring the behavior of the lower transport layer of a node that supports the SAR Configuration Server model.

`mesh models sar tx-get`

Send SAR Configuration Transmitter Get message.

`mesh models sar tx-set <SegIntStep> <UniRetransCnt> <UniRetransWithoutProgCnt> <UniRetransIntStep> <UniRetransIntInc> <MultiRetransCnt> <MultiRetransInt>`

Send SAR Configuration Transmitter Set message.

- `SegIntStep`: SAR Segment Interval Step state.
- `UniRetransCnt`: SAR Unicast Retransmissions Count state.
- `UniRetransWithoutProgCnt`: SAR Unicast Retransmissions Without Progress Count state.
- `UniRetransIntStep`: SAR Unicast Retransmissions Interval Step state.
- `UniRetransIntInc`: SAR Unicast Retransmissions Interval Increment state.
- `MultiRetransCnt`: SAR Multicast Retransmissions Count state.
- `MultiRetransInt`: SAR Multicast Retransmissions Interval state.

`mesh models sar rx-get`

Send SAR Configuration Receiver Get message.

`mesh models sar rx-set <SegThresh> <AckDelayInc> <DiscardTimeout> <RxSegIntStep> <AckRetransCount>`

Send SAR Configuration Receiver Set message.

- `SegThresh`: SAR Segments Threshold state.
- `AckDelayInc`: SAR Acknowledgment Delay Increment state.
- `DiscardTimeout`: SAR Discard Timeout state.
- `RxSegIntStep`: SAR Receiver Segment Interval Step state.
- `AckRetransCount`: SAR Acknowledgment Retransmissions Count state.

Private Beacon Client The Private Beacon Client model is an optional mesh subsystem that can be enabled through the `CONFIG_BT_MESH_PRIV_BEACON_CLI` configuration option.

`mesh models prb priv-beacon-get`

Get the target's Private Beacon state. Possible values:

- `0x00`: The node doesn't broadcast Private beacons.
- `0x01`: The node broadcasts Private beacons.

`mesh models prb priv-beacon-set <Val(off, on)> <RandInt(10s steps)>`

Set the target's Private Beacon state.

- `Val`: Control Private Beacon state.
- `RandInt`: Random refresh interval (in 10-second steps), or 0 to keep current value.

`mesh models prb priv-gatt-proxy-get`

Get the target's Private GATT Proxy state. Possible values:

- `0x00`: The Private Proxy functionality is supported, but disabled.
- `0x01`: The Private Proxy functionality is enabled.
- `0x02`: The Private Proxy functionality is not supported.

`mesh models prb priv-gatt-proxy-set <Val(off, on)>`

Set the target's Private GATT Proxy state.

- `Val`: New Private GATT Proxy value:
  - `0x00`: Disable the Private Proxy functionality.
  - `0x01`: Enable the Private Proxy functionality.

`mesh models prb priv-node-id-get <NetKeyIdx>`

Get the target's Private Node Identity state. Possible values:

- `0x00`: The node does not adverstise with the Private Node Identity.
- `0x01`: The node advertises with the Private Node Identity.
- `0x02`: The node doesn't support advertising with the Private Node Identity.
- `NetKeyIdx`: Network index to get the Private Node Identity state of.

`mesh models prb priv-node-id-set <NetKeyIdx> <State>`

Set the target's Private Node Identity state.

- `NetKeyIdx`: Network index to set the Private Node Identity state of.
- `State`: New Private Node Identity value:
  - `0x00`: Stop advertising with the Private Node Identity.
  - `0x01`: Start advertising with the Private Node Identity.

Opcodes Aggregator Client The Opcodes Aggregator client is an optional Bluetooth Mesh model that can be enabled through the `CONFIG_BT_MESH_OP_AGG_CLI` configuration option. The Opcodes Aggregator Client model is used to support the functionality of dispatching a sequence of access layer messages to nodes supporting the Opcodes Aggregator Server model.

`mesh models opagg seq-start <ElemAddr>`

Start the Opcodes Aggregator Sequence message. This command initiates the context for aggregating messages and sets the destination address for next shell commands to `elem_addr`.

- `ElemAddr`: Element address that will process the aggregated opcodes.

`mesh models opagg seq-send`

Send the Opcodes Aggregator Sequence message. This command completes the procedure, sends the aggregated sequence message to the target node and clears the context.

`mesh models opagg seq-abort`

Abort the Opcodes Aggregator Sequence message. This command clears the Opcodes Aggregator Client context.

Remote Provisioning Client The Remote Provisioning Client is an optional Bluetooth Mesh model enabled through the `CONFIG_BT_MESH_RPR_CLI` configuration option. The Remote Provisioning Client model provides support for remote provisioning of devices into a mesh network by using the Remote Provisioning Server model.

This shell module can be used to trigger interaction between Remote Provisioning Clients and Remote Provisioning Servers on devices in a mesh network.

`mesh models rpr scan <Timeout(s)> [<UUID(1-16 hex)>]`

Start scanning for unprovisioned devices.

- `Timeout`: Scan timeout in seconds. Must be at least 1 second.
- `UUID`: Device UUID to scan for. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest. If omitted, all devices will be reported.

`mesh models rpr scan-ext <Timeout(s)> <UUID(1-16 hex)> [<ADType> ... ]`

Start the extended scanning for unprovisioned devices.

- `Timeout`: Scan timeout in seconds. Valid values from `BT_MESH_RPR_EXT_SCAN_TIME_MIN` to `BT_MESH_RPR_EXT_SCAN_TIME_MAX`.
- `UUID`: Device UUID to start extended scanning for. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest.
- `ADType`: List of AD types to include in the scan report. Must contain 1 to `CONFIG_BT_MESH_RPR_AD_TYPES_MAX` entries.

`mesh models rpr scan-srv [<ADType> ... ]`

Start the extended scanning for the Remote Provisioning Server.

- `ADType`: List of AD types to include in the scan report. Must contain 1 to `CONFIG_BT_MESH_RPR_AD_TYPES_MAX` entries.

`mesh models rpr scan-caps`

Get the scanning capabilities of the Remote Provisioning Server.

`mesh models rpr scan-get`

Get the current scanning state of the Remote Provisioning Server.

`mesh models rpr scan-stop`

Stop any ongoing scanning on the Remote Provisioning Server.

`mesh models rpr link-get`

Get the current link status of the Remote Provisioning Server.

`mesh models rpr link-close`

Close any open links on the Remote Provisioning Server.

`mesh models rpr provision-remote <UUID(1-16 hex)> <NetKeyIdx> <Addr>`

Provision a mesh node using the PB-Remote provisioning bearer.

- `UUID`: UUID of the unprovisioned node. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest.
- `NetKeyIdx`: Network Key Index to give to the unprovisioned node.
- `Addr`: Address to assign to remote device. If `addr` is 0, the lowest available address will be chosen.

`mesh models rpr reprovision-remote <Addr> [<CompChanged(false, true)>]`

Reprovision a mesh node using the PB-Remote provisioning bearer.

- `Addr`: Address to assign to remote device. If `addr` is 0, the lowest available address will be chosen.
- `CompChanged`: The Target node has indicated that its Composition Data has changed. Defaults to false.

`mesh models rpr instance-set <ElemIdx>`

Use the Remote Provisioning Client model instance on the specified element when using the other Remote Provisioning Client model commands.

- `ElemIdx`: The element on which to find the Remote Provisioning Client model instance to use.

`mesh models rpr instance-get-all`

Get a list of all Remote Provisioning Client model instances on the node.

Large Composition Data Client The Large Composition Data Client is an optional Bluetooth Mesh model enabled through the `CONFIG_BT_MESH_LARGE_COMP_DATA_CLI` configuration option. The Large Composition Data Client model is used to support the functionality of reading pages of Composition Data that do not fit in a Config Composition Data Status message, and reading the metadata of the model instances.

`mesh models lcd large-comp-data-get <Page> <Offset>`

Send the Large Composition Data Get message to query a portion of the Composition Data state of a node.

- `Page`: Page number of the Composition Data.
- `Offset`: Offset within the page.

`mesh models lcd models-metadata-get <Page> <Offset>`

Send the Models Metadata Get message to query a portion of a page of the Models Metadata state.

- `Page`: Page number of the Models Metadata.
- `Offset`: Offset within the page.

Configuration database The Configuration database is an optional mesh subsystem that can be enabled through the `CONFIG_BT_MESH_CDB` configuration option. The Configuration database is only available on provisioner devices, and allows them to store all information about the mesh network. To avoid conflicts, there should only be one mesh node in the network with the Configuration database enabled. This node is the Configurator, and is responsible for adding new nodes to the network and configuring them.

`mesh cdb create [NetKey(1-16 hex)]`

Create a Configuration database.

- `NetKey`: Optional network key value of the primary network key (NetKeyIndex=0). Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest. Defaults to the default key value if omitted.

`mesh cdb clear`

Clear all data from the Configuration database.

`mesh cdb show`

Show all data in the Configuration database.

`mesh cdb node-add <UUID(1-16 hex)> <Addr> <ElemCnt> <NetKeyIdx> [DevKey(1-16 hex)]`

Manually add a mesh node to the configuration database. Note that devices provisioned with `mesh provision` and `mesh provision-adv` will be added automatically if the Configuration Database is enabled and created.

- `UUID`: 128-bit hexadecimal UUID of the node. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest.
- `Addr`: Unicast address of the node, or 0 to automatically choose the lowest available address.
- `ElemCnt`: Number of elements on the node.
- `NetKeyIdx`: The network key the node was provisioned with.
- `DevKey`: Optional 128-bit device key value for the device. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest. If omitted, a random value will be generated.

`mesh cdb node-del <Addr>`

Delete a mesh node from the Configuration database. If possible, the node should be reset with `mesh reset` before it is deleted from the Configuration database, to avoid unexpected behavior and uncontrolled access to the network.

- `Addr` Address of the node to delete.

`mesh cdb subnet-add <NetKeyIdx> [<NetKey(1-16 hex)>]`

Add a network key to the Configuration database. The network key can later be passed to mesh nodes in the network. Note that adding a key to the Configuration database does not automatically add it to the local node's list of known network keys.

- `NetKeyIdx`: Key index of the network key to add.
- `NetKey`: Optional 128-bit network key value. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest. If omitted, a random value will be generated.

`mesh cdb subnet-del <NetKeyIdx>`

Delete a network key from the Configuration database.

- `NetKeyIdx`: Key index of the network key to delete.

`mesh cdb app-key-add <NetKeyIdx> <AppKeyIdx> [<AppKey(1-16 hex)>]`

Add an application key to the Configuration database. The application key can later be passed to mesh nodes in the network. Note that adding a key to the Configuration database does not automatically add it to the local node's list of known application keys.

- `NetKeyIdx`: Network key index the application key is bound to.
- `AppKeyIdx`: Key index of the application key to add.
- `AppKey`: Optional 128-bit application key value. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest. If omitted, a random value will be generated.

`mesh cdb app-key-del <AppKeyIdx>`

Delete an application key from the Configuration database.

- `AppKeyIdx`: Key index of the application key to delete.

On-Demand Private GATT Proxy Client The On-Demand Private GATT Proxy Client model is an optional mesh subsystem that can be enabled through the `CONFIG_BT_MESH_OD_PRIV_PROXY_CLI` configuration option.

`mesh models od_priv_proxy od-priv-gatt-proxy [Dur(s)]`

Set the On-Demand Private GATT Proxy state on active target, or fetch the value of this state from it.

- `Dur`: If given, set the state of On-Demand Private GATT Proxy to this value in seconds. Fetch this value otherwise.

Solicitation PDU RPL Client The Solicitation PDU RPL Client model is an optional mesh subsystem that can be enabled through the `CONFIG_BT_MESH_SOL_PDU_RPL_CLI` configuration option.

`mesh models sol_pdu_rpl sol-pdu-rpl-clear <RngStart> <Ackd> [RngLen]`

Clear active target's solicitation replay protection list (SRPL) in given range of solicitation source (SSRC) addresses.

- `RngStart`: Start address of the SSRC range.
- `Ackd`: This argument decides on whether an acknowledged or unacknowledged message will be sent.

- `RngLen`: Range length for the SSRC addresses to be cleared from the solicitiation RPL list. This parameter is optional; if absent, only a single SSRC address will be cleared.

Frame statistic

`mesh stat get`

Get the frame statistic. The command prints numbers of received frames, as well as numbers of planned and succeeded transmission attempts.

`mesh stat clear`

Clear all statistics collected before.

Core Host and drivers

Logical Link Control and Adaptation Protocol (L2CAP) L2CAP layer enables connection-oriented channels which can be enable with the configuration option: `CONFIG_BT_L2CAP_DYNAMIC_CHANNEL`. This channels support segmentation and reassembly transparently, they also support credit based flow control making it suitable for data streams.

Channels instances are represented by the *bt_l2cap_chan* struct which contains the callbacks in the *bt_l2cap_chan_ops* struct to inform when the channel has been connected, disconnected or when the encryption has changed. In addition to that it also contains the `recv` callback which is called whenever an incoming data has been received. Data received this way can be marked as processed by returning 0 or using *bt_l2cap_chan_recv_complete()* API if processing is asynchronous.

Note

The `recv` callback is called directly from RX Thread thus it is not recommended to block for long periods of time.

For sending data the *bt_l2cap_chan_send()* API can be used noting that it may block if no credits are available, and resuming as soon as more credits are available.

Servers can be registered using *bt_l2cap_server_register()* API passing the *bt_l2cap_server* struct which informs what `psm` it should listen to, the required security level `sec_level`, and the callback `accept` which is called to authorize incoming connection requests and allocate channel instances.

Client channels can be initiated with use of *bt_l2cap_chan_connect()* API and can be disconnected with the *bt_l2cap_chan_disconnect()* API. Note that the later can also disconnect channel instances created by servers.

API Reference

group `bt_l2cap`

L2CAP.

Defines

BT_L2CAP_HDR_SIZE

L2CAP PDU header size, used for buffer size calculations.

BT_L2CAP_TX_MTU

Maximum Transmission Unit (MTU) for an outgoing L2CAP PDU.

BT_L2CAP_RX_MTU

Maximum Transmission Unit (MTU) for an incoming L2CAP PDU.

BT_L2CAP_BUF_SIZE(mtu)

Helper to calculate needed buffer size for L2CAP PDUs.

Useful for creating buffer pools.

Parameters

- mtu – Needed L2CAP PDU MTU.

Returns

Needed buffer size to match the requested L2CAP PDU MTU.

BT_L2CAP_SDU_HDR_SIZE

L2CAP SDU header size, used for buffer size calculations.

BT_L2CAP_SDU_TX_MTU

Maximum Transmission Unit for an unsegmented outgoing L2CAP SDU.

The Maximum Transmission Unit for an outgoing L2CAP SDU when sent without segmentation, i.e. a single L2CAP SDU will fit inside a single L2CAP PDU.

The MTU for outgoing L2CAP SDUs with segmentation is defined by the size of the application buffer pool.

BT_L2CAP_SDU_RX_MTU

Maximum Transmission Unit for an unsegmented incoming L2CAP SDU.

The Maximum Transmission Unit for an incoming L2CAP SDU when sent without segmentation, i.e. a single L2CAP SDU will fit inside a single L2CAP PDU.

The MTU for incoming L2CAP SDUs with segmentation is defined by the size of the application buffer pool. The application will have to define an alloc_buf callback for the channel in order to support receiving segmented L2CAP SDUs.

BT_L2CAP_SDU_BUF_SIZE(mtu)

Helper to calculate needed buffer size for L2CAP SDUs.

Useful for creating buffer pools.

Parameters

- mtu – Required BT_L2CAP_*_SDU.

Returns

Needed buffer size to match the requested L2CAP SDU MTU.

BT_L2CAP_LE_CHAN(_ch)

Helper macro getting container object of type *bt_l2cap_le_chan* address having the same container chan member address as object in question.

Parameters

- **_ch** – Address of object of *bt_l2cap_chan* type

Returns
Address of in memory *bt_l2cap_le_chan* object type containing the address of in question object.

BT_L2CAP_CHAN_SEND_RESERVE

Headroom needed for outgoing L2CAP PDUs.

BT_L2CAP_SDU_CHAN_SEND_RESERVE

Headroom needed for outgoing L2CAP SDUs.

Typedefs

typedef void (*bt_l2cap_chan_destroy_t)(struct *bt_l2cap_chan* *chan)

Channel destroy callback.

Param chan
Channel object.

typedef enum *bt_l2cap_chan_state* bt_l2cap_chan_state_t

Life-span states of L2CAP CoC channel.

Used only by internal APIs dealing with setting channel to proper state depending on operational context.

A channel enters the *BT_L2CAP_CONNECTING* state upon *bt_l2cap_chan_connect*, *bt_l2cap_ecred_chan_connect* or upon returning from *bt_l2cap_server::accept*.

When a channel leaves the *BT_L2CAP_CONNECTING* state, *bt_l2cap_chan_ops::connected* is called.

typedef enum *bt_l2cap_chan_status* bt_l2cap_chan_status_t

Status of L2CAP channel.

Enums

enum bt_l2cap_chan_state

Life-span states of L2CAP CoC channel.

Used only by internal APIs dealing with setting channel to proper state depending on operational context.

A channel enters the *BT_L2CAP_CONNECTING* state upon *bt_l2cap_chan_connect*, *bt_l2cap_ecred_chan_connect* or upon returning from *bt_l2cap_server::accept*.

When a channel leaves the *BT_L2CAP_CONNECTING* state, *bt_l2cap_chan_ops::connected* is called.

Values:

enumerator BT_L2CAP_DISCONNECTED

Channel disconnected.

enumerator BT_L2CAP_CONNECTING
Channel in connecting state.

enumerator BT_L2CAP_CONFIG
Channel in config state, BR/EDR specific.

enumerator BT_L2CAP_CONNECTED
Channel ready for upper layer traffic on it.

enumerator BT_L2CAP_DISCONNECTING
Channel in disconnecting state.

enum bt_l2cap_chan_status
Status of L2CAP channel.

Values:

enumerator BT_L2CAP_STATUS_OUT
Channel can send at least one PDU.

enumerator BT_L2CAP_STATUS_SHUTDOWN
Channel shutdown status.

```
Once this status is notified it means the channel will no longer be
able to transmit or receive data.
```

enumerator BT_L2CAP_STATUS_ENCRYPT_PENDING
Channel encryption pending status.

enumerator BT_L2CAP_NUM_STATUS

Functions

int bt_l2cap_server_register(struct *bt_l2cap_server* *server)
Register L2CAP server.

Register L2CAP server for a PSM, each new connection is authorized using the *accept()* callback which in case of success shall allocate the channel structure to be used by the new connection.

For fixed, SIG-assigned PSMs (in the range 0x0001-0x007f) the PSM should be assigned to server->psm before calling this API. For dynamic PSMs (in the range 0x0080-0x00ff) server->psm may be pre-set to a given value (this is however not recommended) or be left as 0, in which case upon return a newly allocated value will have been assigned to it. For dynamically allocated values the expectation is that it's exposed through a GATT service, and that's how L2CAP clients discover how to connect to the server.

Parameters

- server – Server structure.

Returns
0 in case of success or negative value in case of error.

int **bt_l2cap_br_server_register**(struct *bt_l2cap_server* *server)

Register L2CAP server on BR/EDR oriented connection.

Register L2CAP server for a PSM, each new connection is authorized using the *accept()* callback which in case of success shall allocate the channel structure to be used by the new connection.

Parameters

- **server** – Server structure.

Returns
0 in case of success or negative value in case of error.

int **bt_l2cap_ecred_chan_connect**(struct bt_conn *conn, struct *bt_l2cap_chan* **chans, uint16_t psm)

Connect Enhanced Credit Based L2CAP channels.

Connect up to 5 L2CAP channels by PSM, once the connection is completed each channel connected() callback will be called. If the connection is rejected disconnected() callback is called instead.

Parameters

- **conn** – Connection object.
- **chans** – Array of channel objects.
- **psm** – Channel PSM to connect to.

Returns
0 in case of success or negative value in case of error.

int **bt_l2cap_ecred_chan_reconfigure**(struct *bt_l2cap_chan* **chans, uint16_t mtu)

Reconfigure Enhanced Credit Based L2CAP channels.

Reconfigure up to 5 L2CAP channels. Channels must be from the same bt_conn. Once reconfiguration is completed each channel reconfigured() callback will be called. MTU cannot be decreased on any of provided channels.

Parameters

- **chans** – Array of channel objects. Null-terminated. Elements after the first 5 are silently ignored.
- **mtu** – Channel MTU to reconfigure to.

Returns
0 in case of success or negative value in case of error.

int **bt_l2cap_chan_connect**(struct bt_conn *conn, struct *bt_l2cap_chan* *chan, uint16_t psm)

Connect L2CAP channel.

Connect L2CAP channel by PSM, once the connection is completed channel connected() callback will be called. If the connection is rejected disconnected() callback is called instead. Channel object passed (over an address of it) as second parameter shouldn't be instantiated in application as standalone. Instead of, application should create transport dedicated L2CAP objects, i.e. type of *bt_l2cap_le_chan* for LE and/or type of *bt_l2cap_br_chan* for BR/EDR. Then pass to this API the location (address) of *bt_l2cap_chan* type object which is a member of both transport dedicated objects.

Parameters

- **conn** – Connection object.
- **chan** – Channel object.

- **psm** – Channel PSM to connect to.

Returns
0 in case of success or negative value in case of error.

int **bt_l2cap_chan_disconnect**(struct *bt_l2cap_chan* *chan)

Disconnect L2CAP channel.

Disconnect L2CAP channel, if the connection is pending it will be canceled and as a result the channel disconnected() callback is called. Regarding to input parameter, to get details see reference description to *bt_l2cap_chan_connect()* API above.

Parameters

- **chan** – Channel object.

Returns
0 in case of success or negative value in case of error.

int **bt_l2cap_chan_send**(struct *bt_l2cap_chan* *chan, struct *net_buf* *buf)

Send data to L2CAP channel.

Send data from buffer to the channel. If credits are not available, buf will be queued and sent as and when credits are received from peer. Regarding to first input parameter, to get details see reference description to *bt_l2cap_chan_connect()* API above.

Network buffer fragments (ie `buf->frags`) are not supported.

When sending L2CAP data over an BR/EDR connection the application is sending L2CAP PDUs. The application is required to have reserved *BT_L2CAP_CHAN_SEND_RESERVE* bytes in the buffer before sending. The application should use the *BT_L2CAP_BUF_SIZE()* helper to correctly size the buffers for the for the outgoing buffer pool.

When sending L2CAP data over an LE connection the application is sending L2CAP SDUs. The application shall reserve *BT_L2CAP_SDU_CHAN_SEND_RESERVE* bytes in the buffer before sending.

The application can use the *BT_L2CAP_SDU_BUF_SIZE()* helper to correctly size the buffer to account for the reserved headroom.

When segmenting an L2CAP SDU into L2CAP PDUs the stack will first attempt to allocate buffers from the channel's `alloc_seg` callback and will fallback on the stack's global buffer pool (sized `CONFIG_BT_L2CAP_TX_BUF_COUNT`).

Note

Buffer ownership is transferred to the stack in case of success, in case of an error the caller retains the ownership of the buffer.

Returns
0 in case of success or negative value in case of error.

Returns
-EINVAL if `buf` or `chan` is NULL.

Returns
-EINVAL if `chan` is not either BR/EDR or LE credit-based.

Returns
-EINVAL if buffer doesn't have enough bytes reserved to fit header.

Returns
-EINVAL if buffer's reference counter != 1

Returns
-EMSGSIZE if buf is larger than chan's MTU.

Returns
-ENOTCONN if underlying conn is disconnected.

Returns
-ESHUTDOWN if L2CAP channel is disconnected.

Returns
-other (from lower layers) if chan is BR/EDR.

int **bt_l2cap_chan_give_credits**(struct *bt_l2cap_chan* *chan, uint16_t additional_credits)

Give credits to the remote.

Only available for channels using *bt_l2cap_chan_ops::seg_recv*. CONFIG_BT_L2CAP_SEG_RECV must be enabled to make this function available.

Each credit given allows the peer to send one segment.

This function depends on a valid chan object. Make sure to default-initialize or memset chan when allocating or reusing it for new connections.

Adding zero credits is not allowed.

Credits can be given before entering the *BT_L2CAP_CONNECTING* state. Doing so will adjust the 'initial credits' sent in the connection PDU.

Must not be called while the channel is in *BT_L2CAP_CONNECTING* state.

Returns
0 in case of success or negative value in case of error.

int **bt_l2cap_chan_recv_complete**(struct *bt_l2cap_chan* *chan, struct *net_buf* *buf)

Complete receiving L2CAP channel data.

Complete the reception of incoming data. This shall only be called if the channel recv callback has returned -EINPROGRESS to process some incoming data. The buffer shall contain the original user_data as that is used for storing the credits/segments used by the packet.

Parameters

- **chan** – Channel object.
- **buf** – Buffer containing the data.

Returns
0 in case of success or negative value in case of error.

struct **bt_l2cap_chan**

#include <*l2cap.h*> L2CAP Channel structure.

Public Members

struct bt_conn ***conn**

Channel connection reference.

const struct *bt_l2cap_chan_ops* ***ops**

Channel operations reference.

struct **bt_l2cap_le_endpoint**

#include <l2cap.h> LE L2CAP Endpoint structure.

Public Members

uint16_t **cid**

Endpoint Channel Identifier (CID)

uint16_t **mtu**

Endpoint Maximum Transmission Unit.

uint16_t **mps**

Endpoint Maximum PDU payload Size.

atomic_t **credits**

Endpoint credits.

struct **bt_l2cap_le_chan**

#include <l2cap.h> LE L2CAP Channel structure.

Public Members

struct *bt_l2cap_chan* **chan**

Common L2CAP channel reference object.

struct *bt_l2cap_le_endpoint* **rx**

Channel Receiving Endpoint.

If the application has set an alloc_buf channel callback for the channel to support receiving segmented L2CAP SDUs the application should initialize the MTU of the Receiving Endpoint. Otherwise the MTU of the receiving endpoint will be initialized to *BT_L2CAP_SDU_RX_MTU* by the stack.

This is the source of the MTU, MPS and credit values when sending L2CAP_LE_CREDIT_BASED_CONNECTION_REQ/RSP and L2CAP_CONFIGURATION_REQ.

uint16_t **pending_rx_mtu**

Pending RX MTU on ECFC reconfigure, used internally by stack.

struct *bt_l2cap_le_endpoint* **tx**

Channel Transmission Endpoint.

This is an image of the remote's rx.

The MTU and MPS is controlled by the remote by L2CAP_LE_CREDIT_BASED_CONNECTION_REQ/RSP or L2CAP_CONFIGURATION_REQ.

struct k_fifo **tx_queue**

Channel Transmission queue (for SDUs)

struct **bt_l2cap_br_endpoint**

#include <*l2cap.h*> BREDR L2CAP Endpoint structure.

Public Members

uint16_t **cid**

Endpoint Channel Identifier (CID)

uint16_t **mtu**

Endpoint Maximum Transmission Unit.

struct **bt_l2cap_br_chan**

#include <*l2cap.h*> BREDR L2CAP Channel structure.

Public Members

struct *bt_l2cap_chan* **chan**

Common L2CAP channel reference object.

struct *bt_l2cap_br_endpoint* **rx**

Channel Receiving Endpoint.

struct *bt_l2cap_br_endpoint* **tx**

Channel Transmission Endpoint.

uint16_t **psm**

Remote PSM to be connected.

uint8_t **ident**

Helps match request context during CoC.

struct **bt_l2cap_chan_ops**

#include <*l2cap.h*> L2CAP Channel operations structure.

Public Members

void (***connected**)(struct *bt_l2cap_chan* *chan)

Channel connected callback.

If this callback is provided it will be called whenever the connection completes.

Param chan

The channel that has been connected

void (*disconnected)(struct *bt_l2cap_chan* *chan)

Channel disconnected callback.

If this callback is provided it will be called whenever the channel is disconnected, including when a connection gets rejected.

Param chan
The channel that has been Disconnected

void (*encrypt_change)(struct *bt_l2cap_chan* *chan, uint8_t hci_status)

Channel encrypt_change callback.

If this callback is provided it will be called whenever the security level changed (indirectly link encryption done) or authentication procedure fails. In both cases security initiator and responder got the final status (HCI status) passed by related to encryption and authentication events from local host's controller.

Param chan
The channel which has made encryption status changed.

Param status
HCI status of performed security procedure caused by channel security requirements. The value is populated by HCI layer and set to 0 when success and to non-zero (reference to HCI Error Codes) when security/authentication failed.

struct *net_buf* *(*alloc_seg)(struct *bt_l2cap_chan* *chan)

Channel alloc_seg callback.

If this callback is provided the channel will use it to allocate buffers to store segments. This avoids wasting big SDU buffers with potentially much smaller PDUs. If this callback is supplied, it must return a valid buffer.

Param chan
The channel requesting a buffer.

Return
Allocated buffer.

struct *net_buf* *(*alloc_buf)(struct *bt_l2cap_chan* *chan)

Channel alloc_buf callback.

If this callback is provided the channel will use it to allocate buffers to store incoming data. Channels that requires segmentation must set this callback. If the application has not set a callback the L2CAP SDU MTU will be truncated to *BT_L2CAP_SDU_RX_MTU*.

Param chan
The channel requesting a buffer.

Return
Allocated buffer.

int (*recv)(struct *bt_l2cap_chan* *chan, struct *net_buf* *buf)

Channel recv callback.

Param chan
The channel receiving data.

Param buf
Buffer containing incoming data.

Return
0 in case of success or negative value in case of error.

Return
-EINPROGRESS in case where user has to confirm once the data has been processed by calling *bt_l2cap_chan_recv_complete* passing back the buffer

received with its original user_data which contains the number of segments/credits used by the packet.

void (*sent)(struct *bt_l2cap_chan* *chan)

Channel sent callback.

This callback will be called once the controller marks the SDU as completed. When the controller does so is implementation dependent. It could be after the SDU is enqueued for transmission, or after it is sent on air.

Param chan
The channel which has sent data.

void (*status)(struct *bt_l2cap_chan* *chan, atomic_t *status)

Channel status callback.

If this callback is provided it will be called whenever the channel status changes.

Param chan
The channel which status changed

Param status
The channel status

void (*reconfigured)(struct *bt_l2cap_chan* *chan)

Channel reconfigured callback.

If this callback is provided it will be called whenever peer or local device requested reconfiguration. Application may check updated MTU and MPS values by inspecting chan->le endpoints.

Param chan
The channel which was reconfigured

void (*seg_recv)(struct *bt_l2cap_chan* *chan, size_t sdu_len, off_t seg_offset, struct *net_buf_simple* *seg)

Handle L2CAP segments directly.

This is an alternative to *bt_l2cap_chan_ops::recv*. They cannot be used together.

This is called immediately for each received segment.

Unlike with *bt_l2cap_chan_ops::recv*, flow control is explicit. Each time this handler is invoked, the remote has permanently used up one credit. Use *bt_l2cap_chan_give_credits* to give credits.

The start of an SDU is marked by `seg_offset == 0`. The end of an SDU is marked by `seg_offset + seg->len == sdu_len`.

The stack guarantees that:
- The sender had the credit.
- The SDU length does not exceed MTU.
- The segment length does not exceed MPS.

Additionally, the L2CAP protocol is such that:
- Segments come in order.
- SDUs cannot be interleaved or aborted halfway.

Note

With this alternative API, the application is responsible for setting the RX MTU and MPS. The MPS must not exceed *BT_L2CAP_RX_MTU*.

Param chan
The receiving channel.

Param sdu_len
Byte length of the SDU this segment is part of.
Param seg_offset
The byte offset of this segment in the SDU.
Param seg
The segment payload.

struct **bt_l2cap_server**

#include <*l2cap.h*> L2CAP Server structure.

Public Members

uint16_t **psm**

Server PSM.

Possible values: 0 A dynamic value will be auto-allocated when *bt_l2cap_server_register()* is called.

0x0001-0x007f Standard, Bluetooth SIG-assigned fixed values.

0x0080-0x00ff Dynamically allocated. May be pre-set by the application before server registration (not recommended however), or auto-allocated by the stack if the app gave 0 as the value.

bt_security_t **sec_level**

Required minimum security level.

int (***accept**)(struct bt_conn *conn, struct *bt_l2cap_server* *server, struct *bt_l2cap_chan* **chan)

Server accept callback.

This callback is called whenever a new incoming connection requires authorization.
Param conn
The connection that is requesting authorization
Param server
Pointer to the server structure this callback relates to
Param chan
Pointer to received the allocated channel
Return
0 in case of success or negative value in case of error.
Return
-ENOMEM if no available space for new channel.
Return
-EACCES if application did not authorize the connection.
Return
-EPERM if encryption key size is too short.

Connection Management The Zephyr Bluetooth stack uses an abstraction called `bt_conn` to represent connections to other devices. The internals of this struct are not exposed to the application, but a limited amount of information (such as the remote address) can be acquired using the *bt_conn_get_info()* API. Connection objects are reference counted, and the application is expected to use the *bt_conn_ref()* API whenever storing a connection pointer for a longer period of time, since this ensures that the object remains valid (even if the connection would get disconnected). Similarly the *bt_conn_unref()* API is to be used when releasing a reference to a connection.

An application may track connections by registering a *bt_conn_cb* struct using the *bt_conn_cb_register()* or *BT_CONN_CB_DEFINE* APIs. This struct lets the application define callbacks for connection & disconnection events, as well as other events related to a connection such as a change in the security level or the connection parameters. When acting as a central the application will also get hold of the connection object through the return value of the *bt_conn_le_create()* API.

API Reference

group `bt_conn`

Connection management.

Defines

`BT_LE_CONN_PARAM_INIT`(int_min, int_max, lat, to)

Initialize connection parameters.

Parameters

- `int_min` – Minimum Connection Interval (N * 1.25 ms)
- `int_max` – Maximum Connection Interval (N * 1.25 ms)
- `lat` – Connection Latency
- `to` – Supervision Timeout (N * 10 ms)

`BT_LE_CONN_PARAM`(int_min, int_max, lat, to)

Helper to declare connection parameters inline.

Parameters

- `int_min` – Minimum Connection Interval (N * 1.25 ms)
- `int_max` – Maximum Connection Interval (N * 1.25 ms)
- `lat` – Connection Latency
- `to` – Supervision Timeout (N * 10 ms)

`BT_LE_CONN_PARAM_DEFAULT`

Default LE connection parameters: Connection Interval: 30-50 ms Latency: 0 Timeout: 4 s.

`BT_CONN_LE_PHY_PARAM_INIT`(_pref_tx_phy, _pref_rx_phy)

Initialize PHY parameters.

Parameters

- `_pref_tx_phy` – Bitmask of preferred transmit PHYs.
- `_pref_rx_phy` – Bitmask of preferred receive PHYs.

`BT_CONN_LE_PHY_PARAM`(_pref_tx_phy, _pref_rx_phy)

Helper to declare PHY parameters inline.

Parameters

- `_pref_tx_phy` – Bitmask of preferred transmit PHYs.
- `_pref_rx_phy` – Bitmask of preferred receive PHYs.

BT_CONN_LE_PHY_PARAM_1M
: Only LE 1M PHY.

BT_CONN_LE_PHY_PARAM_2M
: Only LE 2M PHY.

BT_CONN_LE_PHY_PARAM_CODED
: Only LE Coded PHY.

BT_CONN_LE_PHY_PARAM_ALL
: All LE PHYs.

BT_CONN_LE_DATA_LEN_PARAM_INIT(_tx_max_len, _tx_max_time)
: Initialize transmit data length parameters.

  Parameters

  - **_tx_max_len** – Maximum Link Layer transmission payload size in bytes.
  - **_tx_max_time** – Maximum Link Layer transmission payload time in us.

BT_CONN_LE_DATA_LEN_PARAM(_tx_max_len, _tx_max_time)
: Helper to declare transmit data length parameters inline.

  Parameters

  - **_tx_max_len** – Maximum Link Layer transmission payload size in bytes.
  - **_tx_max_time** – Maximum Link Layer transmission payload time in us.

BT_LE_DATA_LEN_PARAM_DEFAULT
: Default LE data length parameters.

BT_LE_DATA_LEN_PARAM_MAX
: Maximum LE data length parameters.

BT_CONN_INTERVAL_TO_MS(interval)
: Convert connection interval to milliseconds.

  Multiply by 1.25 to get milliseconds.

  Note that this may be inaccurate, as something like 7.5 ms cannot be accurately presented with integers.

BT_CONN_INTERVAL_TO_US(interval)
: Convert connection interval to microseconds.

  Multiply by 1250 to get microseconds.

BT_CONN_LE_CREATE_PARAM_INIT(_options, _interval, _window)
: Initialize create connection parameters.

  Parameters

  - **_options** – Create connection options.
  - **_interval** – Create connection scan interval (N * 0.625 ms).
  - **_window** – Create connection scan window (N * 0.625 ms).

BT_CONN_LE_CREATE_PARAM(_options, _interval, _window)

Helper to declare create connection parameters inline.

Parameters

- **_options** – Create connection options.
- **_interval** – Create connection scan interval (N * 0.625 ms).
- **_window** – Create connection scan window (N * 0.625 ms).

BT_CONN_LE_CREATE_CONN

Default LE create connection parameters.

Scan continuously by setting scan interval equal to scan window.

BT_CONN_LE_CREATE_CONN_AUTO

Default LE create connection using filter accept list parameters.

Scan window: 30 ms. Scan interval: 60 ms.

BT_CONN_CB_DEFINE(_name)

Register a callback structure for connection events.

Parameters

- **_name** – Name of callback structure.

BT_PASSKEY_INVALID

Special passkey value that can be used to disable a previously set fixed passkey.

BT_BR_CONN_PARAM_INIT(role_switch)

Initialize BR/EDR connection parameters.

Parameters

- **role_switch** – True if role switch is allowed

BT_BR_CONN_PARAM(role_switch)

Helper to declare BR/EDR connection parameters inline.

Parameters

- **role_switch** – True if role switch is allowed

BT_BR_CONN_PARAM_DEFAULT

Default BR/EDR connection parameters: Role switch allowed.

Enums

Connection PHY options.

Values:

enumerator BT_CONN_LE_PHY_OPT_NONE = 0

Convenience value when no options are specified.

enumerator BT_CONN_LE_PHY_OPT_CODED_S2 = *BIT*(0)

LE Coded using S=2 coding preferred when transmitting.

enumerator **BT_CONN_LE_PHY_OPT_CODED_S8** = *BIT*(1)

LE Coded using S=8 coding preferred when transmitting.

enum **bt_conn_type**

Connection Type.

Values:

enumerator **BT_CONN_TYPE_LE** = *BIT*(0)

LE Connection Type.

enumerator **BT_CONN_TYPE_BR** = *BIT*(1)

BR/EDR Connection Type.

enumerator **BT_CONN_TYPE_SCO** = *BIT*(2)

SCO Connection Type.

enumerator **BT_CONN_TYPE_ISO** = *BIT*(3)

ISO Connection Type.

enumerator **BT_CONN_TYPE_ALL** = *BT_CONN_TYPE_LE* | *BT_CONN_TYPE_BR* | *BT_CONN_TYPE_SCO* | *BT_CONN_TYPE_ISO*

All Connection Type.

Values:

enumerator **BT_CONN_ROLE_CENTRAL** = 0

enumerator **BT_CONN_ROLE_PERIPHERAL** = 1

enum **bt_conn_state**

Values:

enumerator **BT_CONN_STATE_DISCONNECTED**

Channel disconnected.

enumerator **BT_CONN_STATE_CONNECTING**

Channel in connecting state.

enumerator **BT_CONN_STATE_CONNECTED**

Channel connected and ready for upper layer traffic on it.

enumerator **BT_CONN_STATE_DISCONNECTING**

Channel in disconnecting state.

enum **bt_security_t**

Security level.

Values:

enumerator BT_SECURITY_L0
Level 0: Only for BR/EDR special cases, like SDP.

enumerator BT_SECURITY_L1
Level 1: No encryption and no authentication.

enumerator BT_SECURITY_L2
Level 2: Encryption and no authentication (no MITM).

enumerator BT_SECURITY_L3
Level 3: Encryption and authentication (MITM).

enumerator BT_SECURITY_L4
Level 4: Authenticated Secure Connections and 128-bit key.

enumerator BT_SECURITY_FORCE_PAIR = *BIT*(7)
Bit to force new pairing procedure, bit-wise OR with requested security level.

enum bt_security_flag
Security Info Flags.
Values:

enumerator BT_SECURITY_FLAG_SC = *BIT*(0)
Paired with Secure Connections.

enumerator BT_SECURITY_FLAG_OOB = *BIT*(1)
Paired with Out of Band method.

enum bt_conn_le_tx_power_phy
Values:

enumerator BT_CONN_LE_TX_POWER_PHY_NONE
Convenience macro for when no PHY is set.

enumerator BT_CONN_LE_TX_POWER_PHY_1M
LE 1M PHY.

enumerator BT_CONN_LE_TX_POWER_PHY_2M
LE 2M PHY.

enumerator BT_CONN_LE_TX_POWER_PHY_CODED_S8
LE Coded PHY using S=8 coding.

enumerator BT_CONN_LE_TX_POWER_PHY_CODED_S2
LE Coded PHY using S=2 coding.

enum `bt_conn_le_path_loss_zone`

Path Loss zone that has been entered.

The path loss zone that has been entered in the most recent LE Path Loss Monitoring Threshold Change event as documented in Core Spec. Version 5.4 Vol.4, Part E, 7.7.65.32.

> **Note**
>
> BT_CONN_LE_PATH_LOSS_ZONE_UNAVAILABLE has been added to notify when path loss becomes unavailable.

Values:

enumerator `BT_CONN_LE_PATH_LOSS_ZONE_ENTERED_LOW`

Low path loss zone entered.

enumerator `BT_CONN_LE_PATH_LOSS_ZONE_ENTERED_MIDDLE`

Middle path loss zone entered.

enumerator `BT_CONN_LE_PATH_LOSS_ZONE_ENTERED_HIGH`

High path loss zone entered.

enumerator `BT_CONN_LE_PATH_LOSS_ZONE_UNAVAILABLE`

Path loss has become unavailable.

enum `bt_conn_auth_keypress`

Passkey Keypress Notification type.

The numeric values are the same as in the Core specification for Pairing Keypress Notification PDU.

Values:

enumerator `BT_CONN_AUTH_KEYPRESS_ENTRY_STARTED` = 0x00

enumerator `BT_CONN_AUTH_KEYPRESS_DIGIT_ENTERED` = 0x01

enumerator `BT_CONN_AUTH_KEYPRESS_DIGIT_ERASED` = 0x02

enumerator `BT_CONN_AUTH_KEYPRESS_CLEARED` = 0x03

enumerator `BT_CONN_AUTH_KEYPRESS_ENTRY_COMPLETED` = 0x04

Values:

enumerator `BT_CONN_LE_OPT_NONE` = 0

Convenience value when no options are specified.

enumerator **BT_CONN_LE_OPT_CODED** = *BIT*(0)

Enable LE Coded PHY.

```
Enable scanning on the LE Coded PHY.
```

enumerator **BT_CONN_LE_OPT_NO_1M** = *BIT*(1)

Disable LE 1M PHY.

```
Disable scanning on the LE 1M PHY.

@note Requires @ref BT_CONN_LE_OPT_CODED.
```

enum **bt_security_err**

Values:

enumerator **BT_SECURITY_ERR_SUCCESS**

Security procedure successful.

enumerator **BT_SECURITY_ERR_AUTH_FAIL**

Authentication failed.

enumerator **BT_SECURITY_ERR_PIN_OR_KEY_MISSING**

PIN or encryption key is missing.

enumerator **BT_SECURITY_ERR_OOB_NOT_AVAILABLE**

OOB data is not available.

enumerator **BT_SECURITY_ERR_AUTH_REQUIREMENT**

The requested security level could not be reached.

enumerator **BT_SECURITY_ERR_PAIR_NOT_SUPPORTED**

Pairing is not supported.

enumerator **BT_SECURITY_ERR_PAIR_NOT_ALLOWED**

Pairing is not allowed.

enumerator **BT_SECURITY_ERR_INVALID_PARAM**

Invalid parameters.

enumerator **BT_SECURITY_ERR_KEY_REJECTED**

Distributed Key Rejected.

enumerator **BT_SECURITY_ERR_UNSPECIFIED**

Pairing failed but the exact reason could not be specified.

Functions

struct bt_conn *bt_conn_ref(struct bt_conn *conn)

Increment a connection's reference count.

Increment the reference count of a connection object.

> **Note**
>
> Will return NULL if the reference count is zero.

Parameters

- conn – Connection object.

Returns

Connection object with incremented reference count, or NULL if the reference count is zero.

void bt_conn_unref(struct bt_conn *conn)

Decrement a connection's reference count.

Decrement the reference count of a connection object.

Parameters

- conn – Connection object.

void bt_conn_foreach(enum *bt_conn_type* type, void (*func)(struct bt_conn *conn, void *data), void *data)

Iterate through all bt_conn objects.

Iterates through all bt_conn objects that are alive in the Host allocator.

To find established connections, combine this with *bt_conn_get_info*. Check that *bt_conn_info::state* is *BT_CONN_STATE_CONNECTED*.

Thread safety: This API is thread safe, but it does not guarantee a sequentially-consistent view for objects allocated during the current invocation of this API. E.g. If preempted while allocations A then B then C happen then results may include A and C but miss B.

Parameters

- type – Connection Type
- func – Function to call for each connection.
- data – Data to pass to the callback function.

struct bt_conn *bt_conn_lookup_addr_le(uint8_t id, const *bt_addr_le_t* *peer)

Look up an existing connection by address.

Look up an existing connection based on the remote address.

The caller gets a new reference to the connection object which must be released with *bt_conn_unref()* once done using the object.

Parameters

- id – Local identity (in most cases BT_ID_DEFAULT).
- peer – Remote address.

Returns

Connection object or NULL if not found.

const *bt_addr_le_t* ***bt_conn_get_dst**(const struct bt_conn *conn)

Get destination (peer) address of a connection.

Parameters

- **conn** – Connection object.

Returns

Destination address.

uint8_t **bt_conn_index**(const struct bt_conn *conn)

Get array index of a connection.

This function is used to map bt_conn to index of an array of connections. The array has CONFIG_BT_MAX_CONN elements.

Parameters

- **conn** – Connection object.

Returns

Index of the connection object. The range of the returned value is 0..CONFIG_BT_MAX_CONN-1

int **bt_conn_get_info**(const struct bt_conn *conn, struct *bt_conn_info* *info)

Get connection info.

Parameters

- **conn** – Connection object.
- **info** – Connection info object.

Returns

Zero on success or (negative) error code on failure.

int **bt_conn_get_remote_info**(struct bt_conn *conn, struct *bt_conn_remote_info* *remote_info)

Get connection info for the remote device.

Note

In order to retrieve the remote version (version, manufacturer and subversion) CONFIG_BT_REMOTE_VERSION must be enabled

Note

The remote information is exchanged directly after the connection has been established. The application can be notified about when the remote information is available through the remote_info_available callback.

Parameters

- **conn** – Connection object.
- **remote_info** – Connection remote info object.

Returns

Zero on success or (negative) error code on failure.

Returns

-EBUSY The remote information is not yet available.

int **bt_conn_le_get_tx_power_level**(struct bt_conn *conn, struct *bt_conn_le_tx_power* *tx_power_level)

Get connection transmit power level.

Parameters

- **conn** – Connection object.
- **tx_power_level** – Transmit power level descriptor.

Returns
Zero on success or (negative) error code on failure.

Returns
-ENOBUFS HCI command buffer is not available.

int **bt_conn_le_enhanced_get_tx_power_level**(struct bt_conn *conn, struct *bt_conn_le_tx_power* *tx_power)

Get local enhanced connection transmit power level.

Parameters

- **conn** – Connection object.
- **tx_power** – Transmit power level descriptor.

Return values
-ENOBUFS – HCI command buffer is not available.

Returns
Zero on success or (negative) error code on failure.

int **bt_conn_le_get_remote_tx_power_level**(struct bt_conn *conn, enum *bt_conn_le_tx_power_phy* phy)

Get remote (peer) transmit power level.

Parameters

- **conn** – Connection object.
- **phy** – PHY information.

Return values
-ENOBUFS – HCI command buffer is not available.

Returns
Zero on success or (negative) error code on failure.

int **bt_conn_le_set_tx_power_report_enable**(struct bt_conn *conn, bool local_enable, bool remote_enable)

Enable transmit power reporting.

Parameters

- **conn** – Connection object.
- **local_enable** – Enable/disable reporting for local.
- **remote_enable** – Enable/disable reporting for remote.

Return values
-ENOBUFS – HCI command buffer is not available.

Returns
Zero on success or (negative) error code on failure.

int **bt_conn_le_set_path_loss_mon_param**(struct bt_conn *conn, const struct *bt_conn_le_path_loss_reporting_param* *param)

Set Path Loss Monitoring Parameters.

Change the configuration for path loss threshold change events for a given conn handle.

> **Note**
>
> To use this API CONFIG_BT_PATH_LOSS_MONITORING must be set.

Parameters

- **conn** – Connection object.
- **param** – Path Loss Monitoring parameters

Returns

Zero on success or (negative) error code on failure.

int **bt_conn_le_set_path_loss_mon_enable**(struct bt_conn *conn, bool enable)

Enable or Disable Path Loss Monitoring.

Enable or disable Path Loss Monitoring, which will decide whether Path Loss Threshold events are sent from the controller to the host.

> **Note**
>
> To use this API CONFIG_BT_PATH_LOSS_MONITORING must be set.

Parameters

- **conn** – Connection Object.
- **enable** – Enable/disable path loss reporting.

Returns

Zero on success or (negative) error code on failure.

int **bt_conn_le_param_update**(struct bt_conn *conn, const struct *bt_le_conn_param* *param)

Update the connection parameters.

If the local device is in the peripheral role then updating the connection parameters will be delayed. This delay can be configured by through the CONFIG_BT_CONN_PARAM_UPDATE_TIMEOUT option.

Parameters

- **conn** – Connection object.
- **param** – Updated connection parameters.

Returns

Zero on success or (negative) error code on failure.

int **bt_conn_le_data_len_update**(struct bt_conn *conn, const struct *bt_conn_le_data_len_param* *param)

Update the connection transmit data length parameters.

Parameters

- **conn** – Connection object.
- **param** – Updated data length parameters.

Returns
Zero on success or (negative) error code on failure.

int **bt_conn_le_phy_update**(struct bt_conn *conn, const struct *bt_conn_le_phy_param* *param)

Update the connection PHY parameters.

Update the preferred transmit and receive PHYs of the connection. Use *BT_GAP_LE_PHY_NONE* to indicate no preference.

Parameters

- **conn** – Connection object.
- **param** – Updated connection parameters.

Returns
Zero on success or (negative) error code on failure.

int **bt_conn_disconnect**(struct bt_conn *conn, uint8_t reason)

Disconnect from a remote device or cancel pending connection.

Disconnect an active connection with the specified reason code or cancel pending outgoing connection.

The disconnect reason for a normal disconnect should be: BT_HCI_ERR_REMOTE_USER_TERM_CONN.

The following disconnect reasons are accepted:

- BT_HCI_ERR_AUTH_FAIL
- BT_HCI_ERR_REMOTE_USER_TERM_CONN
- BT_HCI_ERR_REMOTE_LOW_RESOURCES
- BT_HCI_ERR_REMOTE_POWER_OFF
- BT_HCI_ERR_UNSUPP_REMOTE_FEATURE
- BT_HCI_ERR_PAIRING_NOT_SUPPORTED
- BT_HCI_ERR_UNACCEPT_CONN_PARAM

Parameters

- **conn** – Connection to disconnect.
- **reason** – Reason code for the disconnection.

Returns
Zero on success or (negative) error code on failure.

int **bt_conn_le_create**(const *bt_addr_le_t* *peer, const struct *bt_conn_le_create_param* *create_param, const struct *bt_le_conn_param* *conn_param, struct bt_conn **conn)

Initiate an LE connection to a remote device.

Allows initiate new LE link to remote peer using its address.

The caller gets a new reference to the connection object which must be released with *bt_conn_unref()* once done using the object.

This uses the General Connection Establishment procedure.

The application must disable explicit scanning before initiating a new LE connection if CONFIG_BT_SCAN_AND_INITIATE_IN_PARALLEL is not enabled.

Parameters

- **peer** – **[in]** Remote address.
- **create_param** – **[in]** Create connection parameters.
- **conn_param** – **[in]** Initial connection parameters.
- **conn** – **[out]** Valid connection object on success.

Returns
Zero on success or (negative) error code on failure.

int **bt_conn_le_create_synced**(const struct bt_le_ext_adv *adv, const struct *bt_conn_le_create_synced_param* *synced_param, const struct *bt_le_conn_param* *conn_param, struct bt_conn **conn)

Create a connection to a synced device.

Initiate a connection to a synced device from a Periodic Advertising with Responses (PAwR) train.

The caller gets a new reference to the connection object which must be released with *bt_conn_unref()* once done using the object.

This uses the Periodic Advertising Connection Procedure.

Parameters

- **adv** – **[in]** The adverting set the PAwR advertiser belongs to.
- **synced_param** – **[in]** Create connection parameters.
- **conn_param** – **[in]** Initial connection parameters.
- **conn** – **[out]** Valid connection object on success.

Returns
Zero on success or (negative) error code on failure.

int **bt_conn_le_create_auto**(const struct *bt_conn_le_create_param* *create_param, const struct *bt_le_conn_param* *conn_param)

Automatically connect to remote devices in the filter accept list.

This uses the Auto Connection Establishment procedure. The procedure will continue until a single connection is established or the procedure is stopped through *bt_conn_create_auto_stop*. To establish connections to all devices in the filter accept list the procedure should be started again in the connected callback after a new connection has been established.

Parameters

- **create_param** – Create connection parameters
- **conn_param** – Initial connection parameters.

Returns
Zero on success or (negative) error code on failure.

Returns
-ENOMEM No free connection object available.

int **bt_conn_create_auto_stop**(void)

Stop automatic connect creation.

Returns
Zero on success or (negative) error code on failure.

int **bt_le_set_auto_conn**(const *bt_addr_le_t* *addr, const struct *bt_le_conn_param* *param)

Automatically connect to remote device if it's in range.

This function enables/disables automatic connection initiation. Every time the device loses the connection with peer, this connection will be re-established if connectable advertisement from peer is received.

Note

Auto connect is disabled during explicit scanning.

Parameters

- **addr** – Remote Bluetooth address.
- **param** – If non-NULL, auto connect is enabled with the given parameters. If NULL, auto connect is disabled.

Returns

Zero on success or error code otherwise.

int **bt_conn_set_security**(struct bt_conn *conn, *bt_security_t* sec)

Set security level for a connection.

This function enable security (encryption) for a connection. If the device has bond information for the peer with sufficiently strong key encryption will be enabled. If the connection is already encrypted with sufficiently strong key this function does nothing.

If the device has no bond information for the peer and is not already paired then the pairing procedure will be initiated. Note that sec has no effect on the security level selected for the pairing process. The selection is instead controlled by the values of the registered *bt_conn_auth_cb*. If the device has bond information or is already paired and the keys are too weak then the pairing procedure will be initiated.

This function may return an error if the required level of security defined using sec is not possible to achieve due to local or remote device limitation (e.g., input output capabilities), or if the maximum number of paired devices has been reached.

This function may return an error if the pairing procedure has already been initiated by the local device or the peer device.

Note

When CONFIG_BT_SMP_SC_ONLY is enabled then the security level will always be level 4.

Note

When CONFIG_BT_SMP_OOB_LEGACY_PAIR_ONLY is enabled then the security level will always be level 3.

Note

When *BT_SECURITY_FORCE_PAIR* within sec is enabled then the pairing procedure will always be initiated.

Parameters

- **conn** – Connection object.
- **sec** – Requested minimum security level.

Returns
0 on success or negative error

bt_security_t **bt_conn_get_security**(const struct bt_conn *conn)

Get security level for a connection.

Returns
Connection security level

uint8_t **bt_conn_enc_key_size**(const struct bt_conn *conn)

Get encryption key size.

This function gets encryption key size. If there is no security (encryption) enabled 0 will be returned.

Parameters

- **conn** – Existing connection object.

Returns
Encryption key size.

int **bt_conn_cb_register**(struct *bt_conn_cb* *cb)

Register connection callbacks.

Register callbacks to monitor the state of connections.

Parameters

- **cb** – Callback struct. Must point to memory that remains valid.

Return values

- **0** – Success.
- **-EEXIST** – if cb was already registered.

int **bt_conn_cb_unregister**(struct *bt_conn_cb* *cb)

Unregister connection callbacks.

Unregister the state of connections callbacks.

Parameters

- **cb** – Callback struct point to memory that remains valid.

Return values

- **0** – Success
- **-EINVAL** – If cb is NULL
- **-ENOENT** – if cb was not registered

static inline const char ***bt_security_err_to_str**(enum *bt_security_err* err)

Converts a security error to string.

Returns
The string representation of the security error code. If CONFIG_BT_SECURITY_ERR_TO_STR is not enabled, this just returns the empty string

void **bt_set_bondable**(bool enable)

Enable/disable bonding.

Set/clear the Bonding flag in the Authentication Requirements of SMP Pairing Request/Response data. The initial value of this flag depends on BT_BONDABLE Kconfig setting. For the vast majority of applications calling this function shouldn't be needed.

Parameters

- **enable** – Value allowing/disallowing to be bondable.

int **bt_conn_set_bondable**(struct bt_conn *conn, bool enable)

Set/clear the bonding flag for a given connection.

Set/clear the Bonding flag in the Authentication Requirements of SMP Pairing Request/Response data for a given connection.

The bonding flag for a given connection cannot be set/cleared if security procedures in the SMP module have already started. This function can be called only once per connection.

If the bonding flag is not set/cleared for a given connection, the value will depend on global configuration which is set using bt_set_bondable. The default value of the global configuration is defined using CONFIG_BT_BONDABLE Kconfig option.

Parameters

- **conn** – Connection object.
- **enable** – Value allowing/disallowing to be bondable.

void **bt_le_oob_set_sc_flag**(bool enable)

Allow/disallow remote LE SC OOB data to be used for pairing.

Set/clear the OOB data flag for LE SC SMP Pairing Request/Response data.

Parameters

- **enable** – Value allowing/disallowing remote LE SC OOB data.

void **bt_le_oob_set_legacy_flag**(bool enable)

Allow/disallow remote legacy OOB data to be used for pairing.

Set/clear the OOB data flag for legacy SMP Pairing Request/Response data.

Parameters

- **enable** – Value allowing/disallowing remote legacy OOB data.

int **bt_le_oob_set_legacy_tk**(struct bt_conn *conn, const uint8_t *tk)

Set OOB Temporary Key to be used for pairing.

This function allows to set OOB data for the LE legacy pairing procedure. The function should only be called in response to the oob_data_request() callback provided that the legacy method is user pairing.

Parameters

- **conn** – Connection object
- **tk** – Pointer to 16 byte long TK array

Returns

Zero on success or -EINVAL if NULL

int **bt_le_oob_set_sc_data**(struct bt_conn *conn, const struct *bt_le_oob_sc_data* *oobd_local, const struct *bt_le_oob_sc_data* *oobd_remote)

Set OOB data during LE Secure Connections (SC) pairing procedure.

This function allows to set OOB data during the LE SC pairing procedure. The function should only be called in response to the oob_data_request() callback provided that LE SC method is used for pairing.

The user should submit OOB data according to the information received in the callback. This may yield three different configurations: with only local OOB data present, with only remote OOB data present or with both local and remote OOB data present.

Parameters

- `conn` – Connection object
- `oobd_local` – Local OOB data or NULL if not present
- `oobd_remote` – Remote OOB data or NULL if not present

Returns

Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

int `bt_le_oob_get_sc_data`(struct bt_conn *conn, const struct *bt_le_oob_sc_data* **oobd_local, const struct *bt_le_oob_sc_data* **oobd_remote)

Get OOB data used for LE Secure Connections (SC) pairing procedure.

This function allows to get OOB data during the LE SC pairing procedure that were set by the *bt_le_oob_set_sc_data()* API.

Note

The OOB data will only be available as long as the connection object associated with it is valid.

Parameters

- `conn` – Connection object
- `oobd_local` – Local OOB data or NULL if not set
- `oobd_remote` – Remote OOB data or NULL if not set

Returns

Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

int `bt_passkey_set`(unsigned int passkey)

Set a fixed passkey to be used for pairing.

This API is only available when the CONFIG_BT_FIXED_PASSKEY configuration option has been enabled.

Sets a fixed passkey to be used for pairing. If set, the pairing_confirm() callback will be called for all incoming pairings.

Parameters

- `passkey` – A valid passkey (0 - 999999) or BT_PASSKEY_INVALID to disable a previously set fixed passkey.

Returns

0 on success or a negative error code on failure.

int **bt_conn_auth_cb_register**(const struct *bt_conn_auth_cb* *cb)

Register authentication callbacks.

Register callbacks to handle authenticated pairing. Passing NULL unregisters a previous callbacks structure.

Parameters

- **cb** – Callback struct.

Returns

Zero on success or negative error code otherwise

int **bt_conn_auth_cb_overlay**(struct bt_conn *conn, const struct *bt_conn_auth_cb* *cb)

Overlay authentication callbacks used for a given connection.

This function can be used only for Bluetooth LE connections. The CONFIG_BT_SMP must be enabled for this function.

The authentication callbacks for a given connection cannot be overlaid if security procedures in the SMP module have already started. This function can be called only once per connection.

Parameters

- **conn** – Connection object.
- **cb** – Callback struct.

Returns

Zero on success or negative error code otherwise

int **bt_conn_auth_info_cb_register**(struct *bt_conn_auth_info_cb* *cb)

Register authentication information callbacks.

Register callbacks to get authenticated pairing information. Multiple registrations can be done.

Parameters

- **cb** – Callback struct.

Returns

Zero on success or negative error code otherwise

int **bt_conn_auth_info_cb_unregister**(struct *bt_conn_auth_info_cb* *cb)

Unregister authentication information callbacks.

Unregister callbacks to stop getting authenticated pairing information.

Parameters

- **cb** – Callback struct.

Returns

Zero on success or negative error code otherwise

int **bt_conn_auth_passkey_entry**(struct bt_conn *conn, unsigned int passkey)

Reply with entered passkey.

This function should be called only after passkey_entry callback from *bt_conn_auth_cb* structure was called.

Parameters

- **conn** – Connection object.
- **passkey** – Entered passkey.

Returns
Zero on success or negative error code otherwise

int **bt_conn_auth_keypress_notify**(struct bt_conn *conn, enum *bt_conn_auth_keypress* type)

Send Passkey Keypress Notification during pairing.

This function may be called only after passkey_entry callback from *bt_conn_auth_cb* structure was called.

Requires `CONFIG_BT_PASSKEY_KEYPRESS`.

See also

bt_conn_auth_keypress.

Parameters

- **conn** – Destination for the notification.
- **type** – What keypress event type to send.

Return values

- **0** – Success
- **-EINVAL** – Improper use of the API.
- **-ENOMEM** – Failed to allocate.
- **-ENOBUFS** – Failed to allocate.

int **bt_conn_auth_cancel**(struct bt_conn *conn)

Cancel ongoing authenticated pairing.

This function allows to cancel ongoing authenticated pairing.

Parameters

- **conn** – Connection object.

Returns
Zero on success or negative error code otherwise

int **bt_conn_auth_passkey_confirm**(struct bt_conn *conn)

Reply if passkey was confirmed to match by user.

This function should be called only after passkey_confirm callback from *bt_conn_auth_cb* structure was called.

Parameters

- **conn** – Connection object.

Returns
Zero on success or negative error code otherwise

int **bt_conn_auth_pairing_confirm**(struct bt_conn *conn)

Reply if incoming pairing was confirmed by user.

This function should be called only after pairing_confirm callback from *bt_conn_auth_cb* structure was called if user confirmed incoming pairing.

Parameters

- **conn** – Connection object.

Returns
Zero on success or negative error code otherwise

int **bt_conn_auth_pincode_entry**(struct bt_conn *conn, const char *pin)

Reply with entered PIN code.

This function should be called only after PIN code callback from *bt_conn_auth_cb* structure was called. It's for legacy 2.0 devices.

Parameters

- **conn** – Connection object.
- **pin** – Entered PIN code.

Returns
Zero on success or negative error code otherwise

struct bt_conn *__bt_conn_create_br__(const *bt_addr_t* *peer, const struct *bt_br_conn_param* *param)

Initiate an BR/EDR connection to a remote device.

Allows initiate new BR/EDR link to remote peer using its address.

The caller gets a new reference to the connection object which must be released with *bt_conn_unref()* once done using the object.

Parameters

- **peer** – Remote address.
- **param** – Initial connection parameters.

Returns
Valid connection object on success or NULL otherwise.

struct **bt_le_conn_param**

#include <*conn.h*> Connection parameters for LE connections.

struct **bt_conn_le_phy_info**

#include <*conn.h*> Connection PHY information for LE connections.

Public Members

uint8_t **rx_phy**

Connection transmit PHY.

struct **bt_conn_le_phy_param**

#include <*conn.h*> Preferred PHY parameters for LE connections.

Public Members

uint16_t **options**

Connection PHY options.

uint8_t **pref_tx_phy**

Bitmask of preferred transmit PHYs.

uint8_t **pref_rx_phy**

Bitmask of preferred receive PHYs.

struct **bt_conn_le_data_len_info**

#include <*conn.h*> Connection data length information for LE connections.

Public Members

uint16_t **tx_max_len**

Maximum Link Layer transmission payload size in bytes.

uint16_t **tx_max_time**

Maximum Link Layer transmission payload time in us.

uint16_t **rx_max_len**

Maximum Link Layer reception payload size in bytes.

uint16_t **rx_max_time**

Maximum Link Layer reception payload time in us.

struct **bt_conn_le_data_len_param**

#include <*conn.h*> Connection data length parameters for LE connections.

Public Members

uint16_t **tx_max_len**

Maximum Link Layer transmission payload size in bytes.

uint16_t **tx_max_time**

Maximum Link Layer transmission payload time in us.

struct **bt_conn_le_info**

#include <*conn.h*> LE Connection Info Structure.

Public Members

const *bt_addr_le_t* ***src**

Source (Local) Identity Address.

const *bt_addr_le_t* ***dst**

Destination (Remote) Identity Address or remote Resolvable Private Address (RPA) before identity has been resolved.

const *bt_addr_le_t* ***local**

Local device address used during connection setup.

const *bt_addr_le_t* ***remote**
Remote device address used during connection setup.

uint16_t **interval**
Connection interval.

uint16_t **latency**
Connection peripheral latency.

uint16_t **timeout**
Connection supervision timeout.

struct **bt_conn_br_info**
#include <*conn.h*> BR/EDR Connection Info Structure.

Public Members

const *bt_addr_t* ***dst**
Destination (Remote) BR/EDR address.

struct **bt_security_info**
#include <*conn.h*> Security Info Structure.

Public Members

bt_security_t **level**
Security Level.

uint8_t **enc_key_size**
Encryption Key Size.

enum *bt_security_flag* **flags**
Flags.

struct **bt_conn_info**
#include <*conn.h*> Connection Info Structure.

Public Members

enum *bt_conn_type* **type**
Connection Type.

uint8_t **role**
Connection Role.

uint8_t **id**
: Which local identity the connection was created with.

struct *bt_conn_le_info* **le**
: LE Connection specific Info.

struct *bt_conn_br_info* **br**
: BR/EDR Connection specific Info.

union bt_conn_info
: Connection Type specific Info.

enum *bt_conn_state* **state**
: Connection state.

struct *bt_security_info* **security**
: Security specific info.

struct **bt_conn_le_remote_info**
#include <*conn.h*> LE Connection Remote Info Structure.

Public Members

const uint8_t ***features**
: Remote LE feature set (bitmask).

struct **bt_conn_br_remote_info**
#include <*conn.h*> BR/EDR Connection Remote Info structure.

Public Members

const uint8_t ***features**
: Remote feature set (pages of bitmasks).

uint8_t **num_pages**
: Number of pages in the remote feature set.

struct **bt_conn_remote_info**
#include <*conn.h*> Connection Remote Info Structure.

> **Note**
>
> The version, manufacturer and subversion fields will only contain valid data if `CONFIG_BT_REMOTE_VERSION` is enabled.

Public Members

uint8_t **type**
Connection Type.

uint8_t **version**
Remote Link Layer version.

uint16_t **manufacturer**
Remote manufacturer identifier.

uint16_t **subversion**
Per-manufacturer unique revision.

struct *bt_conn_le_remote_info* **le**
LE connection remote info.

struct *bt_conn_br_remote_info* **br**
BR/EDR connection remote info.

struct **bt_conn_le_tx_power**
#include <*conn.h*> LE Transmit Power Level Structure.

Public Members

uint8_t **phy**
Input: 1M, 2M, Coded S2 or Coded S8.

int8_t **current_level**
Output: current transmit power level.

int8_t **max_level**
Output: maximum transmit power level.

struct **bt_conn_le_tx_power_report**
#include <*conn.h*> LE Transmit Power Reporting Structure.

Public Members

uint8_t **reason**
Reason for Transmit power reporting, as documented in Core Spec.
Version 5.4 Vol. 4, Part E, 7.7.65.33.

enum *bt_conn_le_tx_power_phy* **phy**
Phy of Transmit power reporting.

int8_t **tx_power_level**

Transmit power level.

- 0xXX - Transmit power level
  - Range: -127 to 20
  - Units: dBm
- 0x7E - Remote device is not managing power levels on this PHY.
- 0x7F - Transmit power level is not available

uint8_t **tx_power_level_flag**

Bit 0: Transmit power level is at minimum level.

Bit 1: Transmit power level is at maximum level.

int8_t **delta**

Change in transmit power level.

- 0xXX - Change in transmit power level (positive indicates increased power, negative indicates decreased power, zero indicates unchanged) Units: dB
- 0x7F - Change is not available or is out of range.

struct **bt_conn_le_path_loss_threshold_report**

#include <*conn.h*> LE Path Loss Monitoring Threshold Change Report Structure.

Public Members

enum *bt_conn_le_path_loss_zone* **zone**

Path Loss zone as documented in Core Spec.

Version 5.4 Vol.4, Part E, 7.7.65.32.

uint8_t **path_loss**

Current path loss (dB).

struct **bt_conn_le_path_loss_reporting_param**

#include <*conn.h*> LE Path Loss Monitoring Parameters Structure as defined in Core Spec.

Version 5.4 Vol.4, Part E, 7.8.119 LE Set Path Loss Reporting Parameters command.

Public Members

uint8_t **high_threshold**

High threshold for the path loss (dB).

uint8_t **high_hysteresis**

Hysteresis value for the high threshold (dB).

uint8_t **low_threshold**

Low threshold for the path loss (dB).

uint8_t **low_hysteresis**

Hysteresis value for the low threshold (dB).

uint16_t **min_time_spent**

Minimum time in number of connection events to be observed once the path loss crosses the threshold before an event is generated.

struct **bt_conn_le_create_param**

#include <conn.h>

Public Members

uint32_t **options**

Bit-field of create connection options.

uint16_t **interval**

Scan interval (N * 0.625 ms)

uint16_t **window**

Scan window (N * 0.625 ms)

uint16_t **interval_coded**

Scan interval LE Coded PHY (N * 0.625 MS)

Set zero to use same as LE 1M PHY scan interval

uint16_t **window_coded**

Scan window LE Coded PHY (N * 0.625 MS)

Set zero to use same as LE 1M PHY scan window.

uint16_t **timeout**

Connection initiation timeout (N * 10 MS)

Set zero to use the default `CONFIG_BT_CREATE_CONN_TIMEOUT` timeout.

> **Note**
>
> Unused in *bt_conn_le_create_auto*

struct **bt_conn_le_create_synced_param**

#include <conn.h>

Public Members

const *bt_addr_le_t* **peer**

Remote address.

The peer must be synchronized to the PAwR train.

uint8_t **subevent**

The subevent where the connection will be initiated.

struct **bt_conn_cb**

#include <conn.h> Connection callback structure.

This structure is used for tracking the state of a connection. It is registered with the help of the *bt_conn_cb_register()* API. It's permissible to register multiple instances of this *bt_conn_cb* type, in case different modules of an application are interested in tracking the connection state. If a callback is not of interest for an instance, it may be set to NULL and will as a consequence not be used for that instance.

Public Members

void (***connected**)(struct bt_conn *conn, uint8_t err)

A new connection has been established.

This callback notifies the application of a new connection. In case the err parameter is non-zero it means that the connection establishment failed.

`err` can mean either of the following:

- BT_HCI_ERR_UNKNOWN_CONN_ID Creating the connection started by *bt_conn_le_create* was canceled either by the user through *bt_conn_disconnect* or by the timeout in the host through *bt_conn_le_create_param* timeout parameter, which defaults to `CONFIG_BT_CREATE_CONN_TIMEOUT` seconds.
- `BT_HCI_ERR_ADV_TIMEOUT` High duty cycle directed connectable advertiser started by *bt_le_adv_start* failed to be connected within the timeout.

Note

If the connection was established from an advertising set then the advertising set cannot be restarted directly from this callback. Instead use the connected callback of the advertising set.

Param conn
New connection object.
Param err
HCI error. Zero for success, non-zero otherwise.

void (***disconnected**)(struct bt_conn *conn, uint8_t reason)

A connection has been disconnected.

This callback notifies the application that a connection has been disconnected.

When this callback is called the stack still has one reference to the connection object. If the application in this callback tries to start either a connectable advertiser or create a new connection this might fail because there are no free connection objects available. To avoid this issue it is recommended to either start connectable advertise or create a new connection using *k_work_submit* or increase `CONFIG_BT_MAX_CONN` .

Param conn
Connection object.
Param reason
BT_HCI_ERR_* reason for the disconnection.

void (*`recycled`)(void)

A connection object has been returned to the pool.

This callback notifies the application that it might be able to allocate a connection object. No guarantee, first come, first serve.

Use this to e.g. re-start connectable advertising or scanning.

Treat this callback as an ISR, as it originates from *bt_conn_unref* which is used by the BT stack. Making Bluetooth API calls in this context is error-prone and strongly discouraged.

bool (*`le_param_req`)(struct bt_conn *conn, struct *bt_le_conn_param* *param)

LE connection parameter update request.

This callback notifies the application that a remote device is requesting to update the connection parameters. The application accepts the parameters by returning true, or rejects them by returning false. Before accepting, the application may also adjust the parameters to better suit its needs.

It is recommended for an application to have just one of these callbacks for simplicity. However, if an application registers multiple it needs to manage the potentially different requirements for each callback. Each callback gets the parameters as returned by previous callbacks, i.e. they are not necessarily the same ones as the remote originally sent.

If the application does not have this callback then the default is to accept the parameters.
Param conn
Connection object.
Param param
Proposed connection parameters.
Return
true to accept the parameters, or false to reject them.

void (*`le_param_updated`)(struct bt_conn *conn, uint16_t interval, uint16_t latency, uint16_t timeout)

The parameters for an LE connection have been updated.

This callback notifies the application that the connection parameters for an LE connection have been updated.
Param conn
Connection object.
Param interval
Connection interval.
Param latency
Connection latency.
Param timeout
Connection supervision timeout.

void (*`identity_resolved`)(struct bt_conn *conn, const *bt_addr_le_t* *rpa, const *bt_addr_le_t* *identity)

Remote Identity Address has been resolved.

This callback notifies the application that a remote Identity Address has been resolved

Param conn
Connection object.

Param rpa
Resolvable Private Address.

Param identity
Identity Address.

void (*`security_changed`)(struct bt_conn *conn, *bt_security_t* level, enum *bt_security_err* err)

The security level of a connection has changed.

This callback notifies the application that the security of a connection has changed.

The security level of the connection can either have been increased or remain unchanged. An increased security level means that the pairing procedure has been performed or the bond information from a previous connection has been applied. If the security level remains unchanged this means that the encryption key has been refreshed for the connection.

Param conn
Connection object.

Param level
New security level of the connection.

Param err
Security error. Zero for success, non-zero otherwise.

void (*`remote_info_available`)(struct bt_conn *conn, struct *bt_conn_remote_info* *remote_info)

Remote information procedures has completed.

This callback notifies the application that the remote information has been retrieved from the remote peer.

Param conn
Connection object.

Param remote_info
Connection information of remote device.

void (*`le_phy_updated`)(struct bt_conn *conn, struct *bt_conn_le_phy_info* *param)

The PHY of the connection has changed.

This callback notifies the application that the PHY of the connection has changed.

Param conn
Connection object.

Param info
Connection LE PHY information.

void (*`le_data_len_updated`)(struct bt_conn *conn, struct *bt_conn_le_data_len_info* *info)

The data length parameters of the connection has changed.

This callback notifies the application that the maximum Link Layer payload length or transmission time has changed.

Param conn
Connection object.

Param info
Connection data length information.

struct **bt_conn_oob_info**

#include <conn.h> Info Structure for OOB pairing.

Public Types

Type of OOB pairing method.

Values:

enumerator **BT_CONN_OOB_LE_LEGACY**

LE legacy pairing.

enumerator **BT_CONN_OOB_LE_SC**

LE SC pairing.

Public Members

enum *bt_conn_oob_info* **type**

Type of OOB pairing method.

enum *bt_conn_oob_info* **oob_config**

OOB data configuration.

struct *bt_conn_oob_info* **lesc**

LE Secure Connections OOB pairing parameters.

struct **bt_conn_pairing_feat**

#include <conn.h> Pairing request and pairing response info structure.

This structure is the same for both smp_pairing_req and smp_pairing_rsp and a subset of the packet data, except for the initial Code octet. It is documented in Core Spec. Vol. 3, Part H, 3.5.1 and 3.5.2.

Public Members

uint8_t **io_capability**

IO Capability, Core Spec.

Vol 3, Part H, 3.5.1, Table 3.4

uint8_t **oob_data_flag**

OOB data flag, Core Spec.

Vol 3, Part H, 3.5.1, Table 3.5

uint8_t **auth_req**

AuthReq, Core Spec.

Vol 3, Part H, 3.5.1, Fig. 3.3

uint8_t **max_enc_key_size**

Maximum Encryption Key Size, Core Spec.

Vol 3, Part H, 3.5.1

uint8_t **init_key_dist**

Initiator Key Distribution/Generation, Core Spec.

Vol 3, Part H, 3.6.1, Fig. 3.11

uint8_t **resp_key_dist**

Responder Key Distribution/Generation, Core Spec.

Vol 3, Part H 3.6.1, Fig. 3.11

struct **bt_conn_auth_cb**

#include <conn.h> Authenticated pairing callback structure.

Public Members

enum *bt_security_err* (***pairing_accept**)(struct bt_conn *conn, const struct *bt_conn_pairing_feat* *const feat)

Query to proceed incoming pairing or not.

On any incoming pairing req/rsp this callback will be called for the application to decide whether to allow for the pairing to continue.

The pairing info received from the peer is passed to assist making the decision.

As this callback is synchronous the application should return a response value immediately. Otherwise it may affect the timing during pairing. Hence, this information should not be conveyed to the user to take action.

The remaining callbacks are not affected by this, but do notice that other callbacks can be called during the pairing. Eg. if pairing_confirm is registered both will be called for Just-Works pairings.

This callback may be unregistered in which case pairing continues as if the Kconfig flag was not set.

This callback is not called for BR/EDR Secure Simple Pairing (SSP).

Param conn
Connection where pairing is initiated.

Param feat
Pairing req/resp info.

void (***passkey_display**)(struct bt_conn *conn, unsigned int passkey)

Display a passkey to the user.

When called the application is expected to display the given passkey to the user, with the expectation that the passkey will then be entered on the peer device. The passkey will be in the range of 0 - 999999, and is expected to be padded with zeroes so that six digits are always shown. E.g. the value 37 should be shown as 000037.

This callback may be set to NULL, which means that the local device lacks the ability do display a passkey. If set to non-NULL the cancel callback must also be provided, since this is the only way the application can find out that it should stop displaying the passkey.

Param conn
Connection where pairing is currently active.
Param passkey
Passkey to show to the user.

void (*`passkey_entry`)(struct bt_conn *conn)

Request the user to enter a passkey.

When called the user is expected to enter a passkey. The passkey must be in the range of 0 - 999999, and should be expected to be zero-padded, as that's how the peer device will typically be showing it (e.g. 37 would be shown as 000037).

Once the user has entered the passkey its value should be given to the stack using the *bt_conn_auth_passkey_entry()* API.

This callback may be set to NULL, which means that the local device lacks the ability to enter a passkey. If set to non-NULL the cancel callback must also be provided, since this is the only way the application can find out that it should stop requesting the user to enter a passkey.
Param conn
Connection where pairing is currently active.

void (*`passkey_confirm`)(struct bt_conn *conn, unsigned int passkey)

Request the user to confirm a passkey.

When called the user is expected to confirm that the given passkey is also shown on the peer device.. The passkey will be in the range of 0 - 999999, and should be zero-padded to always be six digits (e.g. 37 would be shown as 000037).

Once the user has confirmed the passkey to match, the *bt_conn_auth_passkey_confirm()* API should be called. If the user concluded that the passkey doesn't match the *bt_conn_auth_cancel()* API should be called.

This callback may be set to NULL, which means that the local device lacks the ability to confirm a passkey. If set to non-NULL the cancel callback must also be provided, since this is the only way the application can find out that it should stop requesting the user to confirm a passkey.
Param conn
Connection where pairing is currently active.
Param passkey
Passkey to be confirmed.

void (*`oob_data_request`)(struct bt_conn *conn, struct *bt_conn_oob_info* *info)

Request the user to provide Out of Band (OOB) data.

When called the user is expected to provide OOB data. The required data are indicated by the information structure.

For LE Secure Connections OOB pairing, the user should provide local OOB data, remote OOB data or both depending on their availability. Their value should be given to the stack using the *bt_le_oob_set_sc_data()* API.

This callback must be set to non-NULL in order to support OOB pairing.
Param conn
Connection where pairing is currently active.
Param info
OOB pairing information.

void (*`cancel`)(struct bt_conn *conn)

Cancel the ongoing user request.

This callback will be called to notify the application that it should cancel any previous user request (passkey display, entry or confirmation).

This may be set to NULL, but must always be provided whenever the passkey_display, passkey_entry passkey_confirm or pairing_confirm callback has been provided.

Param conn
Connection where pairing is currently active.

void (*`pairing_confirm`)(struct bt_conn *conn)

Request confirmation for an incoming pairing.

This callback will be called to confirm an incoming pairing request where none of the other user callbacks is applicable.

If the user decides to accept the pairing the *bt_conn_auth_pairing_confirm()* API should be called. If the user decides to reject the pairing the *bt_conn_auth_cancel()* API should be called.

This callback may be set to NULL, which means that the local device lacks the ability to confirm a pairing request. If set to non-NULL the cancel callback must also be provided, since this is the only way the application can find out that it should stop requesting the user to confirm a pairing request.

Param conn
Connection where pairing is currently active.

void (*`pincode_entry`)(struct bt_conn *conn, bool highsec)

Request the user to enter a passkey.

This callback will be called for a BR/EDR (Bluetooth Classic) connection where pairing is being performed. Once called the user is expected to enter a PIN code with a length between 1 and 16 digits. If the *highsec* parameter is set to true the PIN code must be 16 digits long.

Once entered, the PIN code should be given to the stack using the *bt_conn_auth_pincode_entry()* API.

This callback may be set to NULL, however in that case pairing over BR/EDR will not be possible. If provided, the cancel callback must be provided as well.

Param conn
Connection where pairing is currently active.

Param highsec
true if 16 digit PIN is required.

struct `bt_conn_auth_info_cb`

#include <conn.h> Authenticated pairing information callback structure.

Public Members

void (*`pairing_complete`)(struct bt_conn *conn, bool bonded)

notify that pairing procedure was complete.

This callback notifies the application that the pairing procedure has been completed.

Param conn
Connection object.

Param bonded
Bond information has been distributed during the pairing procedure.

void (*pairing_failed)(struct bt_conn *conn, enum *bt_security_err* reason)

notify that pairing process has failed.

Param conn
Connection object.

Param reason
Pairing failed reason

void (*bond_deleted)(uint8_t id, const *bt_addr_le_t* *peer)

Notify that bond has been deleted.

This callback notifies the application that the bond information for the remote peer has been deleted

Param id
Which local identity had the bond.

Param peer
Remote address.

sys_snode_t node

Internally used field for list handling.

struct bt_br_conn_param

#include <*conn.h*> Connection parameters for BR/EDR connections.

Data Buffers

API Reference

group bt_buf

Data buffers.

Defines

BT_BUF_RESERVE

BT_BUF_SIZE(size)

Helper to include reserved HCI data in buffer calculations.

BT_BUF_ACL_SIZE(size)

Helper to calculate needed buffer size for HCI ACL packets.

BT_BUF_EVT_SIZE(size)

Helper to calculate needed buffer size for HCI Event packets.

BT_BUF_CMD_SIZE(size)

Helper to calculate needed buffer size for HCI Command packets.

BT_BUF_ISO_SIZE(size)

Helper to calculate needed buffer size for HCI ISO packets.

BT_BUF_ACL_RX_SIZE

Data size needed for HCI ACL RX buffers.

BT_BUF_EVT_RX_SIZE
Data size needed for HCI Event RX buffers.

BT_BUF_ISO_RX_SIZE

BT_BUF_ISO_RX_COUNT

BT_BUF_RX_SIZE
Data size needed for HCI ACL, HCI ISO or Event RX buffers.

BT_BUF_RX_COUNT
Buffer count needed for HCI ACL, HCI ISO or Event RX buffers.

BT_BUF_CMD_TX_SIZE
Data size needed for HCI Command buffers.

Enums

enum bt_buf_type
Possible types of buffers passed around the Bluetooth stack.
Values:

enumerator BT_BUF_CMD
HCI command.

enumerator BT_BUF_EVT
HCI event.

enumerator BT_BUF_ACL_OUT
Outgoing ACL data.

enumerator BT_BUF_ACL_IN
Incoming ACL data.

enumerator BT_BUF_ISO_OUT
Outgoing ISO data.

enumerator BT_BUF_ISO_IN
Incoming ISO data.

enumerator BT_BUF_H4
H:4 data.

Functions

struct *net_buf* ***bt_buf_get_rx**(enum *bt_buf_type* type, *k_timeout_t* timeout)

Allocate a buffer for incoming data.

This will set the buffer type so *bt_buf_set_type()* does not need to be explicitly called.

Parameters

- **type** – Type of buffer. Only BT_BUF_EVT, BT_BUF_ACL_IN and BT_BUF_ISO_IN are allowed.
- **timeout** – Non-negative waiting period to obtain a buffer or one of the special values K_NO_WAIT and K_FOREVER.

Returns
A new buffer.

struct *net_buf* ***bt_buf_get_tx**(enum *bt_buf_type* type, *k_timeout_t* timeout, const void *data, size_t size)

Allocate a buffer for outgoing data.

This will set the buffer type so *bt_buf_set_type()* does not need to be explicitly called.

Parameters

- **type** – Type of buffer. Only BT_BUF_CMD, BT_BUF_ACL_OUT or BT_BUF_H4, when operating on H:4 mode, are allowed.
- **timeout** – Non-negative waiting period to obtain a buffer or one of the special values K_NO_WAIT and K_FOREVER.
- **data** – Initial data to append to buffer.
- **size** – Initial data size.

Returns
A new buffer.

struct *net_buf* ***bt_buf_get_evt**(uint8_t evt, bool discardable, *k_timeout_t* timeout)

Allocate a buffer for an HCI Event.

This will set the buffer type so *bt_buf_set_type()* does not need to be explicitly called.

Parameters

- **evt** – HCI event code
- **discardable** – Whether the driver considers the event discardable.
- **timeout** – Non-negative waiting period to obtain a buffer or one of the special values K_NO_WAIT and K_FOREVER.

Returns
A new buffer.

static inline void **bt_buf_set_type**(struct *net_buf* *buf, enum *bt_buf_type* type)

Set the buffer type.

Parameters

- **buf** – Bluetooth buffer
- **type** – The BT_* type to set the buffer to

static inline enum *bt_buf_type* **bt_buf_get_type**(struct *net_buf* *buf)

Get the buffer type.

Parameters

- **buf** – Bluetooth buffer

Returns
The BT_* type to of the buffer

struct **bt_buf_data**
#include <*buf.h*> This is a base type for bt_buf user data.

HCI Drivers

API Reference

group **bt_hci_driver**
HCI drivers.

Deprecated:
This is the old HCI driver API. Drivers should use Bluetooth HCI APIs instead.

Enums

Values:

enumerator **BT_QUIRK_NO_RESET** = *BIT*(0)

enumerator **BT_QUIRK_NO_AUTO_DLE** = *BIT*(1)

enum **bt_hci_driver_bus**
Possible values for the 'bus' member of the *bt_hci_driver* struct.

Values:

enumerator **BT_HCI_DRIVER_BUS_VIRTUAL** = 0

enumerator **BT_HCI_DRIVER_BUS_USB** = 1

enumerator **BT_HCI_DRIVER_BUS_PCCARD** = 2

enumerator **BT_HCI_DRIVER_BUS_UART** = 3

enumerator **BT_HCI_DRIVER_BUS_RS232** = 4

enumerator **BT_HCI_DRIVER_BUS_PCI** = 5

enumerator **BT_HCI_DRIVER_BUS_SDIO** = 6

enumerator **BT_HCI_DRIVER_BUS_SPI** = 7

enumerator **BT_HCI_DRIVER_BUS_I2C** = 8

enumerator BT_HCI_DRIVER_BUS_IPM = 9

Functions

int bt_recv(struct *net_buf* *buf)

Receive data from the controller/HCI driver.

This is the main function through which the HCI driver provides the host with data from the controller. The buffer needs to have its type set with the help of *bt_buf_set_type()* before calling this API.

Deprecated:

Use the new HCI driver interface instead: Bluetooth HCI APIs

Parameters

- buf – Network buffer containing data from the controller.

Returns

0 on success or negative error number on failure.

int bt_hci_driver_register(const struct *bt_hci_driver* *drv)

Register a new HCI driver to the Bluetooth stack.

This needs to be called before any application code runs. The *bt_enable()* API will fail if there is no driver registered.

Deprecated:

Use the new HCI driver interface instead: Bluetooth HCI APIs

Parameters

- drv – A *bt_hci_driver* struct representing the driver.

Returns

0 on success or negative error number on failure.

int bt_hci_transport_setup(const struct *device* *dev)

Setup the HCI transport, which usually means to reset the Bluetooth IC.

Note

A weak version of this function is included in the H4 driver, so defining it is optional per board.

Parameters

- dev – The device structure for the bus connecting to the IC

Returns

0 on success, negative error value on failure

int **bt_hci_transport_teardown**(const struct *device* *dev)

Teardown the HCI transport.

> **Note**
>
> A weak version of this function is included in the IPC driver, so defining it is optional. NRF5340 includes support to put network core in reset state.

Parameters

- **dev** – The device structure for the bus connecting to the IC

Returns

0 on success, negative error value on failure

struct *net_buf* ***bt_hci_evt_create**(uint8_t evt, uint8_t len)

Allocate an HCI event buffer.

This function allocates a new buffer for an HCI event. It is given the event code and the total length of the parameters. Upon successful return the buffer is ready to have the parameters encoded into it.

Parameters

- **evt** – Event OpCode.
- **len** – Length of event parameters.

Returns

Newly allocated buffer.

struct *net_buf* ***bt_hci_cmd_complete_create**(uint16_t op, uint8_t plen)

Allocate an HCI Command Complete event buffer.

This function allocates a new buffer for HCI Command Complete event. It is given the OpCode (encoded e.g. using the BT_OP macro) and the total length of the parameters. Upon successful return the buffer is ready to have the parameters encoded into it.

Parameters

- **op** – Command OpCode.
- **plen** – Length of command parameters.

Returns

Newly allocated buffer.

struct *net_buf* ***bt_hci_cmd_status_create**(uint16_t op, uint8_t status)

Allocate an HCI Command Status event buffer.

This function allocates a new buffer for HCI Command Status event. It is given the OpCode (encoded e.g. using the BT_OP macro) and the status code. Upon successful return the buffer is ready to have the parameters encoded into it.

Parameters

- **op** – Command OpCode.
- **status** – Status code.

Returns

Newly allocated buffer.

struct **bt_hci_setup_params**

#include <bluetooth.h>

Public Members

bt_addr_t **public_addr**

The public identity address to give to the controller.

This field is used when the driver selects CONFIG_BT_HCI_SET_PUBLIC_ADDR to indicate that it supports setting the controller's public address.

struct **bt_hci_driver**

#include <hci_driver.h> Abstraction which represents the HCI transport to the controller.

This struct is used to represent the HCI transport to the Bluetooth controller.

Public Members

const char ***name**

Name of the driver.

enum *bt_hci_driver_bus* **bus**

Bus of the transport (BT_HCI_DRIVER_BUS_*)

uint32_t **quirks**

Specific controller quirks.

These are set by the HCI driver and acted upon by the host. They can either be statically set at buildtime, or set at runtime before the HCI driver's *open()* callback returns.

int (***open**)(void)

Open the HCI transport.

Opens the HCI transport for operation. This function must not return until the transport is ready for operation, meaning it is safe to start calling the *send()* handler.

Return

0 on success or negative error number on failure.

int (***close**)(void)

Close the HCI transport.

Closes the HCI transport. This function must not return until the transport is closed.

Return

0 on success or negative error number on failure.

int (***send**)(struct *net_buf* *buf)

Send HCI buffer to controller.

Send an HCI command or ACL data to the controller. The exact type of the data can be checked with the help of *bt_buf_get_type()*.

> **Note**
>
> This function must only be called from a cooperative thread.

Param buf
Buffer containing data to be sent to the controller.
Return
0 on success or negative error number on failure.

int (***setup**)(const struct *bt_hci_setup_params* *params)

HCI vendor-specific setup.

Executes vendor-specific commands sequence to initialize BT Controller before BT Host executes Reset sequence.

> **Note**
>
> `CONFIG_BT_HCI_SETUP` must be selected for this field to be available.

Return
0 on success or negative error number on failure.

HCI RAW channel

Overview HCI RAW channel API is intended to expose HCI interface to the remote entity. The local Bluetooth controller gets owned by the remote entity and host Bluetooth stack is not used. RAW API provides direct access to packets which are sent and received by the Bluetooth HCI driver.

API Reference

group `hci_raw`

HCI RAW channel.

Defines

`BT_HCI_ERR_EXT_HANDLED`

`BT_HCI_RAW_CMD_EXT`(_op, _min_len, _func)

Helper macro to define a command extension.

Parameters

- `_op` – Opcode of the command.
- `_min_len` – Minimal length of the command.
- `_func` – Handler function to be called.

Enums

Values:

enumerator **BT_HCI_RAW_MODE_PASSTHROUGH** = 0x00

Passthrough mode.

```
While in this mode the buffers are passed as is between the stack
and the driver.
```

enumerator **BT_HCI_RAW_MODE_H4** = 0x01

H:4 mode.

```
While in this mode H:4 headers will added into the buffers
according to the buffer type when coming from the stack and will be
removed and used to set the buffer type.
```

Functions

int **bt_send**(struct *net_buf* *buf)

Send packet to the Bluetooth controller.

Send packet to the Bluetooth controller. Caller needs to implement netbuf pool.

Parameters

- **buf** – netbuf packet to be send

Returns

Zero on success or (negative) error code otherwise.

int **bt_hci_raw_set_mode**(uint8_t mode)

Set Bluetooth RAW channel mode.

Set access mode of Bluetooth RAW channel.

Parameters

- **mode** – Access mode.

Returns

Zero on success or (negative) error code otherwise.

uint8_t **bt_hci_raw_get_mode**(void)

Get Bluetooth RAW channel mode.

Get access mode of Bluetooth RAW channel.

Returns

Access mode.

void **bt_hci_raw_cmd_ext_register**(struct *bt_hci_raw_cmd_ext* *cmds, size_t size)

Register Bluetooth RAW command extension table.

Register Bluetooth RAW channel command extension table, opcodes in this table are intercepted to sent to the handler function.

Parameters

- **cmds** – Pointer to the command extension table.
- **size** – Size of the command extension table.

int **bt_enable_raw**(struct k_fifo *rx_queue)

Enable Bluetooth RAW channel:

Enable Bluetooth RAW HCI channel.

Parameters

- **rx_queue** – netbuf queue where HCI packets received from the Bluetooth controller are to be queued. The queue is defined in the caller while the available buffers pools are handled in the stack.

Returns

Zero on success or (negative) error code otherwise.

struct **bt_hci_raw_cmd_ext**

#include <hci_raw.h>

Public Members

uint16_t **op**

Opcode of the command.

size_t **min_len**

Minimal length of the command.

uint8_t (***func**)(struct *net_buf* *buf)

Handler function.

Handler function to be called when a command is intercepted.

Param buf

Buffer containing the command.

Return

HCI Status code or BT_HCI_ERR_EXT_HANDLED if command has been handled already and a response has been sent as oppose to BT_HCI_ERR_SUCCESS which just indicates that the command can be sent to the controller to be processed.

Cryptography

API Reference

group **bt_crypto**

Cryptography.

Functions

int **bt_rand**(void *buf, size_t len)

Generate random data.

A random number generation helper which utilizes the Bluetooth controller's own RNG.

Parameters

- **buf** – Buffer to insert the random data

- **len** – Length of random data to generate

Returns
Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error

int **bt_encrypt_le**(const uint8_t key[16], const uint8_t plaintext[16], uint8_t enc_data[16])

AES encrypt little-endian data.

An AES encrypt helper is used to request the Bluetooth controller's own hardware to encrypt the plaintext using the key and returns the encrypted data.

Parameters

- **key** – 128 bit LS byte first key for the encryption of the plaintext
- **plaintext** – 128 bit LS byte first plaintext data block to be encrypted
- **enc_data** – 128 bit LS byte first encrypted data block

Returns
Zero on success or error code otherwise.

int **bt_encrypt_be**(const uint8_t key[16], const uint8_t plaintext[16], uint8_t enc_data[16])

AES encrypt big-endian data.

An AES encrypt helper is used to request the Bluetooth controller's own hardware to encrypt the plaintext using the key and returns the encrypted data.

Parameters

- **key** – 128 bit MS byte first key for the encryption of the plaintext
- **plaintext** – 128 bit MS byte first plaintext data block to be encrypted
- **enc_data** – 128 bit MS byte first encrypted data block

Returns
Zero on success or error code otherwise.

int **bt_ccm_decrypt**(const uint8_t key[16], uint8_t nonce[13], const uint8_t *enc_data, size_t len, const uint8_t *aad, size_t aad_len, uint8_t *plaintext, size_t mic_size)

Decrypt big-endian data with AES-CCM.

Decrypts and authorizes enc_data with AES-CCM, as described in https://tools.ietf.org/html/rfc3610.

Assumes that the MIC follows directly after the encrypted data.

Parameters

- **key** – 128 bit MS byte first key
- **nonce** – 13 byte MS byte first nonce
- **enc_data** – Encrypted data
- **len** – Length of the encrypted data
- **aad** – Additional authenticated data
- **aad_len** – Additional authenticated data length
- **plaintext** – Plaintext buffer to place result in
- **mic_size** – Size of the trailing MIC (in bytes)

Return values

- **0** – Successfully decrypted the data.

- `-EINVAL` – Invalid parameters.
- `-EBADMSG` – Authentication failed.

int **bt_ccm_encrypt**(const uint8_t key[16], uint8_t nonce[13], const uint8_t *plaintext, size_t len, const uint8_t *aad, size_t aad_len, uint8_t *enc_data, size_t mic_size)

Encrypt big-endian data with AES-CCM.

Encrypts and generates a MIC from `plaintext` with AES-CCM, as described in https://tools.ietf.org/html/rfc3610.

Places the MIC directly after the encrypted data.

Parameters

- `key` – 128 bit MS byte first key
- `nonce` – 13 byte MS byte first nonce
- `plaintext` – Plaintext buffer to encrypt
- `len` – Length of the encrypted data
- `aad` – Additional authenticated data
- `aad_len` – Additional authenticated data length
- `enc_data` – Buffer to place encrypted data in
- `mic_size` – Size of the trailing MIC (in bytes)

Return values

- `0` – Successfully encrypted the data.
- `-EINVAL` – Invalid parameters.

Other

Bluetooth Controller

API Reference

group **bt_ctrl**

Bluetooth Controller.

Functions

void **bt_ctlr_set_public_addr**(const uint8_t *addr)

Set public address for controller.

Should be called before *bt_enable()*.

Parameters

- `addr` – Public address

Universal Unique Identifiers (UUIDs)

API Reference

group `bt_uuid`

UUIDs.

Defines

`BT_UUID_SIZE_16`

Size in octets of a 16-bit UUID.

`BT_UUID_SIZE_32`

Size in octets of a 32-bit UUID.

`BT_UUID_SIZE_128`

Size in octets of a 128-bit UUID.

`BT_UUID_INIT_16`(value)

Initialize a 16-bit UUID.

Parameters

- `value` – 16-bit UUID value in host endianness.

`BT_UUID_INIT_32`(value)

Initialize a 32-bit UUID.

Parameters

- `value` – 32-bit UUID value in host endianness.

`BT_UUID_INIT_128`(value...)

Initialize a 128-bit UUID.

Parameters

- `value` – 128-bit UUID array values in little-endian format. Can be combined with *BT_UUID_128_ENCODE* to initialize a UUID from the readable form of UUIDs.

`BT_UUID_DECLARE_16`(value)

Helper to declare a 16-bit UUID inline.

Parameters

- `value` – 16-bit UUID value in host endianness.

Returns

Pointer to a generic UUID.

`BT_UUID_DECLARE_32`(value)

Helper to declare a 32-bit UUID inline.

Parameters

- `value` – 32-bit UUID value in host endianness.

Returns

Pointer to a generic UUID.

BT_UUID_DECLARE_128(value...)

Helper to declare a 128-bit UUID inline.

Parameters

- value – 128-bit UUID array values in little-endian format. Can be combined with *BT_UUID_128_ENCODE* to declare a UUID from the readable form of UUIDs.

Returns

Pointer to a generic UUID.

BT_UUID_16(__u)

Helper macro to access the 16-bit UUID from a generic UUID.

BT_UUID_32(__u)

Helper macro to access the 32-bit UUID from a generic UUID.

BT_UUID_128(__u)

Helper macro to access the 128-bit UUID from a generic UUID.

BT_UUID_128_ENCODE(w32, w1, w2, w3, w48)

Encode 128 bit UUID into array values in little-endian format.

Helper macro to initialize a 128-bit UUID array value from the readable form of UUIDs, or encode 128-bit UUID values into advertising data Can be combined with BT_UUID_DECLARE_128 to declare a 128-bit UUID.

Example of how to declare the UUID `6E400001-B5A3-F393-E0A9-E50E24DCCA9E`

```
BT_UUID_DECLARE_128(
     BT_UUID_128_ENCODE(0x6E400001, 0xB5A3, 0xF393, 0xE0A9, 0xE50E24DCCA9E))
```

Example of how to encode the UUID `6E400001-B5A3-F393-E0A9-E50E24DCCA9E` into advertising data.

```
BT_DATA_BYTES(BT_DATA_UUID128_ALL,
     BT_UUID_128_ENCODE(0x6E400001, 0xB5A3, 0xF393, 0xE0A9, 0xE50E24DCCA9E))
```

Just replace the hyphen by the comma and add `0x` prefixes.

Parameters

- w32 – First part of the UUID (32 bits)
- w1 – Second part of the UUID (16 bits)
- w2 – Third part of the UUID (16 bits)
- w3 – Fourth part of the UUID (16 bits)
- w48 – Fifth part of the UUID (48 bits)

Returns

The comma separated values for UUID 128 initializer that may be used directly as an argument for *BT_UUID_INIT_128* or *BT_UUID_DECLARE_128*

BT_UUID_16_ENCODE(w16)

Encode 16-bit UUID into array values in little-endian format.

Helper macro to encode 16-bit UUID values into advertising data.

Example of how to encode the UUID `0x180a` into advertising data.

```
BT_DATA_BYTES(BT_DATA_UUID16_ALL, BT_UUID_16_ENCODE(0x180a))
```

Parameters

- `w16` – UUID value (16-bits)

Returns
The comma separated values for UUID 16 value that may be used directly as an argument for *BT_DATA_BYTES*.

`BT_UUID_32_ENCODE`(w32)

Encode 32-bit UUID into array values in little-endian format.

Helper macro to encode 32-bit UUID values into advertising data.

Example of how to encode the UUID `0x180a01af` into advertising data.

```
BT_DATA_BYTES(BT_DATA_UUID32_ALL, BT_UUID_32_ENCODE(0x180a01af))
```

Parameters

- `w32` – UUID value (32-bits)

Returns
The comma separated values for UUID 32 value that may be used directly as an argument for *BT_DATA_BYTES*.

`BT_UUID_STR_LEN`

Recommended length of user string buffer for Bluetooth UUID.

The recommended length guarantee the output of UUID conversion will not lose valuable information about the UUID being processed. If the length of the UUID is known the string can be shorter.

`BT_UUID_GAP_VAL`

Generic Access UUID value.

`BT_UUID_GAP`

Generic Access.

`BT_UUID_GATT_VAL`

Generic attribute UUID value.

`BT_UUID_GATT`

Generic Attribute.

`BT_UUID_IAS_VAL`

Immediate Alert Service UUID value.

`BT_UUID_IAS`

Immediate Alert Service.

`BT_UUID_LLS_VAL`

Link Loss Service UUID value.

`BT_UUID_LLS`

Link Loss Service.

BT_UUID_TPS_VAL
: Tx Power Service UUID value.

BT_UUID_TPS
: Tx Power Service.

BT_UUID_CTS_VAL
: Current Time Service UUID value.

BT_UUID_CTS
: Current Time Service.

BT_UUID_RTUS_VAL
: Reference Time Update Service UUID value.

BT_UUID_RTUS
: Reference Time Update Service.

BT_UUID_NDSTS_VAL
: Next DST Change Service UUID value.

BT_UUID_NDSTS
: Next DST Change Service.

BT_UUID_GS_VAL
: Glucose Service UUID value.

BT_UUID_GS
: Glucose Service.

BT_UUID_HTS_VAL
: Health Thermometer Service UUID value.

BT_UUID_HTS
: Health Thermometer Service.

BT_UUID_DIS_VAL
: Device Information Service UUID value.

BT_UUID_DIS
: Device Information Service.

BT_UUID_NAS_VAL
: Network Availability Service UUID value.

BT_UUID_NAS
: Network Availability Service.

BT_UUID_WDS_VAL
: Watchdog Service UUID value.

BT_UUID_WDS
: Watchdog Service.

BT_UUID_HRS_VAL
: Heart Rate Service UUID value.

BT_UUID_HRS
: Heart Rate Service.

BT_UUID_PAS_VAL
: Phone Alert Service UUID value.

BT_UUID_PAS
: Phone Alert Service.

BT_UUID_BAS_VAL
: Battery Service UUID value.

BT_UUID_BAS
: Battery Service.

BT_UUID_BPS_VAL
: Blood Pressure Service UUID value.

BT_UUID_BPS
: Blood Pressure Service.

BT_UUID_ANS_VAL
: Alert Notification Service UUID value.

BT_UUID_ANS
: Alert Notification Service.

BT_UUID_HIDS_VAL
: HID Service UUID value.

BT_UUID_HIDS
: HID Service.

BT_UUID_SPS_VAL
: Scan Parameters Service UUID value.

BT_UUID_SPS
: Scan Parameters Service.

`BT_UUID_RSCS_VAL`
: Running Speed and Cadence Service UUID value.

`BT_UUID_RSCS`
: Running Speed and Cadence Service.

`BT_UUID_AIOS_VAL`
: Automation IO Service UUID value.

`BT_UUID_AIOS`
: Automation IO Service.

`BT_UUID_CSC_VAL`
: Cycling Speed and Cadence Service UUID value.

`BT_UUID_CSC`
: Cycling Speed and Cadence Service.

`BT_UUID_CPS_VAL`
: Cycling Power Service UUID value.

`BT_UUID_CPS`
: Cycling Power Service.

`BT_UUID_LNS_VAL`
: Location and Navigation Service UUID value.

`BT_UUID_LNS`
: Location and Navigation Service.

`BT_UUID_ESS_VAL`
: Environmental Sensing Service UUID value.

`BT_UUID_ESS`
: Environmental Sensing Service.

`BT_UUID_BCS_VAL`
: Body Composition Service UUID value.

`BT_UUID_BCS`
: Body Composition Service.

`BT_UUID_UDS_VAL`
: User Data Service UUID value.

`BT_UUID_UDS`
: User Data Service.

`BT_UUID_WSS_VAL`
: Weight Scale Service UUID value.

`BT_UUID_WSS`
: Weight Scale Service.

`BT_UUID_BMS_VAL`
: Bond Management Service UUID value.

`BT_UUID_BMS`
: Bond Management Service.

`BT_UUID_CGMS_VAL`
: Continuous Glucose Monitoring Service UUID value.

`BT_UUID_CGMS`
: Continuous Glucose Monitoring Service.

`BT_UUID_IPSS_VAL`
: IP Support Service UUID value.

`BT_UUID_IPSS`
: IP Support Service.

`BT_UUID_IPS_VAL`
: Indoor Positioning Service UUID value.

`BT_UUID_IPS`
: Indoor Positioning Service.

`BT_UUID_POS_VAL`
: Pulse Oximeter Service UUID value.

`BT_UUID_POS`
: Pulse Oximeter Service.

`BT_UUID_HPS_VAL`
: HTTP Proxy Service UUID value.

`BT_UUID_HPS`
: HTTP Proxy Service.

`BT_UUID_TDS_VAL`
: Transport Discovery Service UUID value.

`BT_UUID_TDS`
: Transport Discovery Service.

BT_UUID_OTS_VAL
: Object Transfer Service UUID value.

BT_UUID_OTS
: Object Transfer Service.

BT_UUID_FMS_VAL
: Fitness Machine Service UUID value.

BT_UUID_FMS
: Fitness Machine Service.

BT_UUID_MESH_PROV_VAL
: Mesh Provisioning Service UUID value.

BT_UUID_MESH_PROV
: Mesh Provisioning Service.

BT_UUID_MESH_PROXY_VAL
: Mesh Proxy Service UUID value.

BT_UUID_MESH_PROXY
: Mesh Proxy Service.

BT_UUID_MESH_PROXY_SOLICITATION_VAL
: Proxy Solicitation UUID value.

BT_UUID_RCSRV_VAL
: Reconnection Configuration Service UUID value.

BT_UUID_RCSRV
: Reconnection Configuration Service.

BT_UUID_IDS_VAL
: Insulin Delivery Service UUID value.

BT_UUID_IDS
: Insulin Delivery Service.

BT_UUID_BSS_VAL
: Binary Sensor Service UUID value.

BT_UUID_BSS
: Binary Sensor Service.

BT_UUID_ECS_VAL
: Emergency Configuration Service UUID value.

BT_UUID_ECS
: Emergency Configuration Service.

BT_UUID_ACLS_VAL
: Authorization Control Service UUID value.

BT_UUID_ACLS
: Authorization Control Service.

BT_UUID_PAMS_VAL
: Physical Activity Monitor Service UUID value.

BT_UUID_PAMS
: Physical Activity Monitor Service.

BT_UUID_AICS_VAL
: Audio Input Control Service UUID value.

BT_UUID_AICS
: Audio Input Control Service.

BT_UUID_VCS_VAL
: Volume Control Service UUID value.

BT_UUID_VCS
: Volume Control Service.

BT_UUID_VOCS_VAL
: Volume Offset Control Service UUID value.

BT_UUID_VOCS
: Volume Offset Control Service.

BT_UUID_CSIS_VAL
: Coordinated Set Identification Service UUID value.

BT_UUID_CSIS
: Coordinated Set Identification Service.

BT_UUID_DTS_VAL
: Device Time Service UUID value.

BT_UUID_DTS
: Device Time Service.

BT_UUID_MCS_VAL
: Media Control Service UUID value.

BT_UUID_MCS
: Media Control Service.

BT_UUID_GMCS_VAL
: Generic Media Control Service UUID value.

BT_UUID_GMCS
: Generic Media Control Service.

BT_UUID_CTES_VAL
: Constant Tone Extension Service UUID value.

BT_UUID_CTES
: Constant Tone Extension Service.

BT_UUID_TBS_VAL
: Telephone Bearer Service UUID value.

BT_UUID_TBS
: Telephone Bearer Service.

BT_UUID_GTBS_VAL
: Generic Telephone Bearer Service UUID value.

BT_UUID_GTBS
: Generic Telephone Bearer Service.

BT_UUID_MICS_VAL
: Microphone Control Service UUID value.

BT_UUID_MICS
: Microphone Control Service.

BT_UUID_ASCS_VAL
: Audio Stream Control Service UUID value.

BT_UUID_ASCS
: Audio Stream Control Service.

BT_UUID_BASS_VAL
: Broadcast Audio Scan Service UUID value.

BT_UUID_BASS
: Broadcast Audio Scan Service.

BT_UUID_PACS_VAL
: Published Audio Capabilities Service UUID value.

BT_UUID_PACS
: Published Audio Capabilities Service.

BT_UUID_BASIC_AUDIO_VAL
: Basic Audio Announcement Service UUID value.

BT_UUID_BASIC_AUDIO
: Basic Audio Announcement Service.

BT_UUID_BROADCAST_AUDIO_VAL
: Broadcast Audio Announcement Service UUID value.

BT_UUID_BROADCAST_AUDIO
: Broadcast Audio Announcement Service.

BT_UUID_CAS_VAL
: Common Audio Service UUID value.

BT_UUID_CAS
: Common Audio Service.

BT_UUID_HAS_VAL
: Hearing Access Service UUID value.

BT_UUID_HAS
: Hearing Access Service.

BT_UUID_TMAS_VAL
: Telephony and Media Audio Service UUID value.

BT_UUID_TMAS
: Telephony and Media Audio Service.

BT_UUID_PBA_VAL
: Public Broadcast Announcement Service UUID value.

BT_UUID_PBA
: Public Broadcast Announcement Service.

BT_UUID_GATT_PRIMARY_VAL
: GATT Primary Service UUID value.

BT_UUID_GATT_PRIMARY
: GATT Primary Service.

BT_UUID_GATT_SECONDARY_VAL
: GATT Secondary Service UUID value.

BT_UUID_GATT_SECONDARY
: GATT Secondary Service.

BT_UUID_GATT_INCLUDE_VAL
: GATT Include Service UUID value.

BT_UUID_GATT_INCLUDE
: GATT Include Service.

BT_UUID_GATT_CHRC_VAL
: GATT Characteristic UUID value.

BT_UUID_GATT_CHRC
: GATT Characteristic.

BT_UUID_GATT_CEP_VAL
: GATT Characteristic Extended Properties UUID value.

BT_UUID_GATT_CEP
: GATT Characteristic Extended Properties.

BT_UUID_GATT_CUD_VAL
: GATT Characteristic User Description UUID value.

BT_UUID_GATT_CUD
: GATT Characteristic User Description.

BT_UUID_GATT_CCC_VAL
: GATT Client Characteristic Configuration UUID value.

BT_UUID_GATT_CCC
: GATT Client Characteristic Configuration.

BT_UUID_GATT_SCC_VAL
: GATT Server Characteristic Configuration UUID value.

BT_UUID_GATT_SCC
: GATT Server Characteristic Configuration.

BT_UUID_GATT_CPF_VAL
: GATT Characteristic Presentation Format UUID value.

BT_UUID_GATT_CPF
: GATT Characteristic Presentation Format.

BT_UUID_GATT_CAF_VAL
: GATT Characteristic Aggregated Format UUID value.

BT_UUID_GATT_CAF
: GATT Characteristic Aggregated Format.

BT_UUID_VALID_RANGE_VAL
: Valid Range Descriptor UUID value.

BT_UUID_VALID_RANGE
: Valid Range Descriptor.

BT_UUID_HIDS_EXT_REPORT_VAL
: HID External Report Descriptor UUID value.

BT_UUID_HIDS_EXT_REPORT
: HID External Report Descriptor.

BT_UUID_HIDS_REPORT_REF_VAL
: HID Report Reference Descriptor UUID value.

BT_UUID_HIDS_REPORT_REF
: HID Report Reference Descriptor.

BT_UUID_VAL_TRIGGER_SETTING_VAL
: Value Trigger Setting Descriptor UUID value.

BT_UUID_VAL_TRIGGER_SETTING
: Value Trigger Setting Descriptor.

BT_UUID_ES_CONFIGURATION_VAL
: Environmental Sensing Configuration Descriptor UUID value.

BT_UUID_ES_CONFIGURATION
: Environmental Sensing Configuration Descriptor.

BT_UUID_ES_MEASUREMENT_VAL
: Environmental Sensing Measurement Descriptor UUID value.

BT_UUID_ES_MEASUREMENT
: Environmental Sensing Measurement Descriptor.

BT_UUID_ES_TRIGGER_SETTING_VAL
: Environmental Sensing Trigger Setting Descriptor UUID value.

BT_UUID_ES_TRIGGER_SETTING
: Environmental Sensing Trigger Setting Descriptor.

BT_UUID_TM_TRIGGER_SETTING_VAL
: Time Trigger Setting Descriptor UUID value.

BT_UUID_TM_TRIGGER_SETTING
: Time Trigger Setting Descriptor.

BT_UUID_GAP_DEVICE_NAME_VAL
: GAP Characteristic Device Name UUID value.

BT_UUID_GAP_DEVICE_NAME
: GAP Characteristic Device Name.

BT_UUID_GAP_APPEARANCE_VAL
: GAP Characteristic Appearance UUID value.

BT_UUID_GAP_APPEARANCE
: GAP Characteristic Appearance.

BT_UUID_GAP_PPF_VAL
: GAP Characteristic Peripheral Privacy Flag UUID value.

BT_UUID_GAP_PPF
: GAP Characteristic Peripheral Privacy Flag.

BT_UUID_GAP_RA_VAL
: GAP Characteristic Reconnection Address UUID value.

BT_UUID_GAP_RA
: GAP Characteristic Reconnection Address.

BT_UUID_GAP_PPCP_VAL
: GAP Characteristic Peripheral Preferred Connection Parameters UUID value.

BT_UUID_GAP_PPCP
: GAP Characteristic Peripheral Preferred Connection Parameters.

BT_UUID_GATT_SC_VAL
: GATT Characteristic Service Changed UUID value.

BT_UUID_GATT_SC
: GATT Characteristic Service Changed.

BT_UUID_ALERT_LEVEL_VAL
: GATT Characteristic Alert Level UUID value.

BT_UUID_ALERT_LEVEL
: GATT Characteristic Alert Level.

BT_UUID_TPS_TX_POWER_LEVEL_VAL
: TPS Characteristic Tx Power Level UUID value.

BT_UUID_TPS_TX_POWER_LEVEL
: TPS Characteristic Tx Power Level.

BT_UUID_GATT_DT_VAL
: GATT Characteristic Date Time UUID value.

BT_UUID_GATT_DT
: GATT Characteristic Date Time.

BT_UUID_GATT_DW_VAL
: GATT Characteristic Day of Week UUID value.

BT_UUID_GATT_DW
: GATT Characteristic Day of Week.

BT_UUID_GATT_DDT_VAL
: GATT Characteristic Day Date Time UUID value.

BT_UUID_GATT_DDT
: GATT Characteristic Day Date Time.

BT_UUID_GATT_ET256_VAL
: GATT Characteristic Exact Time 256 UUID value.

BT_UUID_GATT_ET256
: GATT Characteristic Exact Time 256.

BT_UUID_GATT_DST_VAL
: GATT Characteristic DST Offset UUID value.

BT_UUID_GATT_DST
: GATT Characteristic DST Offset.

BT_UUID_GATT_TZ_VAL
: GATT Characteristic Time Zone UUID value.

BT_UUID_GATT_TZ
: GATT Characteristic Time Zone.

BT_UUID_GATT_LTI_VAL
: GATT Characteristic Local Time Information UUID value.

BT_UUID_GATT_LTI
: GATT Characteristic Local Time Information.

BT_UUID_GATT_TDST_VAL
: GATT Characteristic Time with DST UUID value.

BT_UUID_GATT_TDST
: GATT Characteristic Time with DST.

BT_UUID_GATT_TA_VAL
: GATT Characteristic Time Accuracy UUID value.

BT_UUID_GATT_TA
: GATT Characteristic Time Accuracy.

BT_UUID_GATT_TS_VAL
: GATT Characteristic Time Source UUID value.

BT_UUID_GATT_TS
: GATT Characteristic Time Source.

BT_UUID_GATT_RTI_VAL
: GATT Characteristic Reference Time Information UUID value.

BT_UUID_GATT_RTI
: GATT Characteristic Reference Time Information.

BT_UUID_GATT_TUCP_VAL
: GATT Characteristic Time Update Control Point UUID value.

BT_UUID_GATT_TUCP
: GATT Characteristic Time Update Control Point.

BT_UUID_GATT_TUS_VAL
: GATT Characteristic Time Update State UUID value.

BT_UUID_GATT_TUS
: GATT Characteristic Time Update State.

BT_UUID_GATT_GM_VAL
: GATT Characteristic Glucose Measurement UUID value.

BT_UUID_GATT_GM
: GATT Characteristic Glucose Measurement.

BT_UUID_BAS_BATTERY_LEVEL_VAL
: BAS Characteristic Battery Level UUID value.

BT_UUID_BAS_BATTERY_LEVEL
: BAS Characteristic Battery Level.

BT_UUID_BAS_BATTERY_POWER_STATE_VAL
: BAS Characteristic Battery Power State UUID value.

BT_UUID_BAS_BATTERY_POWER_STATE
: BAS Characteristic Battery Power State.

BT_UUID_BAS_BATTERY_LEVEL_STATE_VAL
: BAS Characteristic Battery Level StateUUID value.

BT_UUID_BAS_BATTERY_LEVEL_STATE
: BAS Characteristic Battery Level State.

BT_UUID_HTS_MEASUREMENT_VAL
: HTS Characteristic Temperature Measurement UUID value.

BT_UUID_HTS_MEASUREMENT
: HTS Characteristic Temperature Measurement Value.

BT_UUID_HTS_TEMP_TYP_VAL
: HTS Characteristic Temperature Type UUID value.

BT_UUID_HTS_TEMP_TYP
: HTS Characteristic Temperature Type.

BT_UUID_HTS_TEMP_INT_VAL
: HTS Characteristic Intermediate Temperature UUID value.

BT_UUID_HTS_TEMP_INT
: HTS Characteristic Intermediate Temperature.

BT_UUID_HTS_TEMP_C_VAL
: HTS Characteristic Temperature Celsius UUID value.

BT_UUID_HTS_TEMP_C
: HTS Characteristic Temperature Celsius.

BT_UUID_HTS_TEMP_F_VAL
: HTS Characteristic Temperature Fahrenheit UUID value.

BT_UUID_HTS_TEMP_F
: HTS Characteristic Temperature Fahrenheit.

BT_UUID_HTS_INTERVAL_VAL
: HTS Characteristic Measurement Interval UUID value.

BT_UUID_HTS_INTERVAL
: HTS Characteristic Measurement Interval.

BT_UUID_HIDS_BOOT_KB_IN_REPORT_VAL
: HID Characteristic Boot Keyboard Input Report UUID value.

BT_UUID_HIDS_BOOT_KB_IN_REPORT
: HID Characteristic Boot Keyboard Input Report.

BT_UUID_DIS_SYSTEM_ID_VAL
: DIS Characteristic System ID UUID value.

BT_UUID_DIS_SYSTEM_ID
: DIS Characteristic System ID.

BT_UUID_DIS_MODEL_NUMBER_VAL
: DIS Characteristic Model Number String UUID value.

BT_UUID_DIS_MODEL_NUMBER
: DIS Characteristic Model Number String.

BT_UUID_DIS_SERIAL_NUMBER_VAL
: DIS Characteristic Serial Number String UUID value.

BT_UUID_DIS_SERIAL_NUMBER
: DIS Characteristic Serial Number String.

BT_UUID_DIS_FIRMWARE_REVISION_VAL
: DIS Characteristic Firmware Revision String UUID value.

BT_UUID_DIS_FIRMWARE_REVISION
: DIS Characteristic Firmware Revision String.

BT_UUID_DIS_HARDWARE_REVISION_VAL
: DIS Characteristic Hardware Revision String UUID value.

BT_UUID_DIS_HARDWARE_REVISION
: DIS Characteristic Hardware Revision String.

BT_UUID_DIS_SOFTWARE_REVISION_VAL
: DIS Characteristic Software Revision String UUID value.

BT_UUID_DIS_SOFTWARE_REVISION
: DIS Characteristic Software Revision String.

BT_UUID_DIS_MANUFACTURER_NAME_VAL
: DIS Characteristic Manufacturer Name String UUID Value.

BT_UUID_DIS_MANUFACTURER_NAME
: DIS Characteristic Manufacturer Name String.

BT_UUID_GATT_IEEE_RCDL_VAL
: GATT Characteristic IEEE Regulatory Certification Data List UUID Value.

BT_UUID_GATT_IEEE_RCDL
: GATT Characteristic IEEE Regulatory Certification Data List.

BT_UUID_CTS_CURRENT_TIME_VAL
: CTS Characteristic Current Time UUID value.

BT_UUID_CTS_CURRENT_TIME
: CTS Characteristic Current Time.

BT_UUID_MAGN_DECLINATION_VAL
: Magnetic Declination Characteristic UUID value.

BT_UUID_MAGN_DECLINATION
: Magnetic Declination Characteristic.

BT_UUID_GATT_LLAT_VAL
: GATT Characteristic Legacy Latitude UUID Value.

BT_UUID_GATT_LLAT
: GATT Characteristic Legacy Latitude.

BT_UUID_GATT_LLON_VAL
: GATT Characteristic Legacy Longitude UUID Value.

BT_UUID_GATT_LLON
: GATT Characteristic Legacy Longitude.

BT_UUID_GATT_POS_2D_VAL
: GATT Characteristic Position 2D UUID Value.

BT_UUID_GATT_POS_2D
: GATT Characteristic Position 2D.

BT_UUID_GATT_POS_3D_VAL
: GATT Characteristic Position 3D UUID Value.

BT_UUID_GATT_POS_3D
: GATT Characteristic Position 3D.

BT_UUID_GATT_SR_VAL
: GATT Characteristic Scan Refresh UUID Value.

BT_UUID_GATT_SR
: GATT Characteristic Scan Refresh.

BT_UUID_HIDS_BOOT_KB_OUT_REPORT_VAL
: HID Boot Keyboard Output Report Characteristic UUID value.

BT_UUID_HIDS_BOOT_KB_OUT_REPORT
: HID Boot Keyboard Output Report Characteristic.

BT_UUID_HIDS_BOOT_MOUSE_IN_REPORT_VAL
: HID Boot Mouse Input Report Characteristic UUID value.

BT_UUID_HIDS_BOOT_MOUSE_IN_REPORT
: HID Boot Mouse Input Report Characteristic.

BT_UUID_GATT_GMC_VAL
: GATT Characteristic Glucose Measurement Context UUID Value.

BT_UUID_GATT_GMC
: GATT Characteristic Glucose Measurement Context.

BT_UUID_GATT_BPM_VAL
: GATT Characteristic Blood Pressure Measurement UUID Value.

BT_UUID_GATT_BPM
: GATT Characteristic Blood Pressure Measurement.

BT_UUID_GATT_ICP_VAL
: GATT Characteristic Intermediate Cuff Pressure UUID Value.

BT_UUID_GATT_ICP
: GATT Characteristic Intermediate Cuff Pressure.

BT_UUID_HRS_MEASUREMENT_VAL
: HRS Characteristic Measurement Interval UUID value.

BT_UUID_HRS_MEASUREMENT
: HRS Characteristic Measurement Interval.

BT_UUID_HRS_BODY_SENSOR_VAL
: HRS Characteristic Body Sensor Location.

BT_UUID_HRS_BODY_SENSOR
: HRS Characteristic Control Point.

BT_UUID_HRS_CONTROL_POINT_VAL
: HRS Characteristic Control Point UUID value.

BT_UUID_HRS_CONTROL_POINT
: HRS Characteristic Control Point.

BT_UUID_GATT_REM_VAL
: GATT Characteristic Removable UUID Value.

BT_UUID_GATT_REM
: GATT Characteristic Removable.

BT_UUID_GATT_SRVREQ_VAL
: GATT Characteristic Service Required UUID Value.

BT_UUID_GATT_SRVREQ
: GATT Characteristic Service Required.

BT_UUID_GATT_SC_TEMP_C_VAL
: GATT Characteristic Scientific Temperature in Celsius UUID Value.

BT_UUID_GATT_SC_TEMP_C
: GATT Characteristic Scientific Temperature in Celsius.

BT_UUID_GATT_STRING_VAL
: GATT Characteristic String UUID Value.

BT_UUID_GATT_STRING
: GATT Characteristic String.

BT_UUID_GATT_NETA_VAL
: GATT Characteristic Network Availability UUID Value.

BT_UUID_GATT_NETA
: GATT Characteristic Network Availability.

BT_UUID_GATT_ALRTS_VAL
: GATT Characteristic Alert Status UUID Value.

BT_UUID_GATT_ALRTS
: GATT Characteristic Alert Status.

BT_UUID_GATT_RCP_VAL
: GATT Characteristic Ringer Control Point UUID Value.

BT_UUID_GATT_RCP
: GATT Characteristic Ringer Control Point.

BT_UUID_GATT_RS_VAL
: GATT Characteristic Ringer Setting UUID Value.

BT_UUID_GATT_RS
: GATT Characteristic Ringer Setting.

BT_UUID_GATT_ALRTCID_MASK_VAL
: GATT Characteristic Alert Category ID Bit Mask UUID Value.

BT_UUID_GATT_ALRTCID_MASK
: GATT Characteristic Alert Category ID Bit Mask.

BT_UUID_GATT_ALRTCID_VAL
: GATT Characteristic Alert Category ID UUID Value.

BT_UUID_GATT_ALRTCID
: GATT Characteristic Alert Category ID.

BT_UUID_GATT_ALRTNCP_VAL
: GATT Characteristic Alert Notification Control Point Value.

BT_UUID_GATT_ALRTNCP
: GATT Characteristic Alert Notification Control Point.

BT_UUID_GATT_UALRTS_VAL
: GATT Characteristic Unread Alert Status UUID Value.

BT_UUID_GATT_UALRTS
: GATT Characteristic Unread Alert Status.

BT_UUID_GATT_NALRT_VAL
: GATT Characteristic New Alert UUID Value.

BT_UUID_GATT_NALRT
: GATT Characteristic New Alert.

BT_UUID_GATT_SNALRTC_VAL
: GATT Characteristic Supported New Alert Category UUID Value.

BT_UUID_GATT_SNALRTC
: GATT Characteristic Supported New Alert Category.

BT_UUID_GATT_SUALRTC_VAL
: GATT Characteristic Supported Unread Alert Category UUID Value.

BT_UUID_GATT_SUALRTC
: GATT Characteristic Supported Unread Alert Category.

BT_UUID_GATT_BPF_VAL
: GATT Characteristic Blood Pressure Feature UUID Value.

BT_UUID_GATT_BPF
: GATT Characteristic Blood Pressure Feature.

BT_UUID_HIDS_INFO_VAL
: HID Information Characteristic UUID value.

`BT_UUID_HIDS_INFO`
: HID Information Characteristic.

`BT_UUID_HIDS_REPORT_MAP_VAL`
: HID Report Map Characteristic UUID value.

`BT_UUID_HIDS_REPORT_MAP`
: HID Report Map Characteristic.

`BT_UUID_HIDS_CTRL_POINT_VAL`
: HID Control Point Characteristic UUID value.

`BT_UUID_HIDS_CTRL_POINT`
: HID Control Point Characteristic.

`BT_UUID_HIDS_REPORT_VAL`
: HID Report Characteristic UUID value.

`BT_UUID_HIDS_REPORT`
: HID Report Characteristic.

`BT_UUID_HIDS_PROTOCOL_MODE_VAL`
: HID Protocol Mode Characteristic UUID value.

`BT_UUID_HIDS_PROTOCOL_MODE`
: HID Protocol Mode Characteristic.

`BT_UUID_GATT_SIW_VAL`
: GATT Characteristic Scan Interval Windows UUID Value.

`BT_UUID_GATT_SIW`
: GATT Characteristic Scan Interval Windows.

`BT_UUID_DIS_PNP_ID_VAL`
: DIS Characteristic PnP ID UUID value.

`BT_UUID_DIS_PNP_ID`
: DIS Characteristic PnP ID.

`BT_UUID_GATT_GF_VAL`
: GATT Characteristic Glucose Feature UUID Value.

`BT_UUID_GATT_GF`
: GATT Characteristic Glucose Feature.

`BT_UUID_RECORD_ACCESS_CONTROL_POINT_VAL`
: Record Access Control Point Characteristic value.

BT_UUID_RECORD_ACCESS_CONTROL_POINT
: Record Access Control Point.

BT_UUID_RSC_MEASUREMENT_VAL
: RSC Measurement Characteristic UUID value.

BT_UUID_RSC_MEASUREMENT
: RSC Measurement Characteristic.

BT_UUID_RSC_FEATURE_VAL
: RSC Feature Characteristic UUID value.

BT_UUID_RSC_FEATURE
: RSC Feature Characteristic.

BT_UUID_SC_CONTROL_POINT_VAL
: SC Control Point Characteristic UUID value.

BT_UUID_SC_CONTROL_POINT
: SC Control Point Characteristic.

BT_UUID_GATT_DI_VAL
: GATT Characteristic Digital Input UUID Value.

BT_UUID_GATT_DI
: GATT Characteristic Digital Input.

BT_UUID_GATT_DO_VAL
: GATT Characteristic Digital Output UUID Value.

BT_UUID_GATT_DO
: GATT Characteristic Digital Output.

BT_UUID_GATT_AI_VAL
: GATT Characteristic Analog Input UUID Value.

BT_UUID_GATT_AI
: GATT Characteristic Analog Input.

BT_UUID_GATT_AO_VAL
: GATT Characteristic Analog Output UUID Value.

BT_UUID_GATT_AO
: GATT Characteristic Analog Output.

BT_UUID_GATT_AGGR_VAL
: GATT Characteristic Aggregate UUID Value.

`BT_UUID_GATT_AGGR`
: GATT Characteristic Aggregate.

`BT_UUID_CSC_MEASUREMENT_VAL`
: CSC Measurement Characteristic UUID value.

`BT_UUID_CSC_MEASUREMENT`
: CSC Measurement Characteristic.

`BT_UUID_CSC_FEATURE_VAL`
: CSC Feature Characteristic UUID value.

`BT_UUID_CSC_FEATURE`
: CSC Feature Characteristic.

`BT_UUID_SENSOR_LOCATION_VAL`
: Sensor Location Characteristic UUID value.

`BT_UUID_SENSOR_LOCATION`
: Sensor Location Characteristic.

`BT_UUID_GATT_PLX_SCM_VAL`
: GATT Characteristic PLX Spot-Check Measurement UUID Value.

`BT_UUID_GATT_PLX_SCM`
: GATT Characteristic PLX Spot-Check Measurement.

`BT_UUID_GATT_PLX_CM_VAL`
: GATT Characteristic PLX Continuous Measurement UUID Value.

`BT_UUID_GATT_PLX_CM`
: GATT Characteristic PLX Continuous Measurement.

`BT_UUID_GATT_PLX_F_VAL`
: GATT Characteristic PLX Features UUID Value.

`BT_UUID_GATT_PLX_F`
: GATT Characteristic PLX Features.

`BT_UUID_GATT_POPE_VAL`
: GATT Characteristic Pulse Oximetry Pulastile Event UUID Value.

`BT_UUID_GATT_POPE`
: GATT Characteristic Pulse Oximetry Pulsatile Event.

`BT_UUID_GATT_POCP_VAL`
: GATT Characteristic Pulse Oximetry Control Point UUID Value.

BT_UUID_GATT_POCP
: GATT Characteristic Pulse Oximetry Control Point.

BT_UUID_GATT_CPS_CPM_VAL
: GATT Characteristic Cycling Power Measurement UUID Value.

BT_UUID_GATT_CPS_CPM
: GATT Characteristic Cycling Power Measurement.

BT_UUID_GATT_CPS_CPV_VAL
: GATT Characteristic Cycling Power Vector UUID Value.

BT_UUID_GATT_CPS_CPV
: GATT Characteristic Cycling Power Vector.

BT_UUID_GATT_CPS_CPF_VAL
: GATT Characteristic Cycling Power Feature UUID Value.

BT_UUID_GATT_CPS_CPF
: GATT Characteristic Cycling Power Feature.

BT_UUID_GATT_CPS_CPCP_VAL
: GATT Characteristic Cycling Power Control Point UUID Value.

BT_UUID_GATT_CPS_CPCP
: GATT Characteristic Cycling Power Control Point.

BT_UUID_GATT_LOC_SPD_VAL
: GATT Characteristic Location and Speed UUID Value.

BT_UUID_GATT_LOC_SPD
: GATT Characteristic Location and Speed.

BT_UUID_GATT_NAV_VAL
: GATT Characteristic Navigation UUID Value.

BT_UUID_GATT_NAV
: GATT Characteristic Navigation.

BT_UUID_GATT_PQ_VAL
: GATT Characteristic Position Quality UUID Value.

BT_UUID_GATT_PQ
: GATT Characteristic Position Quality.

BT_UUID_GATT_LNF_VAL
: GATT Characteristic LN Feature UUID Value.

`BT_UUID_GATT_LNF`
: GATT Characteristic LN Feature.

`BT_UUID_GATT_LNCP_VAL`
: GATT Characteristic LN Control Point UUID Value.

`BT_UUID_GATT_LNCP`
: GATT Characteristic LN Control Point.

`BT_UUID_ELEVATION_VAL`
: Elevation Characteristic UUID value.

`BT_UUID_ELEVATION`
: Elevation Characteristic.

`BT_UUID_PRESSURE_VAL`
: Pressure Characteristic UUID value.

`BT_UUID_PRESSURE`
: Pressure Characteristic.

`BT_UUID_TEMPERATURE_VAL`
: Temperature Characteristic UUID value.

`BT_UUID_TEMPERATURE`
: Temperature Characteristic.

`BT_UUID_HUMIDITY_VAL`
: Humidity Characteristic UUID value.

`BT_UUID_HUMIDITY`
: Humidity Characteristic.

`BT_UUID_TRUE_WIND_SPEED_VAL`
: True Wind Speed Characteristic UUID value.

`BT_UUID_TRUE_WIND_SPEED`
: True Wind Speed Characteristic.

`BT_UUID_TRUE_WIND_DIR_VAL`
: True Wind Direction Characteristic UUID value.

`BT_UUID_TRUE_WIND_DIR`
: True Wind Direction Characteristic.

`BT_UUID_APPARENT_WIND_SPEED_VAL`
: Apparent Wind Speed Characteristic UUID value.

BT_UUID_APPARENT_WIND_SPEED
: Apparent Wind Speed Characteristic.

BT_UUID_APPARENT_WIND_DIR_VAL
: Apparent Wind Direction Characteristic UUID value.

BT_UUID_APPARENT_WIND_DIR
: Apparent Wind Direction Characteristic.

BT_UUID_GUST_FACTOR_VAL
: Gust Factor Characteristic UUID value.

BT_UUID_GUST_FACTOR
: Gust Factor Characteristic.

BT_UUID_POLLEN_CONCENTRATION_VAL
: Pollen Concentration Characteristic UUID value.

BT_UUID_POLLEN_CONCENTRATION
: Pollen Concentration Characteristic.

BT_UUID_UV_INDEX_VAL
: UV Index Characteristic UUID value.

BT_UUID_UV_INDEX
: UV Index Characteristic.

BT_UUID_IRRADIANCE_VAL
: Irradiance Characteristic UUID value.

BT_UUID_IRRADIANCE
: Irradiance Characteristic.

BT_UUID_RAINFALL_VAL
: Rainfall Characteristic UUID value.

BT_UUID_RAINFALL
: Rainfall Characteristic.

BT_UUID_WIND_CHILL_VAL
: Wind Chill Characteristic UUID value.

BT_UUID_WIND_CHILL
: Wind Chill Characteristic.

BT_UUID_HEAT_INDEX_VAL
: Heat Index Characteristic UUID value.

BT_UUID_HEAT_INDEX
: Heat Index Characteristic.

BT_UUID_DEW_POINT_VAL
: Dew Point Characteristic UUID value.

BT_UUID_DEW_POINT
: Dew Point Characteristic.

BT_UUID_GATT_TREND_VAL
: GATT Characteristic Trend UUID Value.

BT_UUID_GATT_TREND
: GATT Characteristic Trend.

BT_UUID_DESC_VALUE_CHANGED_VAL
: Descriptor Value Changed Characteristic UUID value.

BT_UUID_DESC_VALUE_CHANGED
: Descriptor Value Changed Characteristic.

BT_UUID_GATT_AEHRLL_VAL
: GATT Characteristic Aerobic Heart Rate Low Limit UUID Value.

BT_UUID_GATT_AEHRLL
: GATT Characteristic Aerobic Heart Rate Lower Limit.

BT_UUID_GATT_AETHR_VAL
: GATT Characteristic Aerobic Threshold UUID Value.

BT_UUID_GATT_AETHR
: GATT Characteristic Aerobic Threshold.

BT_UUID_GATT_AGE_VAL
: GATT Characteristic Age UUID Value.

BT_UUID_GATT_AGE
: GATT Characteristic Age.

BT_UUID_GATT_ANHRLL_VAL
: GATT Characteristic Anaerobic Heart Rate Lower Limit UUID Value.

BT_UUID_GATT_ANHRLL
: GATT Characteristic Anaerobic Heart Rate Lower Limit.

BT_UUID_GATT_ANHRUL_VAL
: GATT Characteristic Anaerobic Heart Rate Upper Limit UUID Value.

BT_UUID_GATT_ANHRUL
: GATT Characteristic Anaerobic Heart Rate Upper Limit.

BT_UUID_GATT_ANTHR_VAL
: GATT Characteristic Anaerobic Threshold UUID Value.

BT_UUID_GATT_ANTHR
: GATT Characteristic Anaerobic Threshold.

BT_UUID_GATT_AEHRUL_VAL
: GATT Characteristic Aerobic Heart Rate Upper Limit UUID Value.

BT_UUID_GATT_AEHRUL
: GATT Characteristic Aerobic Heart Rate Upper Limit.

BT_UUID_GATT_DATE_BIRTH_VAL
: GATT Characteristic Date of Birth UUID Value.

BT_UUID_GATT_DATE_BIRTH
: GATT Characteristic Date of Birth.

BT_UUID_GATT_DATE_THRASS_VAL
: GATT Characteristic Date of Threshold Assessment UUID Value.

BT_UUID_GATT_DATE_THRASS
: GATT Characteristic Date of Threshold Assessment.

BT_UUID_GATT_EMAIL_VAL
: GATT Characteristic Email Address UUID Value.

BT_UUID_GATT_EMAIL
: GATT Characteristic Email Address.

BT_UUID_GATT_FBHRLL_VAL
: GATT Characteristic Fat Burn Heart Rate Lower Limit UUID Value.

BT_UUID_GATT_FBHRLL
: GATT Characteristic Fat Burn Heart Rate Lower Limit.

BT_UUID_GATT_FBHRUL_VAL
: GATT Characteristic Fat Burn Heart Rate Upper Limit UUID Value.

BT_UUID_GATT_FBHRUL
: GATT Characteristic Fat Burn Heart Rate Upper Limit.

BT_UUID_GATT_FIRST_NAME_VAL
: GATT Characteristic First Name UUID Value.

BT_UUID_GATT_FIRST_NAME
: GATT Characteristic First Name.

BT_UUID_GATT_5ZHRL_VAL
: GATT Characteristic Five Zone Heart Rate Limits UUID Value.

BT_UUID_GATT_5ZHRL
: GATT Characteristic Five Zone Heart Rate Limits.

BT_UUID_GATT_GENDER_VAL
: GATT Characteristic Gender UUID Value.

BT_UUID_GATT_GENDER
: GATT Characteristic Gender.

BT_UUID_GATT_HR_MAX_VAL
: GATT Characteristic Heart Rate Max UUID Value.

BT_UUID_GATT_HR_MAX
: GATT Characteristic Heart Rate Max.

BT_UUID_GATT_HEIGHT_VAL
: GATT Characteristic Height UUID Value.

BT_UUID_GATT_HEIGHT
: GATT Characteristic Height.

BT_UUID_GATT_HC_VAL
: GATT Characteristic Hip Circumference UUID Value.

BT_UUID_GATT_HC
: GATT Characteristic Hip Circumference.

BT_UUID_GATT_LAST_NAME_VAL
: GATT Characteristic Last Name UUID Value.

BT_UUID_GATT_LAST_NAME
: GATT Characteristic Last Name.

BT_UUID_GATT_MRHR_VAL
: GATT Characteristic Maximum Recommended Heart Rate> UUID Value.

BT_UUID_GATT_MRHR
: GATT Characteristic Maximum Recommended Heart Rate.

BT_UUID_GATT_RHR_VAL
: GATT Characteristic Resting Heart Rate UUID Value.

BT_UUID_GATT_RHR
: GATT Characteristic Resting Heart Rate.

BT_UUID_GATT_AEANTHR_VAL
: GATT Characteristic Sport Type for Aerobic and Anaerobic Thresholds UUID Value.

BT_UUID_GATT_AEANTHR
: GATT Characteristic Sport Type for Aerobic and Anaerobic Threshold.

BT_UUID_GATT_3ZHRL_VAL
: GATT Characteristic Three Zone Heart Rate Limits UUID Value.

BT_UUID_GATT_3ZHRL
: GATT Characteristic Three Zone Heart Rate Limits.

BT_UUID_GATT_2ZHRL_VAL
: GATT Characteristic Two Zone Heart Rate Limits UUID Value.

BT_UUID_GATT_2ZHRL
: GATT Characteristic Two Zone Heart Rate Limits.

BT_UUID_GATT_VO2_MAX_VAL
: GATT Characteristic VO2 Max UUID Value.

BT_UUID_GATT_VO2_MAX
: GATT Characteristic VO2 Max.

BT_UUID_GATT_WC_VAL
: GATT Characteristic Waist Circumference UUID Value.

BT_UUID_GATT_WC
: GATT Characteristic Waist Circumference.

BT_UUID_GATT_WEIGHT_VAL
: GATT Characteristic Weight UUID Value.

BT_UUID_GATT_WEIGHT
: GATT Characteristic Weight.

BT_UUID_GATT_DBCHINC_VAL
: GATT Characteristic Database Change Increment UUID Value.

BT_UUID_GATT_DBCHINC
: GATT Characteristic Database Change Increment.

BT_UUID_GATT_USRIDX_VAL
: GATT Characteristic User Index UUID Value.

`BT_UUID_GATT_USRIDX`
: GATT Characteristic User Index.

`BT_UUID_GATT_BCF_VAL`
: GATT Characteristic Body Composition Feature UUID Value.

`BT_UUID_GATT_BCF`
: GATT Characteristic Body Composition Feature.

`BT_UUID_GATT_BCM_VAL`
: GATT Characteristic Body Composition Measurement UUID Value.

`BT_UUID_GATT_BCM`
: GATT Characteristic Body Composition Measurement.

`BT_UUID_GATT_WM_VAL`
: GATT Characteristic Weight Measurement UUID Value.

`BT_UUID_GATT_WM`
: GATT Characteristic Weight Measurement.

`BT_UUID_GATT_WSF_VAL`
: GATT Characteristic Weight Scale Feature UUID Value.

`BT_UUID_GATT_WSF`
: GATT Characteristic Weight Scale Feature.

`BT_UUID_GATT_USRCP_VAL`
: GATT Characteristic User Control Point UUID Value.

`BT_UUID_GATT_USRCP`
: GATT Characteristic User Control Point.

`BT_UUID_MAGN_FLUX_DENSITY_2D_VAL`
: Magnetic Flux Density - 2D Characteristic UUID value.

`BT_UUID_MAGN_FLUX_DENSITY_2D`
: Magnetic Flux Density - 2D Characteristic.

`BT_UUID_MAGN_FLUX_DENSITY_3D_VAL`
: Magnetic Flux Density - 3D Characteristic UUID value.

`BT_UUID_MAGN_FLUX_DENSITY_3D`
: Magnetic Flux Density - 3D Characteristic.

`BT_UUID_GATT_LANG_VAL`
: GATT Characteristic Language UUID Value.

BT_UUID_GATT_LANG
: GATT Characteristic Language.

BT_UUID_BAR_PRESSURE_TREND_VAL
: Barometric Pressure Trend Characteristic UUID value.

BT_UUID_BAR_PRESSURE_TREND
: Barometric Pressure Trend Characteristic.

BT_UUID_BMS_CONTROL_POINT_VAL
: Bond Management Control Point UUID value.

BT_UUID_BMS_CONTROL_POINT
: Bond Management Control Point.

BT_UUID_BMS_FEATURE_VAL
: Bond Management Feature UUID value.

BT_UUID_BMS_FEATURE
: Bond Management Feature.

BT_UUID_CENTRAL_ADDR_RES_VAL
: Central Address Resolution Characteristic UUID value.

BT_UUID_CENTRAL_ADDR_RES
: Central Address Resolution Characteristic.

BT_UUID_CGM_MEASUREMENT_VAL
: CGM Measurement Characteristic value.

BT_UUID_CGM_MEASUREMENT
: CGM Measurement Characteristic.

BT_UUID_CGM_FEATURE_VAL
: CGM Feature Characteristic value.

BT_UUID_CGM_FEATURE
: CGM Feature Characteristic.

BT_UUID_CGM_STATUS_VAL
: CGM Status Characteristic value.

BT_UUID_CGM_STATUS
: CGM Status Characteristic.

BT_UUID_CGM_SESSION_START_TIME_VAL
: CGM Session Start Time Characteristic value.

BT_UUID_CGM_SESSION_START_TIME
: CGM Session Start Time.

BT_UUID_CGM_SESSION_RUN_TIME_VAL
: CGM Session Run Time Characteristic value.

BT_UUID_CGM_SESSION_RUN_TIME
: CGM Session Run Time.

BT_UUID_CGM_SPECIFIC_OPS_CONTROL_POINT_VAL
: CGM Specific Ops Control Point Characteristic value.

BT_UUID_CGM_SPECIFIC_OPS_CONTROL_POINT
: CGM Specific Ops Control Point.

BT_UUID_GATT_IPC_VAL
: GATT Characteristic Indoor Positioning Configuration UUID Value.

BT_UUID_GATT_IPC
: GATT Characteristic Indoor Positioning Configuration.

BT_UUID_GATT_LAT_VAL
: GATT Characteristic Latitude UUID Value.

BT_UUID_GATT_LAT
: GATT Characteristic Latitude.

BT_UUID_GATT_LON_VAL
: GATT Characteristic Longitude UUID Value.

BT_UUID_GATT_LON
: GATT Characteristic Longitude.

BT_UUID_GATT_LNCOORD_VAL
: GATT Characteristic Local North Coordinate UUID Value.

BT_UUID_GATT_LNCOORD
: GATT Characteristic Local North Coordinate.

BT_UUID_GATT_LECOORD_VAL
: GATT Characteristic Local East Coordinate UUID Value.

BT_UUID_GATT_LECOORD
: GATT Characteristic Local East Coordinate.

BT_UUID_GATT_FN_VAL
: GATT Characteristic Floor Number UUID Value.

BT_UUID_GATT_FN
: GATT Characteristic Floor Number.

BT_UUID_GATT_ALT_VAL
: GATT Characteristic Altitude UUID Value.

BT_UUID_GATT_ALT
: GATT Characteristic Altitude.

BT_UUID_GATT_UNCERTAINTY_VAL
: GATT Characteristic Uncertainty UUID Value.

BT_UUID_GATT_UNCERTAINTY
: GATT Characteristic Uncertainty.

BT_UUID_GATT_LOC_NAME_VAL
: GATT Characteristic Location Name UUID Value.

BT_UUID_GATT_LOC_NAME
: GATT Characteristic Location Name.

BT_UUID_URI_VAL
: URI UUID value.

BT_UUID_URI
: URI.

BT_UUID_HTTP_HEADERS_VAL
: HTTP Headers UUID value.

BT_UUID_HTTP_HEADERS
: HTTP Headers.

BT_UUID_HTTP_STATUS_CODE_VAL
: HTTP Status Code UUID value.

BT_UUID_HTTP_STATUS_CODE
: HTTP Status Code.

BT_UUID_HTTP_ENTITY_BODY_VAL
: HTTP Entity Body UUID value.

BT_UUID_HTTP_ENTITY_BODY
: HTTP Entity Body.

BT_UUID_HTTP_CONTROL_POINT_VAL
: HTTP Control Point UUID value.

BT_UUID_HTTP_CONTROL_POINT
: HTTP Control Point.

BT_UUID_HTTPS_SECURITY_VAL
: HTTPS Security UUID value.

BT_UUID_HTTPS_SECURITY
: HTTPS Security.

BT_UUID_GATT_TDS_CP_VAL
: GATT Characteristic TDS Control Point UUID Value.

BT_UUID_GATT_TDS_CP
: GATT Characteristic TDS Control Point.

BT_UUID_OTS_FEATURE_VAL
: OTS Feature Characteristic UUID value.

BT_UUID_OTS_FEATURE
: OTS Feature Characteristic.

BT_UUID_OTS_NAME_VAL
: OTS Object Name Characteristic UUID value.

BT_UUID_OTS_NAME
: OTS Object Name Characteristic.

BT_UUID_OTS_TYPE_VAL
: OTS Object Type Characteristic UUID value.

BT_UUID_OTS_TYPE
: OTS Object Type Characteristic.

BT_UUID_OTS_SIZE_VAL
: OTS Object Size Characteristic UUID value.

BT_UUID_OTS_SIZE
: OTS Object Size Characteristic.

BT_UUID_OTS_FIRST_CREATED_VAL
: OTS Object First-Created Characteristic UUID value.

BT_UUID_OTS_FIRST_CREATED
: OTS Object First-Created Characteristic.

BT_UUID_OTS_LAST_MODIFIED_VAL
: OTS Object Last-Modified Characteristic UUI value.

BT_UUID_OTS_LAST_MODIFIED
: OTS Object Last-Modified Characteristic.

BT_UUID_OTS_ID_VAL
: OTS Object ID Characteristic UUID value.

BT_UUID_OTS_ID
: OTS Object ID Characteristic.

BT_UUID_OTS_PROPERTIES_VAL
: OTS Object Properties Characteristic UUID value.

BT_UUID_OTS_PROPERTIES
: OTS Object Properties Characteristic.

BT_UUID_OTS_ACTION_CP_VAL
: OTS Object Action Control Point Characteristic UUID value.

BT_UUID_OTS_ACTION_CP
: OTS Object Action Control Point Characteristic.

BT_UUID_OTS_LIST_CP_VAL
: OTS Object List Control Point Characteristic UUID value.

BT_UUID_OTS_LIST_CP
: OTS Object List Control Point Characteristic.

BT_UUID_OTS_LIST_FILTER_VAL
: OTS Object List Filter Characteristic UUID value.

BT_UUID_OTS_LIST_FILTER
: OTS Object List Filter Characteristic.

BT_UUID_OTS_CHANGED_VAL
: OTS Object Changed Characteristic UUID value.

BT_UUID_OTS_CHANGED
: OTS Object Changed Characteristic.

BT_UUID_GATT_RPAO_VAL
: GATT Characteristic Resolvable Private Address Only UUID Value.

BT_UUID_GATT_RPAO
: GATT Characteristic Resolvable Private Address Only.

BT_UUID_OTS_TYPE_UNSPECIFIED_VAL
: OTS Unspecified Object Type UUID value.

`BT_UUID_OTS_TYPE_UNSPECIFIED`
: OTS Unspecified Object Type.

`BT_UUID_OTS_DIRECTORY_LISTING_VAL`
: OTS Directory Listing UUID value.

`BT_UUID_OTS_DIRECTORY_LISTING`
: OTS Directory Listing.

`BT_UUID_GATT_FMF_VAL`
: GATT Characteristic Fitness Machine Feature UUID Value.

`BT_UUID_GATT_FMF`
: GATT Characteristic Fitness Machine Feature.

`BT_UUID_GATT_TD_VAL`
: GATT Characteristic Treadmill Data UUID Value.

`BT_UUID_GATT_TD`
: GATT Characteristic Treadmill Data.

`BT_UUID_GATT_CTD_VAL`
: GATT Characteristic Cross Trainer Data UUID Value.

`BT_UUID_GATT_CTD`
: GATT Characteristic Cross Trainer Data.

`BT_UUID_GATT_STPCD_VAL`
: GATT Characteristic Step Climber Data UUID Value.

`BT_UUID_GATT_STPCD`
: GATT Characteristic Step Climber Data.

`BT_UUID_GATT_STRCD_VAL`
: GATT Characteristic Stair Climber Data UUID Value.

`BT_UUID_GATT_STRCD`
: GATT Characteristic Stair Climber Data.

`BT_UUID_GATT_RD_VAL`
: GATT Characteristic Rower Data UUID Value.

`BT_UUID_GATT_RD`
: GATT Characteristic Rower Data.

`BT_UUID_GATT_IBD_VAL`
: GATT Characteristic Indoor Bike Data UUID Value.

BT_UUID_GATT_IBD
: GATT Characteristic Indoor Bike Data.

BT_UUID_GATT_TRSTAT_VAL
: GATT Characteristic Training Status UUID Value.

BT_UUID_GATT_TRSTAT
: GATT Characteristic Training Status.

BT_UUID_GATT_SSR_VAL
: GATT Characteristic Supported Speed Range UUID Value.

BT_UUID_GATT_SSR
: GATT Characteristic Supported Speed Range.

BT_UUID_GATT_SIR_VAL
: GATT Characteristic Supported Inclination Range UUID Value.

BT_UUID_GATT_SIR
: GATT Characteristic Supported Inclination Range.

BT_UUID_GATT_SRLR_VAL
: GATT Characteristic Supported Resistance Level Range UUID Value.

BT_UUID_GATT_SRLR
: GATT Characteristic Supported Resistance Level Range.

BT_UUID_GATT_SHRR_VAL
: GATT Characteristic Supported Heart Rate Range UUID Value.

BT_UUID_GATT_SHRR
: GATT Characteristic Supported Heart Rate Range.

BT_UUID_GATT_SPR_VAL
: GATT Characteristic Supported Power Range UUID Value.

BT_UUID_GATT_SPR
: GATT Characteristic Supported Power Range.

BT_UUID_GATT_FMCP_VAL
: GATT Characteristic Fitness Machine Control Point UUID Value.

BT_UUID_GATT_FMCP
: GATT Characteristic Fitness Machine Control Point.

BT_UUID_GATT_FMS_VAL
: GATT Characteristic Fitness Machine Status UUID Value.

BT_UUID_GATT_FMS
: GATT Characteristic Fitness Machine Status.

BT_UUID_MESH_PROV_DATA_IN_VAL
: Mesh Provisioning Data In UUID value.

BT_UUID_MESH_PROV_DATA_IN
: Mesh Provisioning Data In.

BT_UUID_MESH_PROV_DATA_OUT_VAL
: Mesh Provisioning Data Out UUID value.

BT_UUID_MESH_PROV_DATA_OUT
: Mesh Provisioning Data Out.

BT_UUID_MESH_PROXY_DATA_IN_VAL
: Mesh Proxy Data In UUID value.

BT_UUID_MESH_PROXY_DATA_IN
: Mesh Proxy Data In.

BT_UUID_MESH_PROXY_DATA_OUT_VAL
: Mesh Proxy Data Out UUID value.

BT_UUID_MESH_PROXY_DATA_OUT
: Mesh Proxy Data Out.

BT_UUID_GATT_NNN_VAL
: GATT Characteristic New Number Needed UUID Value.

BT_UUID_GATT_NNN
: GATT Characteristic New Number Needed.

BT_UUID_GATT_AC_VAL
: GATT Characteristic Average Current UUID Value.

BT_UUID_GATT_AC
: GATT Characteristic Average Current.

BT_UUID_GATT_AV_VAL
: GATT Characteristic Average Voltage UUID Value.

BT_UUID_GATT_AV
: GATT Characteristic Average Voltage.

BT_UUID_GATT_BOOLEAN_VAL
: GATT Characteristic Boolean UUID Value.

BT_UUID_GATT_BOOLEAN
: GATT Characteristic Boolean.

BT_UUID_GATT_CRDFP_VAL
: GATT Characteristic Chromatic Distance From Planckian UUID Value.

BT_UUID_GATT_CRDFP
: GATT Characteristic Chromatic Distance From Planckian.

BT_UUID_GATT_CRCOORDS_VAL
: GATT Characteristic Chromaticity Coordinates UUID Value.

BT_UUID_GATT_CRCOORDS
: GATT Characteristic Chromaticity Coordinates.

BT_UUID_GATT_CRCCT_VAL
: GATT Characteristic Chromaticity In CCT And Duv Values UUID Value.

BT_UUID_GATT_CRCCT
: GATT Characteristic Chromaticity In CCT And Duv Values.

BT_UUID_GATT_CRT_VAL
: GATT Characteristic Chromaticity Tolerance UUID Value.

BT_UUID_GATT_CRT
: GATT Characteristic Chromaticity Tolerance.

BT_UUID_GATT_CIEIDX_VAL
: GATT Characteristic CIE 13.3-1995 Color Rendering Index UUID Value.

BT_UUID_GATT_CIEIDX
: GATT Characteristic CIE 13.3-1995 Color Rendering Index.

BT_UUID_GATT_COEFFICIENT_VAL
: GATT Characteristic Coefficient UUID Value.

BT_UUID_GATT_COEFFICIENT
: GATT Characteristic Coefficient.

BT_UUID_GATT_CCTEMP_VAL
: GATT Characteristic Correlated Color Temperature UUID Value.

BT_UUID_GATT_CCTEMP
: GATT Characteristic Correlated Color Temperature.

BT_UUID_GATT_COUNT16_VAL
: GATT Characteristic Count 16 UUID Value.

BT_UUID_GATT_COUNT16
: GATT Characteristic Count 16.

BT_UUID_GATT_COUNT24_VAL
: GATT Characteristic Count 24 UUID Value.

BT_UUID_GATT_COUNT24
: GATT Characteristic Count 24.

BT_UUID_GATT_CNTRCODE_VAL
: GATT Characteristic Country Code UUID Value.

BT_UUID_GATT_CNTRCODE
: GATT Characteristic Country Code.

BT_UUID_GATT_DATEUTC_VAL
: GATT Characteristic Date UTC UUID Value.

BT_UUID_GATT_DATEUTC
: GATT Characteristic Date UTC.

BT_UUID_GATT_EC_VAL
: GATT Characteristic Electric Current UUID Value.

BT_UUID_GATT_EC
: GATT Characteristic Electric Current.

BT_UUID_GATT_ECR_VAL
: GATT Characteristic Electric Current Range UUID Value.

BT_UUID_GATT_ECR
: GATT Characteristic Electric Current Range.

BT_UUID_GATT_ECSPEC_VAL
: GATT Characteristic Electric Current Specification UUID Value.

BT_UUID_GATT_ECSPEC
: GATT Characteristic Electric Current Specification.

BT_UUID_GATT_ECSTAT_VAL
: GATT Characteristic Electric Current Statistics UUID Value.

BT_UUID_GATT_ECSTAT
: GATT Characteristic Electric Current Statistics.

BT_UUID_GATT_ENERGY_VAL
: GATT Characteristic Energy UUID Value.

BT_UUID_GATT_ENERGY
: GATT Characteristic Energy.

BT_UUID_GATT_EPOD_VAL
: GATT Characteristic Energy In A Period Of Day UUID Value.

BT_UUID_GATT_EPOD
: GATT Characteristic Energy In A Period Of Day.

BT_UUID_GATT_EVTSTAT_VAL
: GATT Characteristic Event Statistics UUID Value.

BT_UUID_GATT_EVTSTAT
: GATT Characteristic Event Statistics.

BT_UUID_GATT_FSTR16_VAL
: GATT Characteristic Fixed String 16 UUID Value.

BT_UUID_GATT_FSTR16
: GATT Characteristic Fixed String 16.

BT_UUID_GATT_FSTR24_VAL
: GATT Characteristic Fixed String 24 UUID Value.

BT_UUID_GATT_FSTR24
: GATT Characteristic Fixed String 24.

BT_UUID_GATT_FSTR36_VAL
: GATT Characteristic Fixed String 36 UUID Value.

BT_UUID_GATT_FSTR36
: GATT Characteristic Fixed String 36.

BT_UUID_GATT_FSTR8_VAL
: GATT Characteristic Fixed String 8 UUID Value.

BT_UUID_GATT_FSTR8
: GATT Characteristic Fixed String 8.

BT_UUID_GATT_GENLVL_VAL
: GATT Characteristic Generic Level UUID Value.

BT_UUID_GATT_GENLVL
: GATT Characteristic Generic Level.

BT_UUID_GATT_GTIN_VAL
: GATT Characteristic Global Trade Item Number UUID Value.

BT_UUID_GATT_GTIN
: GATT Characteristic Global Trade Item Number.

BT_UUID_GATT_ILLUM_VAL
: GATT Characteristic Illuminance UUID Value.

BT_UUID_GATT_ILLUM
: GATT Characteristic Illuminance.

BT_UUID_GATT_LUMEFF_VAL
: GATT Characteristic Luminous Efficacy UUID Value.

BT_UUID_GATT_LUMEFF
: GATT Characteristic Luminous Efficacy.

BT_UUID_GATT_LUMNRG_VAL
: GATT Characteristic Luminous Energy UUID Value.

BT_UUID_GATT_LUMNRG
: GATT Characteristic Luminous Energy.

BT_UUID_GATT_LUMEXP_VAL
: GATT Characteristic Luminous Exposure UUID Value.

BT_UUID_GATT_LUMEXP
: GATT Characteristic Luminous Exposure.

BT_UUID_GATT_LUMFLX_VAL
: GATT Characteristic Luminous Flux UUID Value.

BT_UUID_GATT_LUMFLX
: GATT Characteristic Luminous Flux.

BT_UUID_GATT_LUMFLXR_VAL
: GATT Characteristic Luminous Flux Range UUID Value.

BT_UUID_GATT_LUMFLXR
: GATT Characteristic Luminous Flux Range.

BT_UUID_GATT_LUMINT_VAL
: GATT Characteristic Luminous Intensity UUID Value.

BT_UUID_GATT_LUMINT
: GATT Characteristic Luminous Intensity.

BT_UUID_GATT_MASSFLOW_VAL
: GATT Characteristic Mass Flow UUID Value.

BT_UUID_GATT_MASSFLOW
: GATT Characteristic Mass Flow.

BT_UUID_GATT_PERLGHT_VAL
: GATT Characteristic Perceived Lightness UUID Value.

BT_UUID_GATT_PERLGHT
: GATT Characteristic Perceived Lightness.

BT_UUID_GATT_PER8_VAL
: GATT Characteristic Percentage 8 UUID Value.

BT_UUID_GATT_PER8
: GATT Characteristic Percentage 8.

BT_UUID_GATT_PWR_VAL
: GATT Characteristic Power UUID Value.

BT_UUID_GATT_PWR
: GATT Characteristic Power.

BT_UUID_GATT_PWRSPEC_VAL
: GATT Characteristic Power Specification UUID Value.

BT_UUID_GATT_PWRSPEC
: GATT Characteristic Power Specification.

BT_UUID_GATT_RRICR_VAL
: GATT Characteristic Relative Runtime In A Current Range UUID Value.

BT_UUID_GATT_RRICR
: GATT Characteristic Relative Runtime In A Current Range.

BT_UUID_GATT_RRIGLR_VAL
: GATT Characteristic Relative Runtime In A Generic Level Range UUID Value.

BT_UUID_GATT_RRIGLR
: GATT Characteristic Relative Runtime In A Generic Level Range.

BT_UUID_GATT_RVIVR_VAL
: GATT Characteristic Relative Value In A Voltage Range UUID Value.

BT_UUID_GATT_RVIVR
: GATT Characteristic Relative Value In A Voltage Range.

BT_UUID_GATT_RVIIR_VAL
: GATT Characteristic Relative Value In A Illuminance Range UUID Value.

BT_UUID_GATT_RVIIR
: GATT Characteristic Relative Value In A Illuminance Range.

BT_UUID_GATT_RVIPOD_VAL
: GATT Characteristic Relative Value In A Period Of Day UUID Value.

BT_UUID_GATT_RVIPOD
: GATT Characteristic Relative Value In A Period Of Day.

BT_UUID_GATT_RVITR_VAL
: GATT Characteristic Relative Value In A Temperature Range UUID Value.

BT_UUID_GATT_RVITR
: GATT Characteristic Relative Value In A Temperature Range.

BT_UUID_GATT_TEMP8_VAL
: GATT Characteristic Temperature 8 UUID Value.

BT_UUID_GATT_TEMP8
: GATT Characteristic Temperature 8.

BT_UUID_GATT_TEMP8_IPOD_VAL
: GATT Characteristic Temperature 8 In A Period Of Day UUID Value.

BT_UUID_GATT_TEMP8_IPOD
: GATT Characteristic Temperature 8 In A Period Of Day.

BT_UUID_GATT_TEMP8_STAT_VAL
: GATT Characteristic Temperature 8 Statistics UUID Value.

BT_UUID_GATT_TEMP8_STAT
: GATT Characteristic Temperature 8 Statistics.

BT_UUID_GATT_TEMP_RNG_VAL
: GATT Characteristic Temperature Range UUID Value.

BT_UUID_GATT_TEMP_RNG
: GATT Characteristic Temperature Range.

BT_UUID_GATT_TEMP_STAT_VAL
: GATT Characteristic Temperature Statistics UUID Value.

BT_UUID_GATT_TEMP_STAT
: GATT Characteristic Temperature Statistics.

BT_UUID_GATT_TIM_DC8_VAL
: GATT Characteristic Time Decihour 8 UUID Value.

BT_UUID_GATT_TIM_DC8
: GATT Characteristic Time Decihour 8.

BT_UUID_GATT_TIM_EXP8_VAL
: GATT Characteristic Time Exponential 8 UUID Value.

BT_UUID_GATT_TIM_EXP8
: GATT Characteristic Time Exponential 8.

BT_UUID_GATT_TIM_H24_VAL
: GATT Characteristic Time Hour 24 UUID Value.

BT_UUID_GATT_TIM_H24
: GATT Characteristic Time Hour 24.

BT_UUID_GATT_TIM_MS24_VAL
: GATT Characteristic Time Millisecond 24 UUID Value.

BT_UUID_GATT_TIM_MS24
: GATT Characteristic Time Millisecond 24.

BT_UUID_GATT_TIM_S16_VAL
: GATT Characteristic Time Second 16 UUID Value.

BT_UUID_GATT_TIM_S16
: GATT Characteristic Time Second 16.

BT_UUID_GATT_TIM_S8_VAL
: GATT Characteristic Time Second 8 UUID Value.

BT_UUID_GATT_TIM_S8
: GATT Characteristic Time Second 8.

BT_UUID_GATT_V_VAL
: GATT Characteristic Voltage UUID Value.

BT_UUID_GATT_V
: GATT Characteristic Voltage.

BT_UUID_GATT_V_SPEC_VAL
: GATT Characteristic Voltage Specification UUID Value.

BT_UUID_GATT_V_SPEC
: GATT Characteristic Voltage Specification.

BT_UUID_GATT_V_STAT_VAL
: GATT Characteristic Voltage Statistics UUID Value.

BT_UUID_GATT_V_STAT
: GATT Characteristic Voltage Statistics.

BT_UUID_GATT_VOLF_VAL
: GATT Characteristic Volume Flow UUID Value.

BT_UUID_GATT_VOLF
: GATT Characteristic Volume Flow.

BT_UUID_GATT_CRCOORD_VAL
: GATT Characteristic Chromaticity Coordinate (not Coordinates) UUID Value.

BT_UUID_GATT_CRCOORD
: GATT Characteristic Chromaticity Coordinate (not Coordinates)

BT_UUID_GATT_RCF_VAL
: GATT Characteristic RC Feature UUID Value.

BT_UUID_GATT_RCF
: GATT Characteristic RC Feature.

BT_UUID_GATT_RCSET_VAL
: GATT Characteristic RC Settings UUID Value.

BT_UUID_GATT_RCSET
: GATT Characteristic RC Settings.

BT_UUID_GATT_RCCP_VAL
: GATT Characteristic Reconnection Configuration Control Point UUID Value.

BT_UUID_GATT_RCCP
: GATT Characteristic Reconnection Configuration Control Point.

BT_UUID_GATT_IDD_SC_VAL
: GATT Characteristic IDD Status Changed UUID Value.

BT_UUID_GATT_IDD_SC
: GATT Characteristic IDD Status Changed.

BT_UUID_GATT_IDD_S_VAL
: GATT Characteristic IDD Status UUID Value.

BT_UUID_GATT_IDD_S
: GATT Characteristic IDD Status.

BT_UUID_GATT_IDD_AS_VAL
: GATT Characteristic IDD Annunciation Status UUID Value.

BT_UUID_GATT_IDD_AS
: GATT Characteristic IDD Annunciation Status.

BT_UUID_GATT_IDD_F_VAL
: GATT Characteristic IDD Features UUID Value.

BT_UUID_GATT_IDD_F
: GATT Characteristic IDD Features.

BT_UUID_GATT_IDD_SRCP_VAL
: GATT Characteristic IDD Status Reader Control Point UUID Value.

BT_UUID_GATT_IDD_SRCP
: GATT Characteristic IDD Status Reader Control Point.

BT_UUID_GATT_IDD_CCP_VAL
: GATT Characteristic IDD Command Control Point UUID Value.

BT_UUID_GATT_IDD_CCP
: GATT Characteristic IDD Command Control Point.

BT_UUID_GATT_IDD_CD_VAL
: GATT Characteristic IDD Command Data UUID Value.

BT_UUID_GATT_IDD_CD
: GATT Characteristic IDD Command Data.

BT_UUID_GATT_IDD_RACP_VAL
: GATT Characteristic IDD Record Access Control Point UUID Value.

BT_UUID_GATT_IDD_RACP
: GATT Characteristic IDD Record Access Control Point.

BT_UUID_GATT_IDD_HD_VAL
: GATT Characteristic IDD History Data UUID Value.

BT_UUID_GATT_IDD_HD
: GATT Characteristic IDD History Data.

BT_UUID_GATT_CLIENT_FEATURES_VAL
: GATT Characteristic Client Supported Features UUID value.

BT_UUID_GATT_CLIENT_FEATURES
: GATT Characteristic Client Supported Features.

BT_UUID_GATT_DB_HASH_VAL
: GATT Characteristic Database Hash UUID value.

BT_UUID_GATT_DB_HASH
: GATT Characteristic Database Hash.

BT_UUID_GATT_BSS_CP_VAL
: GATT Characteristic BSS Control Point UUID Value.

BT_UUID_GATT_BSS_CP
: GATT Characteristic BSS Control Point.

BT_UUID_GATT_BSS_R_VAL
: GATT Characteristic BSS Response UUID Value.

BT_UUID_GATT_BSS_R
: GATT Characteristic BSS Response.

BT_UUID_GATT_EMG_ID_VAL
: GATT Characteristic Emergency ID UUID Value.

BT_UUID_GATT_EMG_ID
: GATT Characteristic Emergency ID.

BT_UUID_GATT_EMG_TXT_VAL
: GATT Characteristic Emergency Text UUID Value.

BT_UUID_GATT_EMG_TXT
: GATT Characteristic Emergency Text.

BT_UUID_GATT_ACS_S_VAL
: GATT Characteristic ACS Status UUID Value.

BT_UUID_GATT_ACS_S
: GATT Characteristic ACS Status.

BT_UUID_GATT_ACS_DI_VAL
: GATT Characteristic ACS Data In UUID Value.

BT_UUID_GATT_ACS_DI
: GATT Characteristic ACS Data In.

BT_UUID_GATT_ACS_DON_VAL
: GATT Characteristic ACS Data Out Notify UUID Value.

BT_UUID_GATT_ACS_DON
: GATT Characteristic ACS Data Out Notify.

BT_UUID_GATT_ACS_DOI_VAL
: GATT Characteristic ACS Data Out Indicate UUID Value.

BT_UUID_GATT_ACS_DOI
: GATT Characteristic ACS Data Out Indicate.

BT_UUID_GATT_ACS_CP_VAL
: GATT Characteristic ACS Control Point UUID Value.

BT_UUID_GATT_ACS_CP
: GATT Characteristic ACS Control Point.

BT_UUID_GATT_EBPM_VAL
: GATT Characteristic Enhanced Blood Pressure Measurement UUID Value.

BT_UUID_GATT_EBPM
: GATT Characteristic Enhanced Blood Pressure Measurement.

BT_UUID_GATT_EICP_VAL
: GATT Characteristic Enhanced Intermediate Cuff Pressure UUID Value.

BT_UUID_GATT_EICP
: GATT Characteristic Enhanced Intermediate Cuff Pressure.

BT_UUID_GATT_BPR_VAL
: GATT Characteristic Blood Pressure Record UUID Value.

BT_UUID_GATT_BPR
: GATT Characteristic Blood Pressure Record.

BT_UUID_GATT_RU_VAL
: GATT Characteristic Registered User UUID Value.

BT_UUID_GATT_RU
: GATT Characteristic Registered User.

BT_UUID_GATT_BR_EDR_HD_VAL
: GATT Characteristic BR-EDR Handover Data UUID Value.

BT_UUID_GATT_BR_EDR_HD
: GATT Characteristic BR-EDR Handover Data.

BT_UUID_GATT_BT_SIG_D_VAL
: GATT Characteristic Bluetooth SIG Data UUID Value.

BT_UUID_GATT_BT_SIG_D
: GATT Characteristic Bluetooth SIG Data.

BT_UUID_GATT_SERVER_FEATURES_VAL
: GATT Characteristic Server Supported Features UUID value.

BT_UUID_GATT_SERVER_FEATURES
: GATT Characteristic Server Supported Features.

BT_UUID_GATT_PHY_AMF_VAL
: GATT Characteristic Physical Activity Monitor Features UUID Value.

BT_UUID_GATT_PHY_AMF
: GATT Characteristic Physical Activity Monitor Features.

BT_UUID_GATT_GEN_AID_VAL
: GATT Characteristic General Activity Instantaneous Data UUID Value.

BT_UUID_GATT_GEN_AID
: GATT Characteristic General Activity Instantaneous Data.

BT_UUID_GATT_GEN_ASD_VAL
: GATT Characteristic General Activity Summary Data UUID Value.

BT_UUID_GATT_GEN_ASD
: GATT Characteristic General Activity Summary Data.

BT_UUID_GATT_CR_AID_VAL
: GATT Characteristic CardioRespiratory Activity Instantaneous Data UUID Value.

BT_UUID_GATT_CR_AID
: GATT Characteristic CardioRespiratory Activity Instantaneous Data.

BT_UUID_GATT_CR_ASD_VAL
: GATT Characteristic CardioRespiratory Activity Summary Data UUID Value.

BT_UUID_GATT_CR_ASD
: GATT Characteristic CardioRespiratory Activity Summary Data.

BT_UUID_GATT_SC_ASD_VAL
: GATT Characteristic Step Counter Activity Summary Data UUID Value.

BT_UUID_GATT_SC_ASD
: GATT Characteristic Step Counter Activity Summary Data.

BT_UUID_GATT_SLP_AID_VAL
: GATT Characteristic Sleep Activity Instantaneous Data UUID Value.

BT_UUID_GATT_SLP_AID
: GATT Characteristic Sleep Activity Instantaneous Data.

BT_UUID_GATT_SLP_ASD_VAL
: GATT Characteristic Sleep Activity Summary Data UUID Value.

BT_UUID_GATT_SLP_ASD
: GATT Characteristic Sleep Activity Summary Data.

BT_UUID_GATT_PHY_AMCP_VAL
: GATT Characteristic Physical Activity Monitor Control Point UUID Value.

BT_UUID_GATT_PHY_AMCP
: GATT Characteristic Physical Activity Monitor Control Point.

BT_UUID_GATT_ACS_VAL
: GATT Characteristic Activity Current Session UUID Value.

BT_UUID_GATT_ACS
: GATT Characteristic Activity Current Session.

BT_UUID_GATT_PHY_ASDESC_VAL
: GATT Characteristic Physical Activity Session Descriptor UUID Value.

BT_UUID_GATT_PHY_ASDESC
: GATT Characteristic Physical Activity Session Descriptor.

BT_UUID_GATT_PREF_U_VAL
: GATT Characteristic Preferred Units UUID Value.

BT_UUID_GATT_PREF_U
: GATT Characteristic Preferred Units.

BT_UUID_GATT_HRES_H_VAL
: GATT Characteristic High Resolution Height UUID Value.

BT_UUID_GATT_HRES_H
: GATT Characteristic High Resolution Height.

BT_UUID_GATT_MID_NAME_VAL
: GATT Characteristic Middle Name UUID Value.

BT_UUID_GATT_MID_NAME
: GATT Characteristic Middle Name.

BT_UUID_GATT_STRDLEN_VAL
: GATT Characteristic Stride Length UUID Value.

BT_UUID_GATT_STRDLEN
: GATT Characteristic Stride Length.

BT_UUID_GATT_HANDEDNESS_VAL
: GATT Characteristic Handedness UUID Value.

BT_UUID_GATT_HANDEDNESS
: GATT Characteristic Handedness.

BT_UUID_GATT_DEVICE_WP_VAL
: GATT Characteristic Device Wearing Position UUID Value.

BT_UUID_GATT_DEVICE_WP
: GATT Characteristic Device Wearing Position.

BT_UUID_GATT_4ZHRL_VAL
: GATT Characteristic Four Zone Heart Rate Limit UUID Value.

BT_UUID_GATT_4ZHRL
: GATT Characteristic Four Zone Heart Rate Limit.

BT_UUID_GATT_HIET_VAL
: GATT Characteristic High Intensity Exercise Threshold UUID Value.

BT_UUID_GATT_HIET
: GATT Characteristic High Intensity Exercise Threshold.

BT_UUID_GATT_AG_VAL
: GATT Characteristic Activity Goal UUID Value.

BT_UUID_GATT_AG
: GATT Characteristic Activity Goal.

BT_UUID_GATT_SIN_VAL
: GATT Characteristic Sedentary Interval Notification UUID Value.

BT_UUID_GATT_SIN
: GATT Characteristic Sedentary Interval Notification.

BT_UUID_GATT_CI_VAL
: GATT Characteristic Caloric Intake UUID Value.

BT_UUID_GATT_CI
: GATT Characteristic Caloric Intake.

BT_UUID_GATT_TMAPR_VAL
: GATT Characteristic TMAP Role UUID Value.

BT_UUID_GATT_TMAPR
: GATT Characteristic TMAP Role.

BT_UUID_AICS_STATE_VAL
: Audio Input Control Service State value.

BT_UUID_AICS_STATE
: Audio Input Control Service State.

BT_UUID_AICS_GAIN_SETTINGS_VAL
: Audio Input Control Service Gain Settings Properties value.

BT_UUID_AICS_GAIN_SETTINGS
: Audio Input Control Service Gain Settings Properties.

BT_UUID_AICS_INPUT_TYPE_VAL
: Audio Input Control Service Input Type value.

BT_UUID_AICS_INPUT_TYPE
: Audio Input Control Service Input Type.

BT_UUID_AICS_INPUT_STATUS_VAL
: Audio Input Control Service Input Status value.

BT_UUID_AICS_INPUT_STATUS
: Audio Input Control Service Input Status.

BT_UUID_AICS_CONTROL_VAL
: Audio Input Control Service Control Point value.

BT_UUID_AICS_CONTROL
: Audio Input Control Service Control Point.

BT_UUID_AICS_DESCRIPTION_VAL
: Audio Input Control Service Input Description value.

BT_UUID_AICS_DESCRIPTION
: Audio Input Control Service Input Description.

BT_UUID_VCS_STATE_VAL
: Volume Control Setting value.

BT_UUID_VCS_STATE
: Volume Control Setting.

BT_UUID_VCS_CONTROL_VAL
: Volume Control Control point value.

BT_UUID_VCS_CONTROL
: Volume Control Control point.

BT_UUID_VCS_FLAGS_VAL
: Volume Control Flags value.

`BT_UUID_VCS_FLAGS`
: Volume Control Flags.

`BT_UUID_VOCS_STATE_VAL`
: Volume Offset State value.

`BT_UUID_VOCS_STATE`
: Volume Offset State.

`BT_UUID_VOCS_LOCATION_VAL`
: Audio Location value.

`BT_UUID_VOCS_LOCATION`
: Audio Location.

`BT_UUID_VOCS_CONTROL_VAL`
: Volume Offset Control Point value.

`BT_UUID_VOCS_CONTROL`
: Volume Offset Control Point.

`BT_UUID_VOCS_DESCRIPTION_VAL`
: Volume Offset Audio Output Description value.

`BT_UUID_VOCS_DESCRIPTION`
: Volume Offset Audio Output Description.

`BT_UUID_CSIS_SIRK_VAL`
: Set Identity Resolving Key value.

`BT_UUID_CSIS_SIRK`
: Set Identity Resolving Key.

`BT_UUID_CSIS_SET_SIZE_VAL`
: Set size value.

`BT_UUID_CSIS_SET_SIZE`
: Set size.

`BT_UUID_CSIS_SET_LOCK_VAL`
: Set lock value.

`BT_UUID_CSIS_SET_LOCK`
: Set lock.

`BT_UUID_CSIS_RANK_VAL`
: Rank value.

BT_UUID_CSIS_RANK
: Rank.

BT_UUID_GATT_EDKM_VAL
: GATT Characteristic Encrypted Data Key Material UUID Value.

BT_UUID_GATT_EDKM
: GATT Characteristic Encrypted Data Key Material.

BT_UUID_GATT_AE32_VAL
: GATT Characteristic Apparent Energy 32 UUID Value.

BT_UUID_GATT_AE32
: GATT Characteristic Apparent Energy 32.

BT_UUID_GATT_AP_VAL
: GATT Characteristic Apparent Power UUID Value.

BT_UUID_GATT_AP
: GATT Characteristic Apparent Power.

BT_UUID_GATT_CO2CONC_VAL
: GATT Characteristic CO2 Concentration UUID Value.

BT_UUID_GATT_CO2CONC
: GATT Characteristic CO2 Concentration.

BT_UUID_GATT_COS_VAL
: GATT Characteristic Cosine of the Angle UUID Value.

BT_UUID_GATT_COS
: GATT Characteristic Cosine of the Angle.

BT_UUID_GATT_DEVTF_VAL
: GATT Characteristic Device Time Feature UUID Value.

BT_UUID_GATT_DEVTF
: GATT Characteristic Device Time Feature.

BT_UUID_GATT_DEVTP_VAL
: GATT Characteristic Device Time Parameters UUID Value.

BT_UUID_GATT_DEVTP
: GATT Characteristic Device Time Parameters.

BT_UUID_GATT_DEVT_VAL
: GATT Characteristic Device Time UUID Value.

BT_UUID_GATT_DEVT
: GATT Characteristic String.

BT_UUID_GATT_DEVTCP_VAL
: GATT Characteristic Device Time Control Point UUID Value.

BT_UUID_GATT_DEVTCP
: GATT Characteristic Device Time Control Point.

BT_UUID_GATT_TCLD_VAL
: GATT Characteristic Time Change Log Data UUID Value.

BT_UUID_GATT_TCLD
: GATT Characteristic Time Change Log Data.

BT_UUID_MCS_PLAYER_NAME_VAL
: Media player name value.

BT_UUID_MCS_PLAYER_NAME
: Media player name.

BT_UUID_MCS_ICON_OBJ_ID_VAL
: Media Icon Object ID value.

BT_UUID_MCS_ICON_OBJ_ID
: Media Icon Object ID.

BT_UUID_MCS_ICON_URL_VAL
: Media Icon URL value.

BT_UUID_MCS_ICON_URL
: Media Icon URL.

BT_UUID_MCS_TRACK_CHANGED_VAL
: Track Changed value.

BT_UUID_MCS_TRACK_CHANGED
: Track Changed.

BT_UUID_MCS_TRACK_TITLE_VAL
: Track Title value.

BT_UUID_MCS_TRACK_TITLE
: Track Title.

BT_UUID_MCS_TRACK_DURATION_VAL
: Track Duration value.

BT_UUID_MCS_TRACK_DURATION
: Track Duration.

BT_UUID_MCS_TRACK_POSITION_VAL
: Track Position value.

BT_UUID_MCS_TRACK_POSITION
: Track Position.

BT_UUID_MCS_PLAYBACK_SPEED_VAL
: Playback Speed value.

BT_UUID_MCS_PLAYBACK_SPEED
: Playback Speed.

BT_UUID_MCS_SEEKING_SPEED_VAL
: Seeking Speed value.

BT_UUID_MCS_SEEKING_SPEED
: Seeking Speed.

BT_UUID_MCS_TRACK_SEGMENTS_OBJ_ID_VAL
: Track Segments Object ID value.

BT_UUID_MCS_TRACK_SEGMENTS_OBJ_ID
: Track Segments Object ID.

BT_UUID_MCS_CURRENT_TRACK_OBJ_ID_VAL
: Current Track Object ID value.

BT_UUID_MCS_CURRENT_TRACK_OBJ_ID
: Current Track Object ID.

BT_UUID_MCS_NEXT_TRACK_OBJ_ID_VAL
: Next Track Object ID value.

BT_UUID_MCS_NEXT_TRACK_OBJ_ID
: Next Track Object ID.

BT_UUID_MCS_PARENT_GROUP_OBJ_ID_VAL
: Parent Group Object ID value.

BT_UUID_MCS_PARENT_GROUP_OBJ_ID
: Parent Group Object ID.

BT_UUID_MCS_CURRENT_GROUP_OBJ_ID_VAL
: Group Object ID value.

BT_UUID_MCS_CURRENT_GROUP_OBJ_ID
: Group Object ID.

BT_UUID_MCS_PLAYING_ORDER_VAL
: Playing Order value.

BT_UUID_MCS_PLAYING_ORDER
: Playing Order.

BT_UUID_MCS_PLAYING_ORDERS_VAL
: Playing Orders supported value.

BT_UUID_MCS_PLAYING_ORDERS
: Playing Orders supported.

BT_UUID_MCS_MEDIA_STATE_VAL
: Media State value.

BT_UUID_MCS_MEDIA_STATE
: Media State.

BT_UUID_MCS_MEDIA_CONTROL_POINT_VAL
: Media Control Point value.

BT_UUID_MCS_MEDIA_CONTROL_POINT
: Media Control Point.

BT_UUID_MCS_MEDIA_CONTROL_OPCODES_VAL
: Media control opcodes supported value.

BT_UUID_MCS_MEDIA_CONTROL_OPCODES
: Media control opcodes supported.

BT_UUID_MCS_SEARCH_RESULTS_OBJ_ID_VAL
: Search result object ID value.

BT_UUID_MCS_SEARCH_RESULTS_OBJ_ID
: Search result object ID.

BT_UUID_MCS_SEARCH_CONTROL_POINT_VAL
: Search control point value.

BT_UUID_MCS_SEARCH_CONTROL_POINT
: Search control point.

BT_UUID_GATT_E32_VAL
: GATT Characteristic Energy 32 UUID Value.

BT_UUID_GATT_E32
: GATT Characteristic Energy 32.

BT_UUID_OTS_TYPE_MPL_ICON_VAL
: Media Player Icon Object Type value.

BT_UUID_OTS_TYPE_MPL_ICON
: Media Player Icon Object Type.

BT_UUID_OTS_TYPE_TRACK_SEGMENT_VAL
: Track Segments Object Type value.

BT_UUID_OTS_TYPE_TRACK_SEGMENT
: Track Segments Object Type.

BT_UUID_OTS_TYPE_TRACK_VAL
: Track Object Type value.

BT_UUID_OTS_TYPE_TRACK
: Track Object Type.

BT_UUID_OTS_TYPE_GROUP_VAL
: Group Object Type value.

BT_UUID_OTS_TYPE_GROUP
: Group Object Type.

BT_UUID_GATT_CTEE_VAL
: GATT Characteristic Constant Tone Extension Enable UUID Value.

BT_UUID_GATT_CTEE
: GATT Characteristic Constant Tone Extension Enable.

BT_UUID_GATT_ACTEML_VAL
: GATT Characteristic Advertising Constant Tone Extension Minimum Length UUID Value.

BT_UUID_GATT_ACTEML
: GATT Characteristic Advertising Constant Tone Extension Minimum Length.

BT_UUID_GATT_ACTEMTC_VAL
: GATT Characteristic Advertising Constant Tone Extension Minimum Transmit Count UUID Value.

BT_UUID_GATT_ACTEMTC
: GATT Characteristic Advertising Constant Tone Extension Minimum Transmit Count.

BT_UUID_GATT_ACTETD_VAL
: GATT Characteristic Advertising Constant Tone Extension Transmit Duration UUID Value.

BT_UUID_GATT_ACTETD
: GATT Characteristic Advertising Constant Tone Extension Transmit Duration.

BT_UUID_GATT_ACTEI_VAL
: GATT Characteristic Advertising Constant Tone Extension Interval UUID Value.

BT_UUID_GATT_ACTEI
: GATT Characteristic Advertising Constant Tone Extension Interval.

BT_UUID_GATT_ACTEP_VAL
: GATT Characteristic Advertising Constant Tone Extension PHY UUID Value.

BT_UUID_GATT_ACTEP
: GATT Characteristic Advertising Constant Tone Extension PHY.

BT_UUID_TBS_PROVIDER_NAME_VAL
: Bearer Provider Name value.

BT_UUID_TBS_PROVIDER_NAME
: Bearer Provider Name.

BT_UUID_TBS_UCI_VAL
: Bearer UCI value.

BT_UUID_TBS_UCI
: Bearer UCI.

BT_UUID_TBS_TECHNOLOGY_VAL
: Bearer Technology value.

BT_UUID_TBS_TECHNOLOGY
: Bearer Technology.

BT_UUID_TBS_URI_LIST_VAL
: Bearer URI Prefixes Supported List value.

BT_UUID_TBS_URI_LIST
: Bearer URI Prefixes Supported List.

BT_UUID_TBS_SIGNAL_STRENGTH_VAL
: Bearer Signal Strength value.

BT_UUID_TBS_SIGNAL_STRENGTH
: Bearer Signal Strength.

BT_UUID_TBS_SIGNAL_INTERVAL_VAL
: Bearer Signal Strength Reporting Interval value.

BT_UUID_TBS_SIGNAL_INTERVAL
: Bearer Signal Strength Reporting Interval.

BT_UUID_TBS_LIST_CURRENT_CALLS_VAL
: Bearer List Current Calls value.

BT_UUID_TBS_LIST_CURRENT_CALLS
: Bearer List Current Calls.

BT_UUID_CCID_VAL
: Content Control ID value.

BT_UUID_CCID
: Content Control ID.

BT_UUID_TBS_STATUS_FLAGS_VAL
: Status flags value.

BT_UUID_TBS_STATUS_FLAGS
: Status flags.

BT_UUID_TBS_INCOMING_URI_VAL
: Incoming Call Target Caller ID value.

BT_UUID_TBS_INCOMING_URI
: Incoming Call Target Caller ID.

BT_UUID_TBS_CALL_STATE_VAL
: Call State value.

BT_UUID_TBS_CALL_STATE
: Call State.

BT_UUID_TBS_CALL_CONTROL_POINT_VAL
: Call Control Point value.

BT_UUID_TBS_CALL_CONTROL_POINT
: Call Control Point.

BT_UUID_TBS_OPTIONAL_OPCODES_VAL
: Optional Opcodes value.

BT_UUID_TBS_OPTIONAL_OPCODES
: Optional Opcodes.

BT_UUID_TBS_TERMINATE_REASON_VAL
: BT_UUID_TBS_TERMINATE_REASON_VAL.

  Terminate reason value

BT_UUID_TBS_TERMINATE_REASON
: BT_UUID_TBS_TERMINATE_REASON.

  Terminate reason

BT_UUID_TBS_INCOMING_CALL_VAL
: Incoming Call value.

BT_UUID_TBS_INCOMING_CALL
: Incoming Call.

BT_UUID_TBS_FRIENDLY_NAME_VAL
: Incoming Call Friendly name value.

BT_UUID_TBS_FRIENDLY_NAME
: Incoming Call Friendly name.

BT_UUID_MICS_MUTE_VAL
: Microphone Control Service Mute value.

BT_UUID_MICS_MUTE
: Microphone Control Service Mute.

BT_UUID_ASCS_ASE_SNK_VAL
: Audio Stream Endpoint Sink Characteristic value.

BT_UUID_ASCS_ASE_SNK
: Audio Stream Endpoint Sink Characteristic.

BT_UUID_ASCS_ASE_SRC_VAL
: Audio Stream Endpoint Source Characteristic value.

BT_UUID_ASCS_ASE_SRC
: Audio Stream Endpoint Source Characteristic.

BT_UUID_ASCS_ASE_CP_VAL
: Audio Stream Endpoint Control Point Characteristic value.

BT_UUID_ASCS_ASE_CP
: Audio Stream Endpoint Control Point Characteristic.

BT_UUID_BASS_CONTROL_POINT_VAL
: Broadcast Audio Scan Service Scan State value.

BT_UUID_BASS_CONTROL_POINT
: Broadcast Audio Scan Service Scan State.

BT_UUID_BASS_RECV_STATE_VAL
: Broadcast Audio Scan Service Receive State value.

BT_UUID_BASS_RECV_STATE
: Broadcast Audio Scan Service Receive State.

BT_UUID_PACS_SNK_VAL
: Sink PAC Characteristic value.

BT_UUID_PACS_SNK
: Sink PAC Characteristic.

BT_UUID_PACS_SNK_LOC_VAL
: Sink PAC Locations Characteristic value.

BT_UUID_PACS_SNK_LOC
: Sink PAC Locations Characteristic.

BT_UUID_PACS_SRC_VAL
: Source PAC Characteristic value.

BT_UUID_PACS_SRC
: Source PAC Characteristic.

BT_UUID_PACS_SRC_LOC_VAL
: Source PAC Locations Characteristic value.

BT_UUID_PACS_SRC_LOC
: Source PAC Locations Characteristic.

BT_UUID_PACS_AVAILABLE_CONTEXT_VAL
: Available Audio Contexts Characteristic value.

BT_UUID_PACS_AVAILABLE_CONTEXT
: Available Audio Contexts Characteristic.

BT_UUID_PACS_SUPPORTED_CONTEXT_VAL
: Supported Audio Context Characteristic value.

BT_UUID_PACS_SUPPORTED_CONTEXT
: Supported Audio Context Characteristic.

BT_UUID_GATT_NH4CONC_VAL
: GATT Characteristic Ammonia Concentration UUID Value.

BT_UUID_GATT_NH4CONC
: GATT Characteristic Ammonia Concentration.

BT_UUID_GATT_COCONC_VAL
: GATT Characteristic Carbon Monoxide Concentration UUID Value.

BT_UUID_GATT_COCONC
: GATT Characteristic Carbon Monoxide Concentration.

BT_UUID_GATT_CH4CONC_VAL
: GATT Characteristic Methane Concentration UUID Value.

BT_UUID_GATT_CH4CONC
: GATT Characteristic Methane Concentration.

BT_UUID_GATT_NO2CONC_VAL
: GATT Characteristic Nitrogen Dioxide Concentration UUID Value.

BT_UUID_GATT_NO2CONC
: GATT Characteristic Nitrogen Dioxide Concentration.

BT_UUID_GATT_NONCH4CONC_VAL
: GATT Characteristic Non-Methane Volatile Organic Compounds Concentration UUID Value.

BT_UUID_GATT_NONCH4CONC
: GATT Characteristic Non-Methane Volatile Organic Compounds Concentration.

BT_UUID_GATT_O3CONC_VAL
: GATT Characteristic Ozone Concentration UUID Value.

BT_UUID_GATT_O3CONC
: GATT Characteristic Ozone Concentration.

BT_UUID_GATT_PM1CONC_VAL
: GATT Characteristic Particulate Matter - PM1 Concentration UUID Value.

BT_UUID_GATT_PM1CONC
: GATT Characteristic Particulate Matter - PM1 Concentration.

BT_UUID_GATT_PM25CONC_VAL
: GATT Characteristic Particulate Matter - PM2.5 Concentration UUID Value.

BT_UUID_GATT_PM25CONC
: GATT Characteristic Particulate Matter - PM2.5 Concentration.

BT_UUID_GATT_PM10CONC_VAL
: GATT Characteristic Particulate Matter - PM10 Concentration UUID Value.

BT_UUID_GATT_PM10CONC
: GATT Characteristic Particulate Matter - PM10 Concentration.

BT_UUID_GATT_SO2CONC_VAL
: GATT Characteristic Sulfur Dioxide Concentration UUID Value.

BT_UUID_GATT_SO2CONC
: GATT Characteristic Sulfur Dioxide Concentration.

BT_UUID_GATT_SF6CONC_VAL
: GATT Characteristic Sulfur Hexafluoride Concentration UUID Value.

BT_UUID_GATT_SF6CONC
: GATT Characteristic Sulfur Hexafluoride Concentration.

BT_UUID_HAS_HEARING_AID_FEATURES_VAL
: Hearing Aid Features Characteristic value.

BT_UUID_HAS_HEARING_AID_FEATURES
: Hearing Aid Features Characteristic.

BT_UUID_HAS_PRESET_CONTROL_POINT_VAL
: Hearing Aid Preset Control Point Characteristic value.

BT_UUID_HAS_PRESET_CONTROL_POINT
: Hearing Aid Preset Control Point Characteristic.

BT_UUID_HAS_ACTIVE_PRESET_INDEX_VAL
: Active Preset Index Characteristic value.

BT_UUID_HAS_ACTIVE_PRESET_INDEX
: Active Preset Index Characteristic.

BT_UUID_GATT_FSTR64_VAL
: GATT Characteristic Fixed String 64 UUID Value.

BT_UUID_GATT_FSTR64
: GATT Characteristic Fixed String 64.

BT_UUID_GATT_HITEMP_VAL
: GATT Characteristic High Temperature UUID Value.

BT_UUID_GATT_HITEMP
: GATT Characteristic High Temperature.

BT_UUID_GATT_HV_VAL
: GATT Characteristic High Voltage UUID Value.

BT_UUID_GATT_HV
: GATT Characteristic High Voltage.

BT_UUID_GATT_LD_VAL
: GATT Characteristic Light Distribution UUID Value.

BT_UUID_GATT_LD
: GATT Characteristic Light Distribution.

BT_UUID_GATT_LO_VAL
: GATT Characteristic Light Output UUID Value.

BT_UUID_GATT_LO
: GATT Characteristic Light Output.

BT_UUID_GATT_LST_VAL
: GATT Characteristic Light Source Type UUID Value.

BT_UUID_GATT_LST
: GATT Characteristic Light Source Type.

BT_UUID_GATT_NOISE_VAL
: GATT Characteristic Noise UUID Value.

BT_UUID_GATT_NOISE
: GATT Characteristic Noise.

BT_UUID_GATT_RRCCTP_VAL
: GATT Characteristic Relative Runtime in a Correlated Color Temperature Range UUID Value.

BT_UUID_GATT_RRCCTR
: GATT Characteristic Relative Runtime in a Correlated Color Temperature Range.

BT_UUID_GATT_TIM_S32_VAL
: GATT Characteristic Time Second 32 UUID Value.

BT_UUID_GATT_TIM_S32
: GATT Characteristic Time Second 32.

BT_UUID_GATT_VOCCONC_VAL
: GATT Characteristic VOC Concentration UUID Value.

BT_UUID_GATT_VOCCONC
: GATT Characteristic VOC Concentration.

BT_UUID_GATT_VF_VAL
: GATT Characteristic Voltage Frequency UUID Value.

BT_UUID_GATT_VF
: GATT Characteristic Voltage Frequency.

BT_UUID_BAS_BATTERY_CRIT_STATUS_VAL
: BAS Characteristic Battery Critical Status UUID Value.

BT_UUID_BAS_BATTERY_CRIT_STATUS
: BAS Characteristic Battery Critical Status.

BT_UUID_BAS_BATTERY_HEALTH_STATUS_VAL
: BAS Characteristic Battery Health Status UUID Value.

BT_UUID_BAS_BATTERY_HEALTH_STATUS
: BAS Characteristic Battery Health Status.

BT_UUID_BAS_BATTERY_HEALTH_INF_VAL
: BAS Characteristic Battery Health Information UUID Value.

BT_UUID_BAS_BATTERY_HEALTH_INF
: BAS Characteristic Battery Health Information.

BT_UUID_BAS_BATTERY_INF_VAL
: BAS Characteristic Battery Information UUID Value.

BT_UUID_BAS_BATTERY_INF
: BAS Characteristic Battery Information.

BT_UUID_BAS_BATTERY_LEVEL_STATUS_VAL
: BAS Characteristic Battery Level Status UUID Value.

BT_UUID_BAS_BATTERY_LEVEL_STATUS
: BAS Characteristic Battery Level Status.

BT_UUID_BAS_BATTERY_TIME_STATUS_VAL
: BAS Characteristic Battery Time Status UUID Value.

BT_UUID_BAS_BATTERY_TIME_STATUS
: BAS Characteristic Battery Time Status.

BT_UUID_GATT_ESD_VAL
: GATT Characteristic Estimated Service Date UUID Value.

BT_UUID_GATT_ESD
: GATT Characteristic Estimated Service Date.

BT_UUID_BAS_BATTERY_ENERGY_STATUS_VAL
: BAS Characteristic Battery Energy Status UUID Value.

BT_UUID_BAS_BATTERY_ENERGY_STATUS
: BAS Characteristic Battery Energy Status.

BT_UUID_GATT_SL_VAL
: GATT Characteristic LE GATT Security Levels UUID Value.

BT_UUID_GATT_SL
: GATT Characteristic LE GATT Security Levels.

BT_UUID_GMAS_VAL
: Gaming Service UUID value.

BT_UUID_GMAS
: Common Audio Service.

BT_UUID_GMAP_ROLE_VAL
: Gaming Audio Profile Role UUID value.

BT_UUID_GMAP_ROLE
: Gaming Audio Profile Role.

BT_UUID_GMAP_UGG_FEAT_VAL
: Gaming Audio Profile Unicast Game Gateway Features UUID value.

BT_UUID_GMAP_UGG_FEAT
: Gaming Audio Profile Unicast Game Gateway Features.

BT_UUID_GMAP_UGT_FEAT_VAL
: Gaming Audio Profile Unicast Game Terminal Features UUID value.

BT_UUID_GMAP_UGT_FEAT
: Gaming Audio Profile Unicast Game Terminal Features.

BT_UUID_GMAP_BGS_FEAT_VAL
: Gaming Audio Profile Broadcast Game Sender Features UUID value.

BT_UUID_GMAP_BGS_FEAT
: Gaming Audio Profile Broadcast Game Sender Features.

BT_UUID_GMAP_BGR_FEAT_VAL
: Gaming Audio Profile Broadcast Game Receiver Features UUID value.

BT_UUID_GMAP_BGR_FEAT
: Gaming Audio Profile Broadcast Game Receiver Features.

BT_UUID_SDP_VAL

BT_UUID_SDP

BT_UUID_UDP_VAL

BT_UUID_UDP

BT_UUID_RFCOMM_VAL

BT_UUID_RFCOMM

BT_UUID_TCP_VAL

BT_UUID_TCP

BT_UUID_TCS_BIN_VAL

BT_UUID_TCS_BIN

BT_UUID_TCS_AT_VAL

BT_UUID_TCS_AT

BT_UUID_ATT_VAL

BT_UUID_ATT

BT_UUID_OBEX_VAL

BT_UUID_OBEX

BT_UUID_IP_VAL

BT_UUID_IP

BT_UUID_FTP_VAL

BT_UUID_FTP

BT_UUID_HTTP_VAL

BT_UUID_HTTP

BT_UUID_WSP_VAL

BT_UUID_WSP

BT_UUID_BNEP_VAL

BT_UUID_BNEP

BT_UUID_UPNP_VAL

BT_UUID_UPNP

BT_UUID_HIDP_VAL

BT_UUID_HIDP

BT_UUID_HCRP_CTRL_VAL

BT_UUID_HCRP_CTRL

BT_UUID_HCRP_DATA_VAL

BT_UUID_HCRP_DATA

BT_UUID_HCRP_NOTE_VAL

BT_UUID_HCRP_NOTE

BT_UUID_AVCTP_VAL

BT_UUID_AVCTP

BT_UUID_AVDTP_VAL

BT_UUID_AVDTP

BT_UUID_CMTP_VAL

BT_UUID_CMTP

BT_UUID_UDI_VAL

BT_UUID_UDI

BT_UUID_MCAP_CTRL_VAL

BT_UUID_MCAP_CTRL

BT_UUID_MCAP_DATA_VAL

BT_UUID_MCAP_DATA

BT_UUID_L2CAP_VAL

BT_UUID_L2CAP

Enums

Bluetooth UUID types.

Values:

enumerator **BT_UUID_TYPE_16**
UUID type 16-bit.

enumerator **BT_UUID_TYPE_32**
UUID type 32-bit.

enumerator **BT_UUID_TYPE_128**
UUID type 128-bit.

Functions

int **bt_uuid_cmp**(const struct *bt_uuid* *u1, const struct *bt_uuid* *u2)
Compare Bluetooth UUIDs.

Compares 2 Bluetooth UUIDs, if the types are different both UUIDs are first converted to 128 bits format before comparing.

Parameters

- **u1** – First Bluetooth UUID to compare
- **u2** – Second Bluetooth UUID to compare

Returns
negative value if *u1* < *u2*, 0 if *u1* == *u2*, else positive

bool **bt_uuid_create**(struct *bt_uuid* *uuid, const uint8_t *data, uint8_t data_len)
Create a *bt_uuid* from a little-endian data buffer.

Create a *bt_uuid* from a little-endian data buffer. The data_len parameter is used to determine whether the UUID is in 16, 32 or 128 bit format (length 2, 4 or 16). Note: 32 bit format is not allowed over the air.

Parameters

- **uuid** – Pointer to the *bt_uuid* variable
- **data** – pointer to UUID stored in little-endian data buffer
- **data_len** – length of the UUID in the data buffer

Returns
true if the data was valid and the UUID was successfully created.

void **bt_uuid_to_str**(const struct *bt_uuid* *uuid, char *str, size_t len)
Convert Bluetooth UUID to string.

Converts Bluetooth UUID to string. UUID can be in any format, 16-bit, 32-bit or 128-bit.

Parameters

- **uuid** – Bluetooth UUID
- **str** – pointer where to put converted string
- **len** – length of str

struct **bt_uuid**

#include <uuid.h> This is a 'tentative' type and should be used as a pointer only.

struct **bt_uuid_16**

#include <uuid.h>

Public Members

struct *bt_uuid* **uuid**

UUID generic type.

uint16_t **val**

UUID value, 16-bit in host endianness.

struct **bt_uuid_32**

#include <uuid.h>

Public Members

struct *bt_uuid* **uuid**

UUID generic type.

uint32_t **val**

UUID value, 32-bit in host endianness.

struct **bt_uuid_128**

#include <uuid.h>

Public Members

struct *bt_uuid* **uuid**

UUID generic type.

uint8_t **val**[16]

UUID value, 128-bit in little-endian format.

6.1.10 Tools

This page lists and describes tools that can be used to assist during Bluetooth stack or application development in order to help, simplify and speed up the development process.

- *Mobile applications*
- *Using BlueZ with Zephyr*

- *Running on QEMU or native_sim*
  - *Using the Host System Bluetooth Controller*
  - *Using a Zephyr-based BLE Controller*
  - *HCI Tracing*
- *Running on a Virtual Controller and native_sim*
  - *Android Emulator*
- *Using Zephyr-based Controllers with BlueZ*

Mobile applications

It is often useful to make use of existing mobile applications to interact with hardware running Zephyr, to test functionality without having to write any additional code or requiring extra hardware.

The recommended mobile applications for interacting with Zephyr are:

- Android:
  - nRF Connect for Android
  - nRF Mesh for Android
  - LightBlue for Android
- iOS:
  - nRF Connect for iOS
  - nRF Mesh for iOS
  - LightBlue for iOS

Using BlueZ with Zephyr

The Linux Bluetooth Protocol Stack, BlueZ, comes with a very useful set of tools that can be used to debug and interact with Zephyr's BLE Host and Controller. In order to benefit from these tools you will need to make sure that you are running a recent version of the Linux Kernel and BlueZ:

- Linux Kernel 4.10+
- BlueZ 4.45+

Additionally, some of the BlueZ tools might not be bundled by default by your Linux distribution. If you need to build BlueZ from scratch to update to a recent version or to obtain all of its tools you can follow the steps below:

```
git clone git://git.kernel.org/pub/scm/bluetooth/bluez.git
cd bluez
./bootstrap-configure --disable-android --disable-midi
make
```

You can then find `btattach`, `btmgt` and `btproxy` in the `tools/` folder and `btmon` in the `monitor/` folder.

You'll need to enable BlueZ's experimental features so you can access its most recent BLE functionality. Do this by editing the file `/lib/systemd/system/bluetooth.service` and making sure to include the `-E` option in the daemon's execution start line:

```
ExecStart=/usr/libexec/bluetooth/bluetoothd -E
```

Finally, reload and restart the daemon:

```
sudo systemctl daemon-reload
sudo systemctl restart bluetooth
```

Running on QEMU or native_sim

It's possible to run Bluetooth applications using either the *QEMU emulator* or native_sim.

In either case, a Bluetooth controller needs to be exported from the host OS (Linux) to the emulator. For this purpose you will need some tools described in the *Using BlueZ with Zephyr* section.

Using the Host System Bluetooth Controller The host OS's Bluetooth controller is connected in the following manner:

- To the second QEMU serial line using a UNIX socket. This socket gets used with the help of the QEMU option `-serial unix:/tmp/bt-server-bredr`. This option gets passed to QEMU through `QEMU_EXTRA_FLAGS` automatically whenever an application has enabled Bluetooth support.
- To native_sim's BT User Channel driver through the use of a command-line option passed to the native_sim executable: `--bt-dev=hci0`

On the host side, BlueZ allows you to export its Bluetooth controller through a so-called user channel for QEMU and native_sim to use.

Note

You only need to run `btproxy` when using QEMU. native_sim handles the UNIX socket proxying automatically

If you are using QEMU, in order to make the Controller available you will need one additional step using `btproxy`:

1. Make sure that the Bluetooth controller is down
2. Use the btproxy tool to open the listening UNIX socket, type:

   ```
   sudo tools/btproxy -u -i 0
   Listening on /tmp/bt-server-bredr
   ```

   You might need to replace `-i 0` with the index of the Controller you wish to proxy.

   If you see `Received unknown host packet type 0x00` when running QEMU, then add `-z` to the `btproxy` command line to ignore any null bytes transmitted at startup.

Once the hardware is connected and ready to use, you can then proceed to building and running a sample:

- Choose one of the Bluetooth sample applications located in `samples/bluetooth`.
- To run a Bluetooth application in QEMU, type:

  ```
  west build -b qemu_x86 samples/bluetooth/<sample>
  west build -t run
  ```

  Running QEMU now results in a connection with the second serial line to the `bt-server-bredr` UNIX socket, letting the application access the Bluetooth controller.

- To run a Bluetooth application in native_sim, first build it:

```
west build -b native_sim samples/bluetooth/<sample>
```

  And then run it with:

```
$ sudo ./build/zephyr/zephyr.exe --bt-dev=hci0
```

Using a Zephyr-based BLE Controller Depending on which hardware you have available, you can choose between two transports when building a single-mode, Zephyr-based BLE Controller:

- UART: Use the hci_uart sample and follow the instructions in bluetooth-hci-uart-qemu-posix.
- USB: Use the hci_usb sample and then treat it as a Host System Bluetooth Controller (see previous section)

HCI Tracing When running the Host on a computer connected to an external Controller, it is very useful to be able to see the full log of exchanges between the two, in the format of a *Host Controller Interface* log. In order to see those logs, you can use the built-in `btmon` tool from BlueZ:

```
$ btmon
```

The output looks like this:

```
= New Index: 00:00:00:00:00:00 (Primary,Virtual,Control)                    0.274200
= Open Index: 00:00:00:00:00:00                                             0.274500
< HCI Command: Reset (0x03|0x0003) plen 0                                #1 0.274600
> HCI Event: Command Complete (0x0e) plen 4                              #2 0.274700
      Reset (0x03|0x0003) ncmd 1
      Status: Success (0x00)
< HCI Command: Read Local Supported Features (0x04|0x0003) plen 0         #3 0.274800
> HCI Event: Command Complete (0x0e) plen 12                             #4 0.274900
      Read Local Supported Features (0x04|0x0003) ncmd 1
      Status: Success (0x00)
      Features: 0x00 0x00 0x00 0x00 0x60 0x00 0x00 0x00
         BR/EDR Not Supported
         LE Supported (Controller)
```

Embedded HCI tracing When running both Host and Controller in actual Integrated Circuits, you will only see normal log messages on the console by default, without any way of accessing the HCI traffic between the Host and the Controller. However, there is a special Bluetooth logging mode that converts the console to use a binary protocol that interleaves both normal log messages as well as the HCI traffic.

Set the following Kconfig options to enable this protocol before building your application:

```
CONFIG_BT_DEBUG_MONITOR_UART=y
CONFIG_UART_CONSOLE=n
```

- Setting `CONFIG_BT_DEBUG_MONITOR_UART` activates the formatting
- Clearing `CONFIG_UART_CONSOLE` makes the UART unavailable for the system console. E.g. for `printk` and the `boot banner`

To decode the binary protocol that will now be sent to the console UART you need to use the btmon tool from *BlueZ*:

```
$ btmon --tty <console TTY> --tty-speed 115200
```

If UART is not available (or you still want non-binary logs), you can set CONFIG_BT_DEBUG_MONITOR_RTT instead, which will use Segger RTT. For example, if trying to connect to a nRF52840DK with S/N 683578642:

```
$ btmon --jlink nRF52840_xxAA,683578642
```

Running on a Virtual Controller and native_sim

An alternative to a Bluetooth physical controller is the use of a virtual controller. This controller can be connected over an HCI TCP server. This TCP server must support the HCI H4 protocol. In comparison to the physical controller variant, the virtual controller allows to test a Zephyr application running on the native boards without a physical Bluetooth controller.

The main use case for a virtual controller is to do Bluetooth connectivity tests without the need of Bluetooth hardware. This allows to automate Bluetooth integration tests with external applications such as a Bluetooth gateway or a mobile application.

To demonstrate this functionality an example is given to interact with a virtual controller. For this purpose, the experimental python module Bumble from Google is used as it allows to create a TCP Bluetooth virtual controller and connect with the Zephyr Bluetooth host. To install bumble follow the Bumble Getting Started Guide.

Note

If your Zephyr application requires the use of the HCI LE Set extended commands, install the branch `controller-extended-advertising` from Bumble.

Android Emulator You can test the virtual controller by connecting a Bluetooth Zephyr application to the Android Emulator.

To connect your application to the Android Emulator follow the next steps:

1. Build your Zephyr application and disable the HCI ACL flow control (i.e. CONFIG_BT_HCI_ACL_FLOW_CONTROL=n) as the virtual controller from android does not support it at the moment.
2. Install Android Emulator version >= 33.1.4.0. The easiest way to do this is by installing the latest Android Studio Preview version.
3. Create a new Android Virtual Device (AVD) with the Android Device Manager. The AVD should use at least SDK API 34.
4. Run the Android Emulator via terminal as follows:

   ```
   emulator avd YOUR_AVD -packet-streamer-endpoint default
   ```

5. Create a Bluetooth bridge between the Zephyr application and the virtual controller from Android Emulator with the Bumble utility `hci-bridge`.

   ```
   bumble-hci-bridge tcp-server:_:1234 android-netsim
   ```

   This command will create a TCP server bridge on the local host IP address `127.0.0.1` and port number `1234`.

6. Run the Zephyr application and connect to the TCP server created in the last step.

   ```
   ./zephyr.exe --bt-dev=127.0.0.1:1234
   ```

After following these steps the Zephyr application will be available to the Android Emulator over the virtual Bluetooth controller that was bridged with Bumble. You can verify that the Zephyr application can communicate over Bluetooth by opening the Bluetooth settings in your AVD and

scanning for your Zephyr application device. To test this you can build the Bluetooth peripheral samples such as Peripheral HR or Peripheral DIS

Using Zephyr-based Controllers with BlueZ

If you want to test a Zephyr-powered BLE Controller using BlueZ's Bluetooth Host, you will need a few tools described in the *Using BlueZ with Zephyr* section. Once you have installed the tools you can then use them to interact with your Zephyr-based controller:

```
sudo tools/btmgmt --index 0
[hci0]# auto-power
[hci0]# find -l
```

You might need to replace `--index 0` with the index of the Controller you wish to manage. Additional information about `btmgmt` can be found in its manual pages.

6.1.11 Shell

The Bluetooth Shell is an application based on the *Shell* module. It offer a collection of commands made to easily interact with the Bluetooth stack.

Bluetooth Shell Setup and Usage

First you need to build and flash your board with the Bluetooth shell. For how to do that, see the *Getting Started Guide*. The Bluetooth shell itself is located in tests/bluetooth/shell/.

When it's done, connect to the CLI using your favorite serial terminal application. You should see the following prompt:

```
uart:~$
```

For more details on general usage of the shell, see *Shell*.

The first step is enabling Bluetooth. To do so, use the `bt init` command. The following message is printed to confirm Bluetooth has been initialized.

```
uart:~$ bt init
Bluetooth initialized
Settings Loaded
[00:02:26.771,148] <inf> fs_nvs: nvs_mount: 8 Sectors of 4096 bytes
[00:02:26.771,148] <inf> fs_nvs: nvs_mount: alloc wra: 0, fe8
[00:02:26.771,179] <inf> fs_nvs: nvs_mount: data wra: 0, 0
[00:02:26.777,984] <inf> bt_hci_core: hci_vs_init: HW Platform: Nordic Semiconductor␣
↪(0x0002)
[00:02:26.778,015] <inf> bt_hci_core: hci_vs_init: HW Variant: nRF52x (0x0002)
[00:02:26.778,045] <inf> bt_hci_core: hci_vs_init: Firmware: Standard Bluetooth controller␣
↪(0x00) Version 3.2 Build 99
[00:02:26.778,656] <inf> bt_hci_core: bt_init: No ID address. App must call settings_load()
[00:02:26.794,738] <inf> bt_hci_core: bt_dev_show_info: Identity: EB:BF:36:26:42:09 (random)
[00:02:26.794,769] <inf> bt_hci_core: bt_dev_show_info: HCI: version 5.3 (0x0c) revision␣
↪0x0000, manufacturer 0x05f1
[00:02:26.794,799] <inf> bt_hci_core: bt_dev_show_info: LMP: version 5.3 (0x0c) subver␣
↪0xffff
```

Identities

Identities are a Zephyr host concept, allowing a single physical device to behave like multiple logical Bluetooth devices.

The shell allows the creation of multiple identities, to a maximum that is set by the Kconfig symbol `CONFIG_BT_ID_MAX`. To create a new identity, use `bt id-create` command. You can then use it by selecting it with its ID `bt id-select <id>`. Finally, you can list all the available identities with `id-show`.

Scan for devices

Start scanning by using the `bt scan on` command. Depending on the environment you're in, you may see a lot of lines printed on the shell. To stop the scan, run `bt scan off`, the scrolling should stop.

Here is an example of what you can expect:

```
uart:~$ bt scan on
Bluetooth active scan enabled
[DEVICE]: CB:01:1A:2D:6E:AE (random), AD evt type 0, RSSI -78  C:1 S:1 D:0 SR:0 E:0 Prim:␣
↪LE 1M, Secn: No packets, Interval: 0x0000 (0 us), SID: 0xff
[DEVICE]: 20:C2:EE:59:85:5B (random), AD evt type 3, RSSI -62  C:0 S:0 D:0 SR:0 E:0 Prim:␣
↪LE 1M, Secn: No packets, Interval: 0x0000 (0 us), SID: 0xff
[DEVICE]: E3:72:76:87:2F:E8 (random), AD evt type 3, RSSI -74  C:0 S:0 D:0 SR:0 E:0 Prim:␣
↪LE 1M, Secn: No packets, Interval: 0x0000 (0 us), SID: 0xff
[DEVICE]: 1E:19:25:8A:CB:84 (random), AD evt type 3, RSSI -67  C:0 S:0 D:0 SR:0 E:0 Prim:␣
↪LE 1M, Secn: No packets, Interval: 0x0000 (0 us), SID: 0xff
[DEVICE]: 26:42:F3:D5:A0:86 (random), AD evt type 3, RSSI -73  C:0 S:0 D:0 SR:0 E:0 Prim:␣
↪LE 1M, Secn: No packets, Interval: 0x0000 (0 us), SID: 0xff
[DEVICE]: 0C:61:D1:B9:5D:9E (random), AD evt type 3, RSSI -87  C:0 S:0 D:0 SR:0 E:0 Prim:␣
↪LE 1M, Secn: No packets, Interval: 0x0000 (0 us), SID: 0xff
[DEVICE]: 20:C2:EE:59:85:5B (random), AD evt type 3, RSSI -66  C:0 S:0 D:0 SR:0 E:0 Prim:␣
↪LE 1M, Secn: No packets, Interval: 0x0000 (0 us), SID: 0xff
[DEVICE]: 25:3F:7A:EE:0F:55 (random), AD evt type 3, RSSI -83  C:0 S:0 D:0 SR:0 E:0 Prim:␣
↪LE 1M, Secn: No packets, Interval: 0x0000 (0 us), SID: 0xff
uart:~$ bt scan off
Scan successfully stopped
```

As you can see, this can lead to a high number of results. To reduce that number and easily find a specific device, you can enable scan filters. There are four types of filters: by name, by RSSI, by address and by periodic advertising interval. To apply a filter, use the `bt scan-set-filter` command followed by the type of filters. You can add multiple filters by using the commands again.

For example, if you want to look only for devices with the name *test shell*:

```
uart:~$ bt scan-filter-set name "test shell"
```

Or if you want to look for devices at a very close range:

```
uart:~$ bt scan-filter-set rssi -40
RSSI cutoff set at -40 dB
```

Finally, if you want to remove all filters:

```
uart:~$ bt scan-filter-clear all
```

You can use the command `bt scan on` to create an *active* scanner, meaning that the scanner will ask the advertisers for more information by sending a *scan request* packet. Alternatively, you can create a *passive scanner* by using the `bt scan passive` command, so the scanner will not ask the advertiser for more information.

Connecting to a device

To connect to a device, you need to know its address and type of address and use the `bt connect` command with the address and the type as arguments.

Here is an example:

```
uart:~$ bt connect 52:84:F6:BD:CE:48 random
Connection pending
Connected: 52:84:F6:BD:CE:48 (random)
Remote LMP version 5.3 (0x0c) subversion 0xffff manufacturer 0x05f1
LE Features: 0x000000000001412f
LE PHY updated: TX PHY LE 2M, RX PHY LE 2M
LE conn  param req: int (0x0018, 0x0028) lat 0 to 42
LE conn param updated: int 0x0028 lat 0 to 42
```

You can list the active connections of the shell using the `bt connections` command. The shell maximum number of connections is defined by `CONFIG_BT_MAX_CONN`. You can disconnect from a connection with the `bt disconnect <address: XX:XX:XX:XX:XX:XX> <type: (public|random)>` command.

Note

If you were scanning just before, you can connect to the last scanned device by simply running the `bt connect` command.

Alternatively, you can use the `bt connect-name <name>` command to automatically enable scanning with a name filter and connect to the first match.

Advertising

Begin advertising by using the `bt advertise on` command. This will use the default parameters and advertise a resolvable private address with the name of the device. You can choose to use the identity address instead by running the `bt advertise on identity` command. To stop advertising use the `bt advertise off` command.

To enable more advanced features of advertising, you should create an advertiser using the `bt adv-create` command. Parameters for the advertiser can be passed either at the creation of it or by using the `bt adv-param` command. To begin advertising with this newly created advertiser, use the `bt adv-start` command, and then the `bt adv-stop` command to stop advertising.

When using the custom advertisers, you can choose if it will be connectable or scannable. This leads to four options: `conn-scan`, `conn-nscan`, `nconn-scan` and `nconn-nscan`. Those parameters are mandatory when creating an advertiser or updating its parameters.

For example, if you want to create a connectable and scannable advertiser and start it:

```
uart:~$ bt adv-create conn-scan
Created adv id: 0, adv: 0x200022f0
uart:~$ bt adv-start
Advertiser[0] 0x200022f0 set started
```

You may notice that with this, the custom advertiser does not advertise the device name; you need to add it. Continuing from the previous example:

```
uart:~$ bt adv-stop
Advertiser set stopped
uart:~$ bt adv-data dev-name
uart:~$ bt adv-start
Advertiser[0] 0x200022f0 set started
```

You should now see the name of the device in the advertising data. You can also set a custom name by using `name <custom name>` instead of `dev-name`. It is also possible to set the advertising data manually with the `bt adv-data` command. The following example shows how to set the advertiser name with it using raw advertising data:

```
uart:~$ bt adv-create conn-scan
Created adv id: 0, adv: 0x20002348
uart:~$ bt adv-data 1009426C7565746F6F74682D5368656C6C
uart:~$ bt adv-start
Advertiser[0] 0x20002348 set started
```

The data must be formatted according to the Bluetooth Core Specification (see version 5.3, vol. 3, part C, 11). In this example, the first octet is the size of the data (the data and one octet for the data type), the second one is the type of data, `0x09` is the Complete Local Name and the remaining data are the name in ASCII. So, on the other device you should see the name *Bluetooth-Shell*.

When advertising, if others devices use an *active* scanner, you may receive *scan request* packets. To visualize those packets, you can add `scan-reports` to the parameters of your advertiser.

Directed Advertising It is possible to use directed advertising on the shell if you want to reconnect to a device. The following example demonstrates how to create a directed advertiser with the address specified right after the parameter `directed`. The `low` parameter indicates that we want to use the low duty cycle mode, and the `dir-rpa` parameter is required if the remote device is privacy-enabled and supports address resolution of the target address in directed advertisement.

```
uart:~$ bt adv-create conn-scan directed D7:54:03:CE:F3:B4 random low dir-rpa
Created adv id: 0, adv: 0x20002348
```

After that, you can start the advertiser and then the target device will be able to reconnect.

Extended Advertising Let's now have a look at some extended advertising features. To enable extended advertising, use the *ext-adv* parameter.

```
uart:~$ bt adv-create conn-nscan ext-adv name-ad
Created adv id: 0, adv: 0x200022f0
uart:~$ bt adv-start
Advertiser[0] 0x200022f0 set started
```

This will create an extended advertiser, that is connectable and non-scannable.

Encrypted Advertising Data Zephyr has support for the Encrypted Advertising Data feature. The `bt encrypted-ad` sub-commands allow managing the advertising data of a given advertiser.

To encrypt the advertising data, key materials need to be provided, that can be done with `bt encrypted-ad set-keys <session key> <init vector>`. The session key is 16 bytes long and the initialisation vector is 8 bytes long.

You can add advertising data by using `bt encrypted-ad add-ad` and `bt encrypted-ad add-ead`. The former will take add one advertising data structure (as defined in the Core Specification), when the later will read the given data, encrypt them and then add the generated encrypted advertising data structure. It's possible to mix encrypted and non-encrypted data, when done adding advertising data, `bt encrypted-ad commit-ad` can be used to apply the change to the data to the selected advertiser. After that the advertiser can be started as described previously. It's possible to clear the advertising data by using `bt encrypted-ad clear-ad`.

On the Central side, it's possible to decrypt the received encrypted advertising data by setting the correct keys material as described earlier and then enabling the decrypting of the data with `bt encrypted-ad decrypt-scan on`.

Note

To see the advertising data in the scan report `bt scan-verbose-output` need to be enabled.

Note

It's possible to increase the length of the advertising data by increasing the value of `CONFIG_BT_CTLR_ADV_DATA_LEN_MAX` and `CONFIG_BT_CTLR_SCAN_DATA_LEN_MAX`.

Here is a simple example demonstrating the usage of EAD:

Peripheral

```
uart:~$ bt init
...
uart:~$ bt adv-create conn-nscan ext-adv
Created adv id: 0, adv: 0x81769a0
uart:~$ bt encrypted-ad set-keys 9ba22d3824efc70feb800c80294cba38 2e83f3d4d47695b6
session key set to:
00000000: 9b a2 2d 38 24 ef c7 0f  eb 80 0c 80 29 4c ba 38 |..-8$... ....)L.8|
initialisation vector set to:
00000000: 2e 83 f3 d4 d4 76 95 b6                          |.....v..         |
uart:~$ bt encrypted-ad add-ad 06097368656C6C
uart:~$ bt encrypted-ad add-ead 03ffdead03ffbeef
uart:~$ bt encrypted-ad commit-ad
Advertising data for Advertiser[0] 0x81769a0 updated.
uart:~$ bt adv-start
Advertiser[0] 0x81769a0 set started
```

Central

```
uart:~$ bt init
...
uart:~$ bt scan-verbose-output on
uart:~$ bt encrypted-ad set-keys 9ba22d3824efc70feb800c80294cba38 2e83f3d4d47695b6
session key set to:
00000000: 9b a2 2d 38 24 ef c7 0f  eb 80 0c 80 29 4c ba 38 |..-8$... ....)L.8|
initialisation vector set to:
00000000: 2e 83 f3 d4 d4 76 95 b6                          |.....v..         |
uart:~$ bt encrypted-ad decrypt-scan on
Received encrypted advertising data will now be decrypted using provided key materials.
uart:~$ bt scan on
Bluetooth active scan enabled
[DEVICE]: 68:49:30:68:49:30 (random), AD evt type 5, RSSI -59   shell C:1 S:0 D:0 SR:0 E:1␣
↪Prim: LE 1M, Secn: LE 2M, Interval: 0x0000 (0 us), SID: 0x0
        [SCAN DATA START - EXT_ADV]
        Type 0x09:    shell
        Type 0x31: Encrypted Advertising Data: 0xe2, 0x17, 0xed, 0x04, 0xe7, 0x02, 0x1d,␣
↪0xc9, 0x40, 0x07, uart:~0x18, 0x90, 0x6c, 0x4b, 0xfe, 0x34, 0xad
        [START DECRYPTED DATA]
        Type 0xff: 0xde, 0xad
        Type 0xff: 0xbe, 0xef
        [END DECRYPTED DATA]
        [SCAN DATA END]
...
```

Filter Accept List

It's possible to create a list of allowed addresses that can be used to connect to those addresses automatically. Here is how to do it:

```
uart:~$ bt fal-add 47:38:76:EA:29:36 random
uart:~$ bt fal-add 66:C8:80:2A:05:73 random
uart:~$ bt fal-connect on
```

The shell will then connect to the first available device. In the example, if both devices are advertising at the same time, we will connect to the first address added to the list.

The Filter Accept List can also be used for scanning or advertising by using the option `fal`. For example, if we want to scan for a bunch of selected addresses, we can set up a Filter Accept List:

```
uart:~$ bt fal-add 65:4B:9E:83:AF:73 random
uart:~$ bt fal-add 73:72:82:B4:8F:B9 random
uart:~$ bt fal-add 5D:85:50:1C:72:64 random
uart:~$ bt scan on fal
```

You should see only those three addresses reported by the scanner.

Enabling security

When connected to a device, you can enable multiple levels of security, here is the list for Bluetooth LE:

- **1** No encryption and no authentication;
- **2** Encryption and no authentication;
- **3** Encryption and authentication;
- **4** Bluetooth LE Secure Connection.

To enable security, use the `bt security <level>` command. For levels requiring authentication (level 3 and above), you must first set the authentication method. To do it, you can use the `bt auth all` command. After that, when you will set the security level, you will be asked to confirm the passkey on both devices. On the shell side, do it with the command `bt auth-passkey-confirm`.

Pairing Enabling authentication requires the devices to be bondable. By default the shell is bondable. You can make the shell not bondable using `bt bondable off`. You can list all the devices you are paired with using the command `bt bonds`.

The maximum number of paired devices is set using `CONFIG_BT_MAX_PAIRED`. You can remove a paired device using `bt clear <address: XX:XX:XX:XX:XX:XX> <type: (public|random)>` or remove all paired devices with the command `bt clear all`.

GATT

The following examples assume that you have two devices already connected.

To perform service discovery on the client side, use the `gatt discover` command. This should print all the services that are available on the GATT server.

On the server side, you can register pre-defined test services using the `gatt register` command. When done, you should see the newly added services on the client side when running the discovery command.

You can now subscribe to those new services on the client side. Here is an example on how to subscribe to the test service:

```
uart:~$ gatt subscribe 26 25
Subscribed
```

The server can now notify the client with the command `gatt notify`.

Another option available through the GATT command is initiating the MTU exchange. To do it, use the `gatt exchange-mtu` command. To update the shell maximum MTU, you need to update Kconfig symbols in the configuration file of the shell. For more details, see bluetooth_mtu_update_sample.

L2CAP

The `l2cap` command exposes parts of the L2CAP API. The following example shows how to register a LE PSM, connect to it from another device and send 3 packets of 14 octets each.

The example assumes that the two devices are already connected.

On device A, register the LE PSM:

```
uart:~$ l2cap register 29
L2CAP psm 41 sec_level 1 registered
```

On device B, connect to the registered LE PSM and send data:

```
uart:~$ l2cap connect 29
Chan sec: 1
L2CAP connection pending
Channel 0x20000210 connected
Channel 0x20000210 status 1
uart:~$ l2cap send 3 14
Rem 2
Rem 1
Rem 0
Outgoing data channel 0x20000210 transmitted
Outgoing data channel 0x20000210 transmitted
Outgoing data channel 0x20000210 transmitted
```

On device A, you should have received the data:

```
Incoming conn 0x20002398
Channel 0x20000210 status 1
Channel 0x20000210 connected
Channel 0x20000210 requires buffer
Incoming data channel 0x20000210 len 14
00000000: ff ff ff ff ff ff ff ff  ff ff ff ff ff ff       |........ ......  |
Channel 0x20000210 requires buffer
Incoming data channel 0x20000210 len 14
00000000: ff ff ff ff ff ff ff ff  ff ff ff ff ff ff       |........ ......  |
Channel 0x20000210 requires buffer
Incoming data channel 0x20000210 len 14
00000000: ff ff ff ff ff ff ff ff  ff ff ff ff ff ff       |........ ......  |
```

A2DP

The `a2dp` command exposes parts of the A2DP API.

The following examples assume that you have two devices already connected.

Here is a example connecting two devices:

- Source and Sink sides register a2dp callbacks. using `a2dp register_cb`.

- Source and Sink sides register stream endpoints. using `a2dp register_ep source sbc` and `a2dp register_ep sink sbc`.
- Source establish A2dp connection. It will create the AVDTP Signaling and Media L2CAP channels. using `a2dp connect`.
- Source and Sink side can discover remote device's stream endpoints. using `a2dp discover_peer_eps`
- Source or Sink configure the stream to create the stream after discover remote's endpoints. using `a2dp configure`.
- Source or Sink establish the stream. using `a2dp establish`.
- Source or Sink start the media. using `a2dp start`.
- Source test the media sending. using `a2dp send_media` to send one test packet data.

Device A (Audio Source Side)

```
uart:~$ a2dp register_cb
success
uart:~$ a2dp register_ep source sbc
SBC source endpoint is registered
uart:~$ a2dp connect
Bonded with XX:XX:XX:XX:XX:XX
Security changed: XX:XX:XX:XX:XX:XX level 2
a2dp connected
uart:~$ a2dp discover_peer_eps
endpoint id: 1, (sink), (idle):
  codec type: SBC
  sample frequency:
          44100
          48000
  channel mode:
          Mono
          Stereo
          Joint-Stereo
  Block Length:
          16
  Subbands:
          8
  Allocation Method:
          Loudness
  Bitpool Range: 18 - 35
uart:~$ a2dp configure
success to configure
stream configured
uart:~$ a2dp establish
success to establish
stream established
uart:~$ a2dp start
success to start
stream started
uart:~$ a2dp send_media
frames num: 1, data length: 160
data: 1, 2, 3, 4, 5, 6 ......
```

Device B (Audio Sink Side)

```
uart:~$ a2dp register_cb
success
uart:~$ a2dp register_ep sink sbc
SBC sink endpoint is registered
<after a2dp connect>
```

(continues on next page)

(continued from previous page)

```
Connected: XX:XX:XX:XX:XX:XX
Bonded with XX:XX:XX:XX:XX:XX
Security changed: XX:XX:XX:XX:XX:XX level 2
a2dp connected
<after a2dp configure of source side>
receive requesting config and accept
SBC configure success
sample rate 44100Hz
stream configured
<after a2dp establish of source side>
receive requesting establishment and accept
stream established
<after a2dp start of source side>
receive requesting start and accept
stream started
<after a2dp send_media of source side>
received, num of frames: 1, data length: 160
data: 1, 2, 3, 4, 5, 6 ......
...
```

Logging

You can configure the logging level per module at runtime. This depends on the maximum logging level that is compiled in. To configure, use the `log` command. Here are some examples:

- List the available modules and their current logging level

```
uart:~$ log status
```

- Disable logging for *bt_hci_core*

```
uart:~$ log disable bt_hci_core
```

- Enable error logs for *bt_att* and *bt_smp*

```
uart:~$ log enable err bt_att bt_smp
```

- Disable logging for all modules

```
uart:~$ log disable
```

- Enable warning logs for all modules

```
uart:~$ log enable wrn
```

6.2 Controller Area Network (CAN) Bus Protocols

6.2.1 ISO-TP Transport Protocol

- *Overview*
- *API Reference*

Overview

ISO-TP is a transport protocol defined in the ISO-Standard ISO15765-2 Road vehicles - Diagnostic communication over Controller Area Network (DoCAN). Part2: Transport protocol and network layer services. As its name already implies, it is originally designed, and still used in road vehicle diagnostic over Controller Area Networks. Nevertheless, it's not limited to applications in road vehicles or the automotive domain.

This transport protocol extends the limited payload data size for classical CAN (8 bytes) and CAN FD (64 bytes) to theoretically four gigabytes. Additionally, it adds a flow control mechanism to influence the sender's behavior. ISO-TP segments packets into small fragments depending on the payload size of the CAN frame. The header of those segments is called Protocol Control Information (PCI).

Packets smaller or equal to seven bytes on Classical CAN are called single-frames (SF). They don't need to fragment and do not have any flow-control.

Packets larger than that are segmented into a first-frame (FF) and as many consecutive-frames (CF) as required. The FF contains information about the length of the entire payload data and additionally, the first few bytes of payload data. The receiving peer sends back a flow-control-frame (FC) to either deny, postpone, or accept the following consecutive frames. The FC also defines the conditions of sending, namely the block-size (BS) and the minimum separation time between frames (STmin). The block size defines how many CF the sender is allowed to send, before he has to wait for another FC.

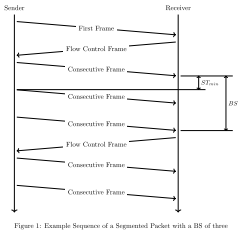


Figure 1: Example Sequence of a Segmented Packet with a BS of three

API Reference

Related code samples

ISO-TP library
Use ISO-TP library to exchange messages between two boards.

group `can_isotp`

CAN ISO-TP Protocol.

ISO-TP message ID flags

`ISOTP_MSG_EXT_ADDR`

Message uses ISO-TP extended addressing (first payload byte of CAN frame)

`ISOTP_MSG_FIXED_ADDR`

Message uses ISO-TP fixed addressing (according to SAE J1939).

Only valid in combination with `ISOTP_MSG_IDE`.

ISOTP_MSG_IDE
: Message uses extended (29-bit) CAN ID.

ISOTP_MSG_FDF
: Message uses CAN FD format (FDF)

ISOTP_MSG_BRS
: Message uses CAN FD Baud Rate Switch (BRS).

  Only valid in combination with `ISOTP_MSG_FDF`.

Defines

ISOTP_N_OK
: Completed successfully.

ISOTP_N_TIMEOUT_A
: Ar/As has timed out.

ISOTP_N_TIMEOUT_BS
: Reception of next FC has timed out.

ISOTP_N_TIMEOUT_CR
: Cr has timed out.

ISOTP_N_WRONG_SN
: Unexpected sequence number.

ISOTP_N_INVALID_FS
: Invalid flow status received.

ISOTP_N_UNEXP_PDU
: Unexpected PDU received.

ISOTP_N_WFT_OVRN
: Maximum number of WAIT flowStatus PDUs exceeded.

ISOTP_N_BUFFER_OVERFLW
: FlowStatus OVFLW PDU was received.

ISOTP_N_ERROR
: General error.

ISOTP_NO_FREE_FILTER
: Implementation specific errors.

  Can’t bind or send because the CAN device has no filter left

ISOTP_NO_NET_BUF_LEFT
: No net buffer left to allocate.

ISOTP_NO_BUF_DATA_LEFT
: Not sufficient space in the buffer left for the data.

ISOTP_NO_CTX_LEFT
: No context buffer left to allocate.

ISOTP_RECV_TIMEOUT
: Timeout for recv.

ISOTP_FIXED_ADDR_SA_POS
: Position of fixed source address (SA)

ISOTP_FIXED_ADDR_SA_MASK
: Mask to obtain fixed source address (SA)

ISOTP_FIXED_ADDR_TA_POS
: Position of fixed target address (TA)

ISOTP_FIXED_ADDR_TA_MASK
: Mask to obtain fixed target address (TA)

ISOTP_FIXED_ADDR_PRIO_POS
: Position of priority in fixed addressing mode.

ISOTP_FIXED_ADDR_PRIO_MASK
: Mask for priority in fixed addressing mode.

ISOTP_FIXED_ADDR_RX_MASK
: CAN filter RX mask to match any priority and source address (SA)

Typedefs

typedef void (*isotp_tx_callback_t)(int error_nr, void *arg)

Transmission callback.

This callback is called when a transmission is completed.

Param error_nr
: ISOTP_N_OK on success, ISOTP_N_* on error

Param arg
: Callback argument passed to the send function

Functions

int **isotp_bind**(struct isotp_recv_ctx *rctx, const struct *device* *can_dev, const struct *isotp_msg_id* *rx_addr, const struct *isotp_msg_id* *tx_addr, const struct *isotp_fc_opts* *opts, *k_timeout_t* timeout)

Bind an address to a receiving context.

This function binds an RX and TX address combination to an RX context. When data arrives from the specified address, it is buffered and can be read by calling isotp_recv. When calling this routine, a filter is applied in the CAN device, and the context is initialized. The context must be valid until calling unbind.

Parameters

- **rctx** – Context to store the internal states.
- **can_dev** – The CAN device to be used for sending and receiving.
- **rx_addr** – Identifier for incoming data.
- **tx_addr** – Identifier for FC frames.
- **opts** – Flow control options.
- **timeout** – Timeout for FF SF buffer allocation.

Return values

- **ISOTP_N_OK** – on success
- **ISOTP_NO_FREE_FILTER** – if CAN device has no filters left.

void **isotp_unbind**(struct isotp_recv_ctx *rctx)

Unbind a context from the interface.

This function removes the binding from isotp_bind. The filter is detached from the CAN device, and if a transmission is ongoing, buffers are freed. The context can be discarded safely after calling this function.

Parameters

- **rctx** – Context that should be unbound.

int **isotp_recv**(struct isotp_recv_ctx *rctx, uint8_t *data, size_t len, *k_timeout_t* timeout)

Read out received data from fifo.

This function reads the data from the receive FIFO of the context. It blocks if the FIFO is empty. If an error occurs, the function returns a negative number and leaves the data buffer unchanged.

Parameters

- **rctx** – Context that is already bound.
- **data** – Pointer to a buffer where the data is copied to.
- **len** – Size of the buffer.
- **timeout** – Timeout for incoming data.

Return values

- **Number** – of bytes copied on success
- **ISOTP_RECV_TIMEOUT** – when “timeout” timed out
- **ISOTP_N_*** – on error

int **isotp_recv_net**(struct isotp_recv_ctx *rctx, struct *net_buf* **buffer, *k_timeout_t* timeout)

Get the net buffer on data reception.

This function reads incoming data into net-buffers. It blocks until the entire packet is received, BS is reached, or an error occurred. If BS was zero, the data is in a single *net_buf*. Otherwise, the data is fragmented in chunks of BS size. The net-buffers are referenced and must be freed with net_buf_unref after the data is processed.

Parameters

- `rctx` – Context that is already bound.
- `buffer` – Pointer where the *net_buf* pointer is written to.
- `timeout` – Timeout for incoming data.

Return values

- `Remaining` – data length for this transfer if BS > 0, 0 for BS = 0
- `ISOTP_RECV_TIMEOUT` – when "timeout" timed out
- `ISOTP_N_*` – on error

int `isotp_send`(struct isotp_send_ctx *sctx, const struct *device* *can_dev, const uint8_t *data, size_t len, const struct *isotp_msg_id* *tx_addr, const struct *isotp_msg_id* *rx_addr, *isotp_tx_callback_t* complete_cb, void *cb_arg)

Send data.

This function is used to send data to a peer that listens to the tx_addr. An internal work-queue is used to transfer the segmented data. Data and context must be valid until the transmission has finished. If a complete_cb is given, this function is non-blocking, and the callback is called on completion with the return value as a parameter.

Parameters

- `sctx` – Context to store the internal states.
- `can_dev` – The CAN device to be used for sending and receiving.
- `data` – Data to be sent.
- `len` – Length of the data to be sent.
- `rx_addr` – Identifier for FC frames.
- `tx_addr` – Identifier for outgoing frames the receiver listens on.
- `complete_cb` – Function called on completion or NULL.
- `cb_arg` – Argument passed to the complete callback.

Return values

- `ISOTP_N_OK` – on success
- `ISOTP_N_*` – on error

struct `isotp_msg_id`

#include <isotp.h> ISO-TP message id struct.

Used to pass addresses to the bind and send functions.

Public Members

union isotp_msg_id

CAN identifier.

If ISO-TP fixed addressing is used, isotp_bind ignores SA and priority sections and modifies TA section in flow control frames.

uint8_t ext_addr

ISO-TP extended address (if used)

uint8_t dl

ISO-TP frame data length (TX_DL for TX address or RX_DL for RX address).

Valid values are 8 for classical CAN or 8, 12, 16, 20, 24, 32, 48 and 64 for CAN FD.

0 will be interpreted as 8 or 64 (if ISOTP_MSG_FDF is set).

The value for incoming transmissions (RX_DL) is determined automatically based on the received first frame and does not need to be set during initialization.

uint8_t flags

Flags.

See also

ISOTP_MSG_FLAGS.

struct isotp_fc_opts

#include <*isotp.h*> ISO-TP frame control options struct.

Used to pass the options to the bind and send functions.

Public Members

uint8_t bs

Block size.

Number of CF PDUs before next CF is sent

uint8_t stmin

Minimum separation time.

Min time between frames

6.3 Networking

The networking section contains information regarding the network stack of the Zephyr kernel. Use the information to understand the principles behind the operation of the stacks and how they were implemented.

6.3.1 Overview

- *Supported Features*
- *Source Tree Layout*

Supported Features

The networking IP stack is modular and highly configurable via build-time configuration options. You can minimize system memory consumption by enabling only those network features required by your application. Almost all features can be disabled if not needed.

- **IPv6** The support for IPv6 is enabled by default. Various IPv6 sub-options can be enabled or disabled depending on networking needs.
  - Developer can set the number of unicast and multicast IPv6 addresses that are active at the same time.
  - The IPv6 address for the device can be set either statically or dynamically using SLAAC (Stateless Address Auto Configuration) (RFC 4862).
  - The system also supports multiple IPv6 prefixes and the maximum IPv6 prefix count can be configured at build time.
  - The IPv6 neighbor cache can be disabled if not needed, and its size can be configured at build time.
  - The IPv6 neighbor discovery support (RFC 4861) is enabled by default.
  - Multicast Listener Discovery v2 support (RFC 3810) is enabled by default.
  - IPv6 header compression (6lo) is available for IPv6 connectivity for IEEE 802.15.4 networks (RFC 4944).
- **IPv4** The legacy IPv4 is supported by the networking stack. It cannot be used by IEEE 802.15.4 as this network technology supports only IPv6. IPv4 can be used in Ethernet based networks. By default IPv4 support is disabled.
  - DHCP (Dynamic Host Configuration Protocol) client is supported (RFC 2131).
  - The IPv4 address can also be configured manually. Static IPv4 addresses are supported by default.
- **Dual stack support.** The networking stack allows a developer to configure the system to use both IPv6 and IPv4 at the same time.
- **UDP** User Datagram Protocol (RFC 768) is supported. The developer can send UDP datagrams (client side support) or create a listener to receive UDP packets destined to certain port (server side support).
- **TCP** Transmission Control Protocol (RFC 793) is supported. Both server and client roles can be used the application. The amount of TCP sockets that are available to applications can be configured at build time.
- **BSD Sockets API** Support for a subset of a *BSD sockets compatible API* is implemented. Both blocking and non-blocking datagram (UDP) and stream (TCP) sockets are supported.
- **Secure Sockets API** Experimental support for TLS/DTLS secure protocols and configuration options for sockets API. Secure functions for the implementation are provided by mbedTLS library.
- **MQTT** Message Queue Telemetry Transport (ISO/IEC PRF 20922) is supported. A sample mqtt-publisher client application for MQTT v3.1.1 is implemented.

- **CoAP** Constrained Application Protocol (RFC 7252) is supported. Both coap-client and coap-server sample applications are implemented.
- **LWM2M** OMA Lightweight Machine-to-Machine Protocol (LwM2M specification 1.0.2) is supported via the "Bootstrap", "Client Registration", "Device Management & Service Enablement" and "Information Reporting" interfaces. The required core LwM2M objects are implemented as well as several IPSO Smart Objects. (LwM2M specification 1.1.1) is supported in similar manner when enabled with a Kconfig option. lwm2m-client sample implements the library as an example.
- **HTTP** Hypertext Transfer Protocol client and server are supported. *HTTP Client* library supports HTTP/1.1 (RFC 2616). *HTTP Server* library supports HTTP/1.1 (RFC 2616) and HTTP/2 (RFC 9113). sockets-http-client and sockets-http-server samples are provided.
- **DNS** Domain Name Service (RFC 1035) client functionality is supported. Applications can use the DNS API to query domain name information or IP addresses from the DNS server. Both IPv4 (A) and IPv6 (AAAA) records can be queried. Both multicast DNS (mDNS) (RFC 6762) and link-local multicast name resolution (LLMNR) (RFC 4795) are supported.
- **Network Management API.** Applications can use network management API to listen management events generated by core stack when for example IP address is added to the device, or network interface is coming up etc.
- **Wi-Fi Management API.** Applications can use Wi-Fi management API to manage the interface, in example to connect to Wi-Fi network and to scan available Wi-Fi networks.
- **Wi-Fi Network Manager API.** Wi-Fi Network Managers can now register themselves to the Wi-Fi stack. The Network Managers can then implement the Wi-Fi Management API and manage the Wi-Fi interface.
- **Multiple Network Technologies.** The Zephyr OS can be configured to support multiple network technologies at the same time simply by enabling them in Kconfig: for example, Ethernet, Wi-Fi and 802.15.4 support. Note that no automatic IP routing functionality is provided between these technologies. Applications can send data according to their needs to desired network interface.
- **Minimal Copy Network Buffer Management.** It is possible to have minimal copy network data path. This means that the system tries to avoid copying application data when it is sent to the network.
- **Virtual LAN support.** Virtual LANs (VLANs) allow partitioning of physical ethernet networks into logical networks. See *VLAN support* for more details.
- **Network traffic classification.** The sent and received network packets can be prioritized depending on application needs. See *traffic classification* for more details.
- **Time Sensitive Networking.** The gPTP (generalized Precision Time Protocol) is supported. See *gPTP support* for more details.
- **Network shell.** The network shell provides helpers for figuring out network status, enabling/disabling features, and issuing commands like ping or DNS resolving. The net-shell is useful when developing network software. See *network shell* for more details.

Additionally these network technologies (link layers) are supported in Zephyr OS v1.7 and later:

- IEEE 802.15.4
- Bluetooth
- Ethernet
- SLIP (IP over serial line). Used for testing with QEMU. It provides ethernet interface to host system (like Linux) and test applications can be run in Linux host and send network data to Zephyr OS device.

Source Tree Layout

The networking stack source code tree is organized as follows:

`subsys/net/ip/`
: This is where the IP stack code is located.

`subsys/net/l2/`
: This is where the IP stack layer 2 code is located. This includes generic support for Ethernet, IEEE 802.15.4 and Wi-Fi.

`subsys/net/lib/`
: Application-level protocols (DNS, MQTT, etc.) and additional stack components (BSD Sockets, etc.).

`include/net/`
: Public API header files. These are the header files applications need to include to use IP networking functionality.

`samples/net/`
: Sample networking code. This is a good reference to get started with network application development.

`tests/net/`
: Test applications. These applications are used to verify the functionality of the IP stack, but are not the best source for sample code (see `samples/net` instead).

6.3.2 Network Stack Architecture

Network Packet Processing Statistics

This page describes how to get information about network packet processing statistics inside network stack.

Network stack contains infrastructure to figure out how long the network packet processing takes either in sending or receiving path. There are two Kconfig options that control this. For transmit (TX) path the option is called `CONFIG_NET_PKT_TXTIME_STATS` and for receive (RX) path the options is called `CONFIG_NET_PKT_RXTIME_STATS`. Note that for TX, all kind of network packet statistics is collected. For RX, only UDP, TCP or raw packet type network packet statistics is collected.

After enabling these options, the *net stats* network shell command will show this information:

```
Avg TX net_pkt (11484) time 67 us
Avg RX net_pkt (11474) time 43 us
```

Note

The values above and below are from emulated qemu_x86 board and UDP traffic

The TX time tells how long it took for network packet from its creation to when it was sent to the network. The RX time tells the time from its creation to when it was passed to the application. The values are in microseconds. The statistics will be collected per traffic class if there are more than one transmit or receive queues defined in the system. These are controlled by `CONFIG_NET_TC_TX_COUNT` and `CONFIG_NET_TC_RX_COUNT` options.

If you enable `CONFIG_NET_PKT_TXTIME_STATS_DETAIL` or `CONFIG_NET_PKT_RXTIME_STATS_DETAIL` options, then additional information for TX or RX network packets are collected when the network packet traverses the IP stack.

After enabling these options, the *net stats* will show this information:

```
Avg TX net_pkt (18902) time 63 us    [0->22->15->23=60 us]
Avg RX net_pkt (18892) time 42 us    [0->9->6->11->13=39 us]
```

The numbers inside the brackets contain information how many microseconds it took for a network packet to go from previous state to next.

In the TX example above, the values are averages over **18902** packets and contain this information:

- Packet was created by application so the time is **0**.
- Packet is about to be placed to transmit queue. The time it took from network packet creation to this state, is **22** microseconds in this example.
- The correct TX thread is invoked, and the packet is read from the transmit queue. It took **15** microseconds from previous state.
- The network packet was just sent and the network stack is about to free the network packet. It took **23** microseconds from previous state.
- In total it took on average **60** microseconds to get the network packet sent. The value **63** tells also the same information, but is calculated differently so there is slight difference because of rounding errors.

In the RX example above, the values are averages over **18892** packets and contain this information:

- Packet was created network device driver so the time is **0**.
- Packet is about to be placed to receive queue. The time it took from network packet creation to this state, is **9** microseconds in this example.
- The correct RX thread is invoked, and the packet is read from the receive queue. It took **6** microseconds from previous state.
- The network packet is then processed and placed to correct socket queue. It took **11** microseconds from previous state.
- The last value tells how long it took from there to the application. Here the value is **13** microseconds.
- In total it took on average **39** microseconds to get the network packet sent. The value **42** tells also the same information, but is calculated differently so there is slight difference because of rounding errors.

The Zephyr network stack is a native network stack specifically designed for Zephyr OS. It consists of layers, each meant to provide certain services to other layers. Network stack functionality is highly configurable via Kconfig options.

- *High level overview of the network stack*
- *Network data flow*
  - *Data receiving (RX)*
  - *Data sending (TX)*
- *Network packet processing statistics*

High level overview of the network stack

The network stack is layered and consists of the following parts:

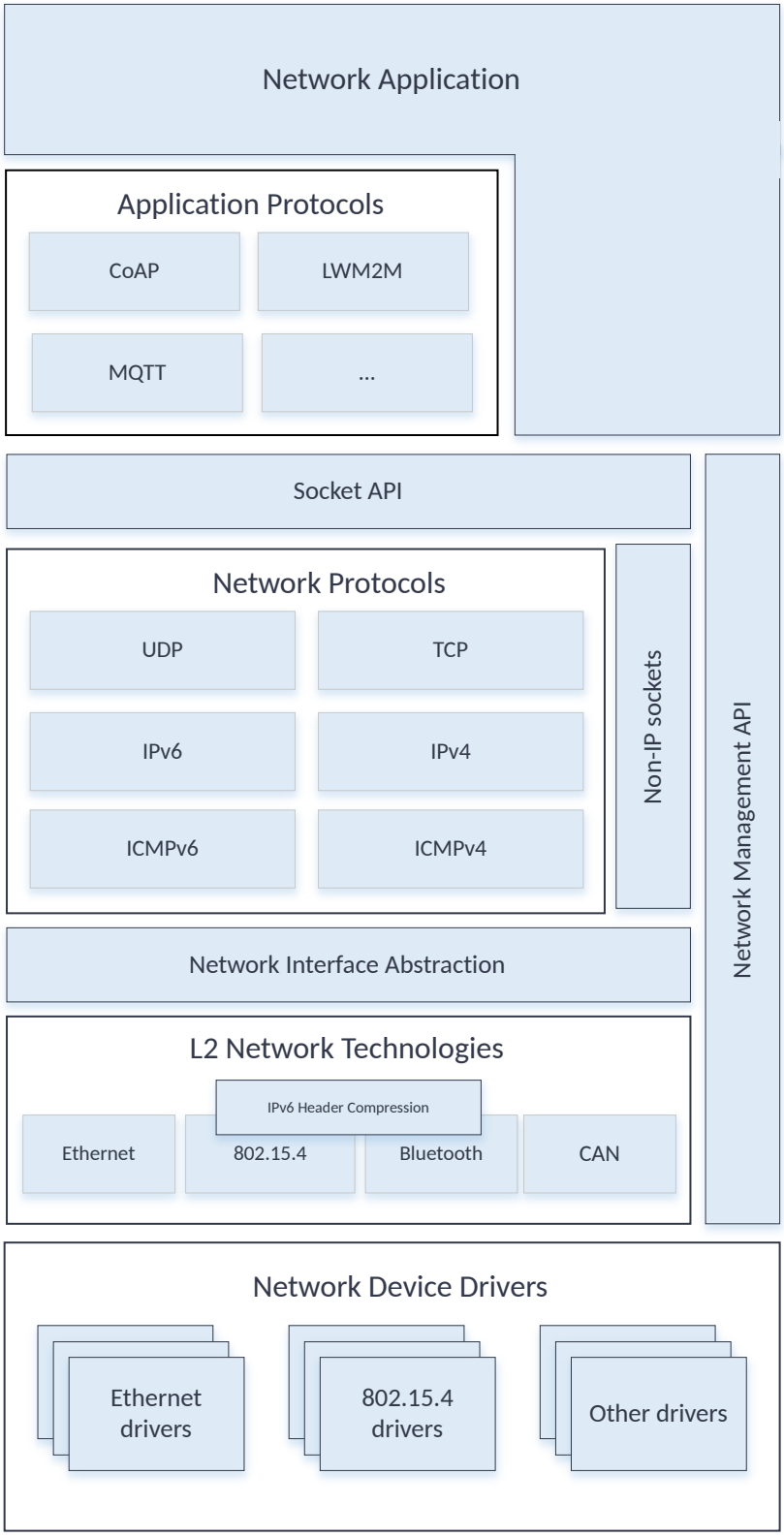


Fig. 12: Network stack overview

- **Network Application.** The network application can either use the provided application-level protocol libraries or access the *BSD socket API* directly to create a network connection, send or receive data, and close a connection. The application can also use the *network management API* to configure the network and set related parameters such as network link options, starting a scan (when applicable), listen network configuration events, etc. The *network interface API* can be used to set IP address to a network interface, taking the network interface down, etc.
- **Network Protocols.** This provides implementations for various protocols such as
  - Application-level network protocols like CoAP, LWM2M, and MQTT. See *application protocols chapter* for information about them.
  - Core network protocols like IPv6, IPv4, UDP, TCP, ICMPv4, and ICMPv6. You access these protocols by using the *BSD socket API*.
- **Network Interface Abstraction.** This provides functionality that is common in all the network interfaces, such as setting network interface down, etc. There can be multiple network interfaces in the system. See *network interface overview* for more details.
- **L2 Network Technologies.** This provides a common API for sending and receiving data to and from an actual network device. See *L2 overview* for more details. These network technologies include *Ethernet*, *IEEE 802.15.4*, *Bluetooth*, *CANBUS*, etc. Some of these technologies support IPv6 header compression (6Lo), see RFC 6282 for details. For example ARP for IPv4 is done by the *Ethernet component*.
- **Network Device Drivers.** The actual low-level device drivers handle the physical sending or receiving of network packets.

Network data flow

An application typically consists of one or more *threads* that execute the application logic. When using the *BSD socket API*, the following things will happen.

Data receiving (RX)

1. A network data packet is received by a device driver.
2. The device driver allocates enough network buffers to store the received data. The network packet is placed in the proper RX queue (implemented by *k_fifo*). By default there is only one receive queue in the system, but it is possible to have up to 8 receive queues. These queues will process incoming packets with different priority. See *Traffic Classification* for more details. The receive queues also act as a way to separate the data processing pipeline (bottom-half) as the device driver is running in an interrupt context and it must do its processing as fast as possible.
3. The network packet is then passed to the correct L2 driver. The L2 driver can check if the packet is proper and modify it if needed, e.g. strip L2 header and frame check sequence, etc.
4. The packet is processed by a network interface. The network statistics are collected if enabled by `CONFIG_NET_STATISTICS`.
5. The packet is then passed to L3 processing. If the packet is IP based, then the L3 layer checks if the packet is a proper IPv6 or IPv4 packet.
6. A socket handler then finds an active socket to which the network packet belongs and puts it in a queue for that socket, in order to separate the networking code from the application. Typically the application is run in userspace context and the network stack is run in kernel context.
7. The application will then receive the data and can process it as needed. The application should have used the *BSD socket API* to create a socket that will receive the data.

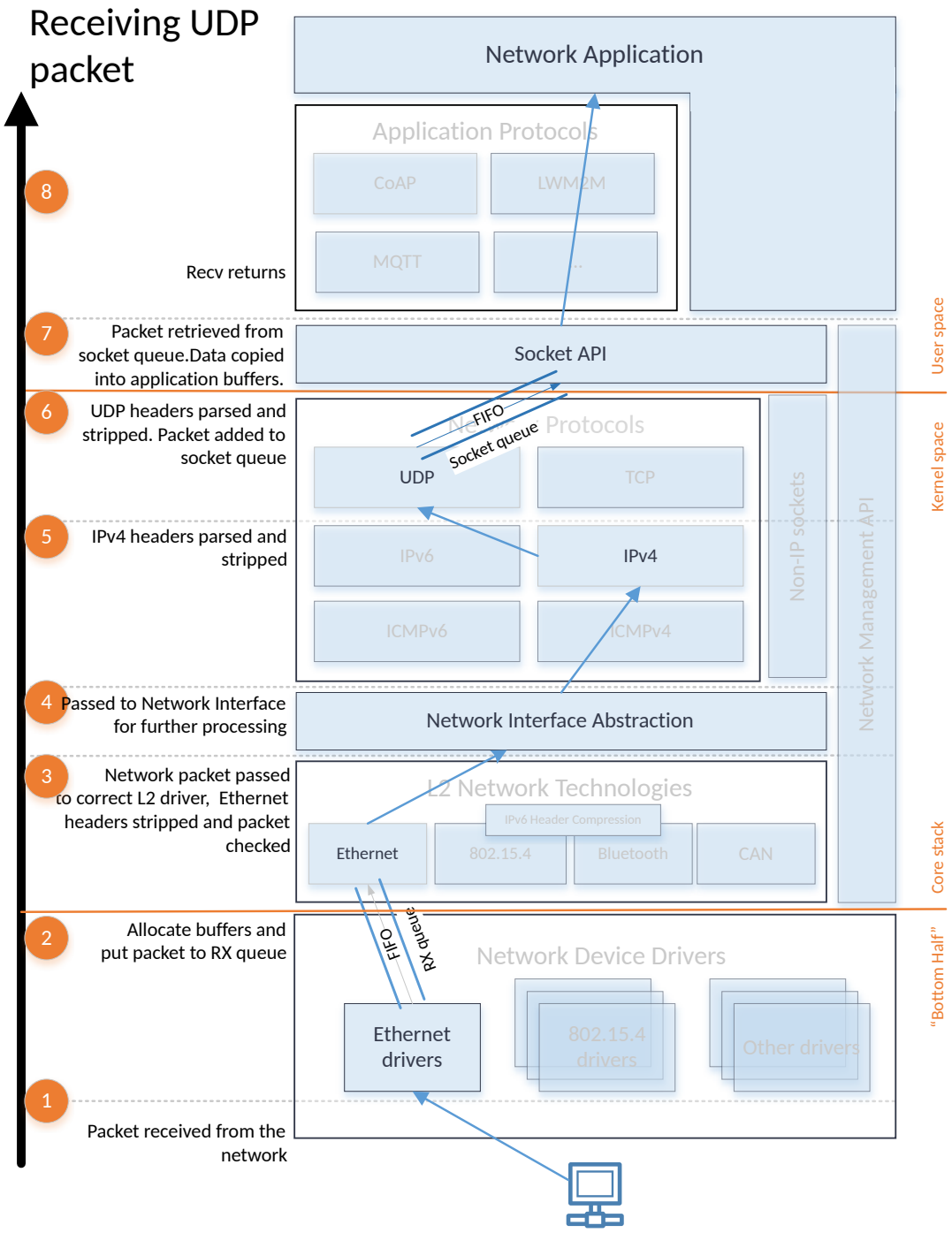


Fig. 13: Network RX data flow

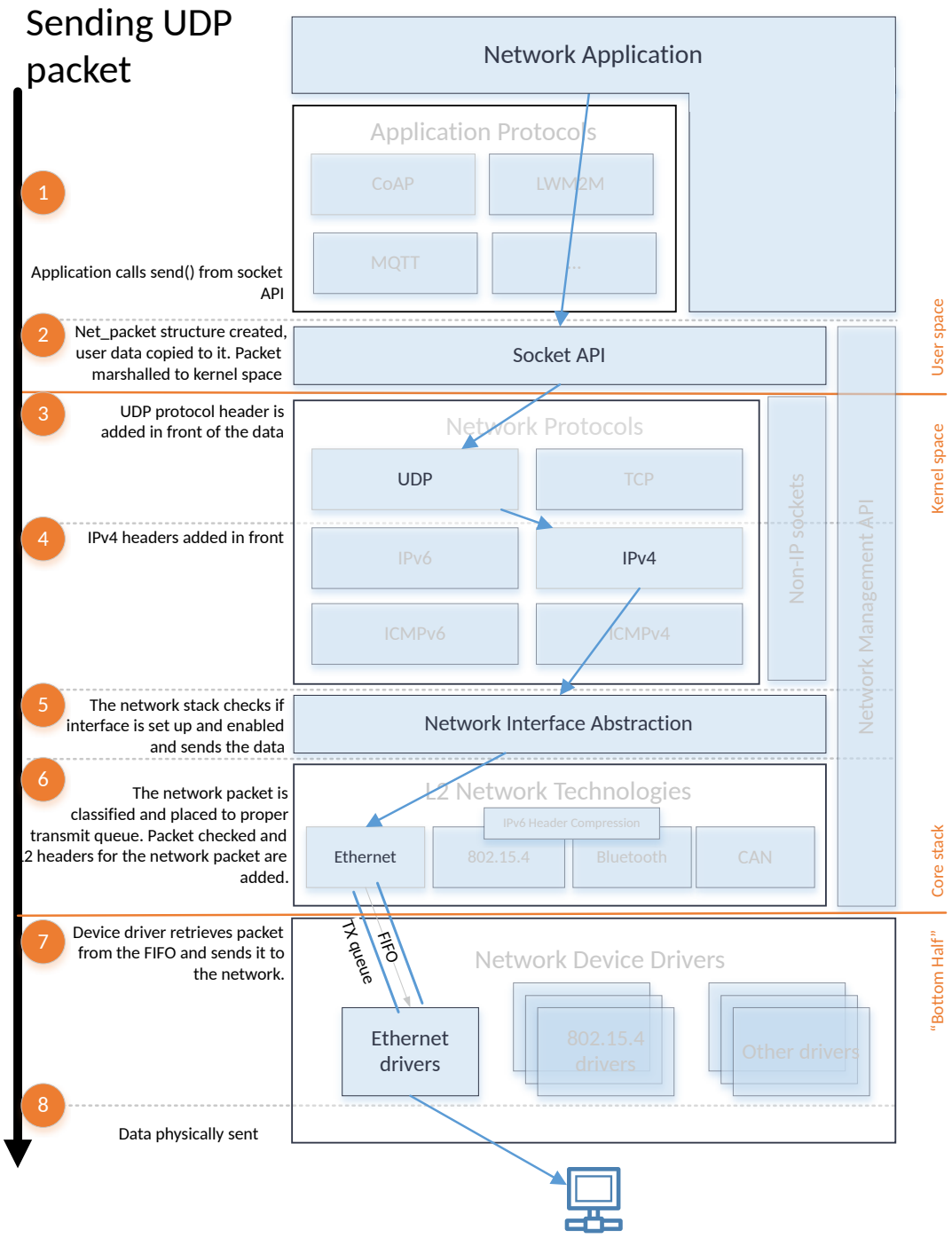


Fig. 14: Network TX data flow

Data sending (TX)

1. The application should use the *BSD socket API* when sending the data.
2. The application data is prepared for sending to kernel space and then copied to internal net_buf structures.
3. Depending on the socket type, a protocol header is added in front of the data. For example, if the socket is a UDP socket, then a UDP header is constructed and placed in front of the data.
4. An IP header is added to the network packet for a UDP or TCP packet.
5. The network stack will check that the network interface is properly set for the network packet, and also will make sure that the network interface is enabled before the data is queued to be sent.
6. The network packet is then classified and placed to the proper transmit queue (implemented by *k_fifo*). By default there is only one transmit queue in the system, but it is possible to have up to 8 transmit queues. These queues will process the sent packets with different priority. See *Traffic Classification* for more details. After the transmit packet classification, the packet is checked by the correct L2 layer module. The L2 module will do additional checks for the data and it will also create any L2 headers for the network packet. If everything is ok, the data is given to the network device driver to be sent out.
7. The device driver will send the packet to the network.

Note that in both the TX and RX data paths, the queues (*k_fifo's*) form separation points where data is passed from one *thread* to another. These *threads* might run in different contexts (*kernel* vs. *userspace*) and with different *priorities*.

Network packet processing statistics

See information about network processing statistics *here*.

6.3.3 Network Configuration Guide

- *Network Buffer Configuration Options*
- *Connection Options*
- *Socket Options*
- *TLS Options*
- *IPv4/6 Options*
- *TCP Options*
- *Traffic Class Options*
- *Stack Size Options*

This document describes how various network configuration options can be set according to available resources in the system.

Network Buffer Configuration Options

The network buffer configuration options control how much data we are able to either send or receive at the same time.

CONFIG_NET_PKT_RX_COUNT
: Maximum amount of network packets we can receive at the same time.

CONFIG_NET_PKT_TX_COUNT
: Maximum amount of network packet sends pending at the same time.

CONFIG_NET_BUF_RX_COUNT
: How many network buffers are allocated for receiving data. Each net_buf contains a small header and either a fixed or variable length data buffer. The CONFIG_NET_BUF_DATA_SIZE is used when CONFIG_NET_BUF_FIXED_DATA_SIZE is set. This is the default setting. The default size of the buffer is 128 bytes.

  The CONFIG_NET_BUF_VARIABLE_DATA_SIZE is an experimental setting. There each net_buf data portion is allocated from a memory pool and can be the amount of data we have received from the network. When data is received from the network, it is placed into net_buf data portion. Depending on device resources and desired network usage, user can tweak the size of the fixed buffer by setting CONFIG_NET_BUF_DATA_SIZE, and the size of the data pool size by setting CONFIG_NET_PKT_BUF_RX_DATA_POOL_SIZE and CONFIG_NET_PKT_BUF_TX_DATA_POOL_SIZE if variable size buffers are used.

  When using the fixed size data buffers, the memory consumption of network buffers can be tweaked by selecting the size of the data part according to what kind of network data we are receiving. If one sets the data size to 256, but only receives packets that are 32 bytes long, then we are "wasting" 224 bytes for each packet because we cannot utilize the remaining data. One should not set the data size too low because there is some overhead involved for each net_buf. For these reasons the default network buffer size is set to 128 bytes.

  The variable size data buffer feature is marked as experimental as it has not received as much testing as the fixed size buffers. Using variable size data buffers tries to improve memory utilization by allocating minimum amount of data we need for the network data. The extra cost here is the amount of time that is needed when dynamically allocating the buffer from the memory pool.

  For example, in Ethernet the maximum transmission unit (MTU) size is 1500 bytes. If one wants to receive two full frames, then the net_pkt RX count should be set to 2, and net_buf RX count to (1500 / 128) * 2 which is 24. If TCP is being used, then these values need to be higher because we can queue the packets internally before delivering to the application.

CONFIG_NET_BUF_TX_COUNT
: How many network buffers are allocated for sending data. This is similar setting as the receive buffer count but for sending.

Connection Options

CONFIG_NET_MAX_CONN
: This option tells how many network connection endpoints are supported. For example each TCP connection requires one connection endpoint. Similarly each listening UDP connection requires one connection endpoint. Also various system services like DHCP and DNS need connection endpoints to work. The network shell command **net conn** can be used at runtime to see the network connection information.

CONFIG_NET_MAX_CONTEXTS
: Number of network contexts to allocate. Each network context describes a network 5-tuple that is used when listening or sending network traffic. Each BSD socket in the system uses one network context.

Socket Options

CONFIG_NET_SOCKETS_POLL_MAX
: Maximum number of supported poll() entries. One needs to select proper value here de-

pending on how many BSD sockets are polled in the system.

CONFIG_ZVFS_OPEN_MAX
: Maximum number of open file descriptors, this includes files, sockets, special devices, etc. One needs to select proper value here depending on how many BSD sockets are created in the system.

CONFIG_NET_SOCKETPAIR_BUFFER_SIZE
: This option is used by socketpair() function. It sets the size of the internal intermediate buffer, in bytes. This sets the limit how large messages can be passed between two socketpair endpoints.

TLS Options

CONFIG_NET_SOCKETS_TLS_MAX_CONTEXTS
: Maximum number of TLS/DTLS contexts. Each TLS/DTLS connection needs one context.

CONFIG_NET_SOCKETS_TLS_MAX_CREDENTIALS
: This variable sets maximum number of TLS/DTLS credentials that can be used with a specific socket.

CONFIG_NET_SOCKETS_TLS_MAX_CIPHERSUITES
: Maximum number of TLS/DTLS ciphersuites per socket. This variable sets maximum number of TLS/DTLS ciphersuites that can be used with specific socket, if set explicitly by socket option. By default, all ciphersuites that are available in the system are available to the socket.

CONFIG_NET_SOCKETS_TLS_MAX_APP_PROTOCOLS
: Maximum number of supported application layer protocols. This variable sets maximum number of supported application layer protocols over TLS/DTLS that can be set explicitly by a socket option. By default, no supported application layer protocol is set.

CONFIG_NET_SOCKETS_TLS_MAX_CLIENT_SESSION_COUNT
: This variable specifies maximum number of stored TLS/DTLS sessions, used for TLS/DTLS session resumption.

CONFIG_TLS_MAX_CREDENTIALS_NUMBER
: Maximum number of TLS credentials that can be registered. Make sure that this value is high enough so that all the certificates can be loaded to the store.

IPv4/6 Options

CONFIG_NET_IF_MAX_IPV4_COUNT
: Maximum number of IPv4 network interfaces in the system. This tells how many network interfaces there will be in the system that will have IPv4 enabled. For example if you have two network interfaces, but only one of them can use IPv4 addresses, then this value can be set to 1. If both network interface could use IPv4, then the setting should be set to 2.

CONFIG_NET_IF_MAX_IPV6_COUNT
: Maximum number of IPv6 network interfaces in the system. This is similar setting as the IPv4 count option but for IPv6.

TCP Options

CONFIG_NET_TCP_TIME_WAIT_DELAY
: How long to wait in TCP *TIME_WAIT* state (in milliseconds). To avoid a (low-probability) issue when delayed packets from previous connection get delivered to next connection

reusing the same local/remote ports, RFC 793 (TCP) suggests to keep an old, closed connection in a special *TIME_WAIT* state for the duration of 2*MSL (Maximum Segment Lifetime). The RFC suggests to use MSL of 2 minutes, but notes

This is an engineering choice, and may be changed if experience indicates it is desirable to do so.

For low-resource systems, having large MSL may lead to quick resource exhaustion (and related DoS attacks). At the same time, the issue of packet misdelivery is largely alleviated in the modern TCP stacks by using random, non-repeating port numbers and initial sequence numbers. Due to this, Zephyr uses much lower value of 1500ms by default. Value of 0 disables *TIME_WAIT* state completely.

`CONFIG_NET_TCP_RETRY_COUNT`
Maximum number of TCP segment retransmissions. The following formula can be used to determine the time (in ms) that a segment will be buffered awaiting retransmission:

$$\sum_{n=0}^{\texttt{NET\_TCP\_RETRY\_COUNT}} \left(1 \ll n\right) \times \texttt{NET\_TCP\_INIT\_RETRANSMISSION\_TIMEOUT}$$

With the default value of 9, the IP stack will try to retransmit for up to 1:42 minutes. This is as close as possible to the minimum value recommended by RFC 1122 (1:40 minutes). Only 5 bits are dedicated for the retransmission count, so accepted values are in the 0-31 range. It's highly recommended to not go below 9, though.

Should a retransmission timeout occur, the receive callback is called with `-ETIMEDOUT` error code and the context is dereferenced.

`CONFIG_NET_TCP_MAX_SEND_WINDOW_SIZE`
Maximum sending window size to use. This value affects how the TCP selects the maximum sending window size. The default value 0 lets the TCP stack select the value according to amount of network buffers configured in the system. Note that if there are multiple active TCP connections in the system, then this value might require finetuning (lowering), otherwise multiple TCP connections could easily exhaust net_buf pool for the queued TX data.

`CONFIG_NET_TCP_MAX_RECV_WINDOW_SIZE`
Maximum receive window size to use. This value defines the maximum TCP receive window size. Increasing this value can improve connection throughput, but requires more receive buffers available in the system for efficient operation. The default value 0 lets the TCP stack select the value according to amount of network buffers configured in the system.

`CONFIG_NET_TCP_RECV_QUEUE_TIMEOUT`
How long to queue received data (in ms). If we receive out-of-order TCP data, we queue it. This value tells how long the data is kept before it is discarded if we have not been able to pass the data to the application. If set to 0, then receive queueing is not enabled. The value is in milliseconds.

Note that we only queue data sequentially in current version i.e., there should be no holes in the queue. For example, if we receive SEQs 5,4,3,6 and are waiting SEQ 2, the data in segments 3,4,5,6 is queued (in this order), and then given to application when we receive SEQ 2. But if we receive SEQs 5,4,3,7 then the SEQ 7 is discarded because the list would not be sequential as number 6 is be missing.

Traffic Class Options

It is possible to configure multiple traffic classes (queues) when receiving or sending network data. Each traffic class queue is implemented as a thread with different priority. This means that higher priority network packet can be placed to a higher priority network queue in order to send or receive it faster or slower. Because of thread scheduling latencies, in practice the fastest way to send a packet out, is to directly send the packet without using a dedicated traffic class

thread. This is why by default the CONFIG_NET_TC_TX_COUNT option is set to 0 if userspace is not enabled. If userspace is enabled, then the minimum TX traffic class count is 1. Reason for this is that the userspace application does not have enough permissions to deliver the message directly.

In receiving side, it is recommended to have at least one receiving traffic class queue. Reason is that typically the network device driver is running in IRQ context when it receives the packet, in which case it should not try to deliver the network packet directly to the upper layers, but to place the packet to the traffic class queue. If the network device driver is not running in IRQ context when it gets the packet, then the RX traffic class option CONFIG_NET_TC_RX_COUNT could be set to 0.

Stack Size Options

There several network specific threads in a network enabled system. Some of the threads might depend on a configure option which can be used to enable or disable a feature. Each thread stack size is optimized to allow normal network operations.

The network management API is using a dedicated thread by default. The thread is responsible to deliver network management events to the event listeners that are setup in the system if the CONFIG_NET_MGMT and CONFIG_NET_MGMT_EVENT options are enabled. If the options are enabled, the user is able to register a callback function that the net_mgmt thread is calling for each network management event. By default the net_mgmt event thread stack size is rather small. The idea is that the callback function does minimal things so that new events can be delivered to listeners as fast as possible and they are not lost. The net_mgmt event thread stack size is controlled by CONFIG_NET_MGMT_EVENT_QUEUE_SIZE option. It is recommended to not do any blocking operations in the callback function.

The network thread stack utilization can be monitored from kernel shell by the **kernel threads** command.

6.3.4 Networking with the host system

Networking with native_sim board

- *Using virtual/TAP Ethernet driver*
  - *Prerequisites*
  - *Basic Setup*
- *Using offloaded sockets*
  - *Step 1 - Start app in native_sim board*
  - *Step 2 - run echo-client from net-tools*

Using virtual/TAP Ethernet driver This paragraph describes how to set up a virtual network between a (Linux) host and a Zephyr application running in a native_sim board.

In this example, the sockets-echo-server sample application from the Zephyr source distribution is run in native_sim board. The Zephyr native_sim board instance is connected to a Linux host using a tuntap device which is modeled in Linux as an Ethernet network interface.

Prerequisites On the Linux Host, fetch the Zephyr `net-tools` project, which is located in a separate Git repository:

```
git clone https://github.com/zephyrproject-rtos/net-tools
```

Basic Setup For the steps below, you will need three terminal windows:

- Terminal #1 is terminal window with net-tools being the current directory (`cd net-tools`)
- Terminal #2 is your usual Zephyr development terminal, with the Zephyr environment initialized.
- Terminal #3 is the console to the running Zephyr native_sim instance (optional).

Step 1 - Create Ethernet interface Before starting native_sim with network emulation, a network interface should be created.

In terminal #1, type:

```
./net-setup.sh
```

You can tweak the behavior of the net-setup.sh script. See various options by running `net-setup.sh` like this:

```
./net-setup.sh --help
```

Step 2 - Start app in native_sim board Build and start the `echo_server` sample application.

In terminal #2, type:

```
west build -b native_sim samples/net/sockets/echo_server
west build -t run
```

Step 3 - Connect to console (optional) The console window should be launched automatically when the Zephyr instance is started but if it does not show up, you can manually connect to the console. The native_sim board will print a string like this when it starts:

```
UART connected to pseudotty: /dev/pts/5
```

You can manually connect to it like this:

```
screen /dev/pts/5
```

Using offloaded sockets The main advantage over *Using virtual/TAP Ethernet driver* is not needing to setup a virtual network interface on the host machine. This means that no leveraged (root) privileges are needed.

Step 1 - Start app in native_sim board Build and start the `echo_server` sample application:

```
west build -b native_sim samples/net/sockets/echo_server -- -DEXTRA_CONF_FILE=overlay-nsos.
↪conf
west build -t run
```

Step 2 - run echo-client from net-tools On the Linux Host, fetch the Zephyr `net-tools` project, which is located in a separate Git repository:

```
git clone https://github.com/zephyrproject-rtos/net-tools
```

> **Note**
>
> Native Simulator with the offloaded sockets network driver is using the same network interface/namespace as any other (Linux) application that uses BSD sockets API. This means that sockets-echo-server and `echo-client` applications will communicate over localhost/loopback interface (address `127.0.0.1`).

To run UDP test, type:

```
./echo-client 127.0.0.1
```

For TCP test, type:

```
./echo-client -t 127.0.0.1
```

Networking with QEMU Ethernet

- *Prerequisites*
- *Basic Setup*
  - *Step 1 - Create Ethernet interface*
  - *Step 2 - Start app in QEMU board*

This page describes how to set up a virtual network between a (Linux) host and a Zephyr application running in QEMU.

In this example, the sockets-echo-server sample application from the Zephyr source distribution is run in QEMU. The Zephyr instance is connected to a Linux host using a tuntap device which is modeled in Linux as an Ethernet network interface.

Prerequisites On the Linux Host, fetch the Zephyr `net-tools` project, which is located in a separate Git repository:

```
git clone https://github.com/zephyrproject-rtos/net-tools
```

Basic Setup For the steps below, you will need two terminal windows:

- Terminal #1 is terminal window with net-tools being the current directory (`cd net-tools`)
- Terminal #2 is your usual Zephyr development terminal, with the Zephyr environment initialized.

When configuring the Zephyr instance, you must select the correct Ethernet driver for QEMU connectivity:

- For `qemu_x86`, select `Intel(R) PRO/1000 Gigabit Ethernet driver` Ethernet driver. Driver is called `e1000` in Zephyr source tree.

- For `qemu_cortex_m3`, select `TI Stellaris MCU family ethernet driver` Ethernet driver. Driver is called `stellaris` in Zephyr source tree.
- For `mps2_an385`, select `SMSC911x/9220 Ethernet driver` Ethernet driver. Driver is called `smsc911x` in Zephyr source tree.
- For `qemu_cortex_a53`, `Intel(R) PRO/1000 Gigabit Ethernet driver` Ethernet driver is selected by default.

Step 1 - Create Ethernet interface Before starting QEMU with network connectivity, a network interface should be created in the host system.

In terminal #1, type:

```
./net-setup.sh
```

You can tweak the behavior of the `net-setup.sh` script. See various options by running `net-setup.sh` like this:

```
./net-setup.sh --help
```

Step 2 - Start app in QEMU board Build and start the sockets-echo-server sample application. In this example, the qemu_x86 board is used.

In terminal #2, type:

```
west build -b qemu_x86 samples/net/sockets/echo_server -- -DEXTRA_CONF_FILE=overlay-e1000.
↪conf
west build -t run
```

Exit QEMU by pressing `CTRL+A x`.

Networking with QEMU

- *Prerequisites*
- *Basic Setup*
  - *Step 1 - Create helper socket*
  - *Step 2 - Start TAP device routing daemon*
  - *Step 3 - Start app in QEMU*
  - *Step 4 - Run apps on host*
  - *Step 5 - Stop supporting daemons*
- *Setting up Zephyr and NAT/masquerading on host to access Internet*
- *Network connection between two QEMU VMs*
  - *Terminal #1:*
  - *Terminal #2:*
- *Running multiple QEMU VMs of the same sample*
  - *Terminal #1:*
  - *Terminal #2:*

This page describes how to set up a virtual network between a (Linux) host and a Zephyr application running in a QEMU virtual machine (built for Zephyr targets such as qemu_x86 and qemu_cortex_m3). Some virtual ARM boards (such as qemu_cortex_a53) only support a single UART, in this case QEMU Ethernet is preferred, see *Networking with QEMU Ethernet* for details.

In this example, the sockets-echo-server sample application from the Zephyr source distribution is run in QEMU. The QEMU instance is connected to a Linux host using a serial port, and SLIP is used to transfer data between the Zephyr application and Linux (over a chain of virtual connections).

Prerequisites On the Linux Host, fetch the Zephyr `net-tools` project, which is located in a separate Git repository:

```
git clone https://github.com/zephyrproject-rtos/net-tools
cd net-tools
make
```

Note

If you get an error about AX_CHECK_COMPILE_FLAG, install package `autoconf-archive` package on Debian/Ubuntu.

Basic Setup For the steps below, you will need at least 4 terminal windows:

- Terminal #1 is your usual Zephyr development terminal, with the Zephyr environment initialized.
- Terminals #2, #3, and #4 are terminal windows with net-tools being the current directory (`cd net-tools`)

Step 1 - Create helper socket Before starting QEMU with network emulation, a Unix socket for the emulation should be created.

In terminal #2, type:

```
./loop-socat.sh
```

Step 2 - Start TAP device routing daemon In terminal #3, type:

```
sudo ./loop-slip-tap.sh
```

For applications requiring DNS, you may need to restart the host's DNS server at this point, as described in *Setting up Zephyr and NAT/masquerading on host to access Internet*.

Step 3 - Start app in QEMU Build and start the `echo_server` sample application.

In terminal #1, type:

```
west build -b qemu_x86 samples/net/sockets/echo_server
west build -t run
```

If you see an error from QEMU about unix:/tmp/slip.sock, it means you missed Step 1 above.

Step 4 - Run apps on host Now in terminal #4, you can run various tools to communicate with the application running in QEMU.

You can start with pings:

```
ping 192.0.2.1
ping6 2001:db8::1
```

You can use the netcat ("nc") utility, connecting using UDP:

```
echo foobar | nc -6 -u 2001:db8::1 4242
foobar
```

```
echo foobar | nc -u 192.0.2.1 4242
foobar
```

If echo_server is compiled with TCP support (now enabled by default for the echo_server sample, CONFIG_NET_TCP=y):

```
echo foobar | nc -6 -q2 2001:db8::1 4242
foobar
```

Note

Use Ctrl+C to exit.

You can also use the telnet command to achieve the above.

Step 5 - Stop supporting daemons When you are finished with network testing using QEMU, you should stop any daemons or helpers started in the initial steps, to avoid possible networking or routing problems such as address conflicts in local network interfaces. For example, stop them if you switch from testing networking with QEMU to using real hardware, or to return your host laptop to normal Wi-Fi use.

To stop the daemons, press Ctrl+C in the corresponding terminal windows (you need to stop both `loop-slip-tap.sh` and `loop-socat.sh`).

Exit QEMU by pressing `CTRL+A x`.

Setting up Zephyr and NAT/masquerading on host to access Internet To access the internet from a Zephyr application, some additional setup on the host may be required. This setup is common for both application running in QEMU and on real hardware, assuming that a development board is connected to the development host. If a board is connected to a dedicated router, it should not be needed.

To access the internet from a Zephyr application using IPv4, a gateway should be set via DHCP or configured manually. For applications using the "Settings" facility (with the config option `CONFIG_NET_CONFIG_SETTINGS` enabled), set the `CONFIG_NET_CONFIG_MY_IPV4_GW` option to the IP address of the gateway. For apps not using the "Settings" facility, set up the gateway by calling the *`net_if_ipv4_set_gw()`* at runtime. For example: `CONFIG_NET_CONFIG_MY_IPV4_GW="192.0.2.2"`

To access the internet from a custom application running in QEMU, NAT (masquerading) should be set up for QEMU's source address. Assuming `192.0.2.1` is used and the Zephyr network interface is `zeth`, the following command should be run as root:

```
iptables -t nat -A POSTROUTING -j MASQUERADE -s 192.0.2.1/24
iptables -I FORWARD 1 -i zeth -j ACCEPT
iptables -I FORWARD 1 -o zeth -m state --state RELATED,ESTABLISHED -j ACCEPT
```

Additionally, IPv4 forwarding should be enabled on the host, and you may need to check that other firewall (iptables) rules don't interfere with masquerading. To enable IPv4 forwarding the following command should be run as root:

```
sysctl -w net.ipv4.ip_forward=1
```

Some applications may also require a DNS server. A number of Zephyr-provided samples assume by default that the DNS server is available on the host (IP `192.0.2.2`), which, in modern Linux distributions, usually runs at least a DNS proxy. When running with QEMU, it may be required to restart the host's DNS, so it can serve requests on the newly created TAP interface. For example, on Debian-based systems:

```
service dnsmasq restart
```

An alternative to relying on the host's DNS server is to use one in the network. For example, `8.8.8.8` is a publicly available DNS server. You can configure it using `CONFIG_DNS_SERVER1` option.

Network connection between two QEMU VMs Unlike the VM-to-Host setup described above, VM-to-VM setup is automatic. For sample applications that support this mode (such as the echo_server and echo_client samples), you will need two terminal windows, set up for Zephyr development.

Terminal #1:

```
west build -b qemu_x86 samples/net/sockets/echo_server
```

This will start QEMU, waiting for a connection from a client QEMU.

Terminal #2:

```
west build -b qemu_x86 samples/net/sockets/echo_client
```

This will start a second QEMU instance, where you should see logging of data sent and received in both.

Running multiple QEMU VMs of the same sample If you find yourself wanting to run multiple instances of the same Zephyr sample application, which do not need to talk to each other, use the `QEMU_INSTANCE` argument.

Start `socat` and `tunslip6` manually (instead of using the `loop-xxx.sh` scripts) for as many instances as you want. Use the following as a guide, replacing MAIN or OTHER.

Terminal #1:

```
socat PTY,link=/tmp/slip.devMAIN UNIX-LISTEN:/tmp/slip.sockMAIN
$ZEPHYR_BASE/../net-tools/tunslip6 -t tapMAIN -T -s /tmp/slip.devMAIN \
     2001:db8::1/64
# Now run Zephyr
make -Cbuild run QEMU_INSTANCE=MAIN
```

Terminal #2:

```
socat PTY,link=/tmp/slip.devOTHER UNIX-LISTEN:/tmp/slip.sockOTHER
$ZEPHYR_BASE/../net-tools/tunslip6 -t tapOTHER -T -s /tmp/slip.devOTHER \
     2001:db8::1/64
make -Cbuild run QEMU_INSTANCE=OTHER
```

USB Device Networking

- *Basic Setup*
  - *Choosing IP addresses*
  - *Setting IPv4 address and routing*
  - *Setting IPv6 address and routing*
- *Testing connection*

This page describes how to set up networking between a Linux host and a Zephyr application running on USB supported devices.

The board is connected to Linux host using USB cable and provides an Ethernet interface to the host. The sockets-echo-server application from the Zephyr source distribution is run on supported board. The board is connected to a Linux host using a USB cable providing an Ethernet interface to the host.

Basic Setup To communicate with the Zephyr application over a newly created Ethernet interface, we need to assign IP addresses and set up a routing table for the Linux host. After plugging a USB cable from the board to the Linux host, the `cdc_ether` driver registers a new Ethernet device with a provided MAC address.

You can check that network device is created and MAC address assigned by running dmesg from the Linux host.

```
cdc_ether 1-2.7:1.0 eth0: register 'cdc_ether' at usb-0000:00:01.2-2.7, CDC Ethernet Device,
↪ 00:00:5e:00:53:01
```

We need to set it up and assign IP addresses as explained in the following section.

Choosing IP addresses To establish network connection to the board we need to choose IP address for the interface on the Linux host.

It make sense to choose addresses in the same subnet we have in Zephyr application. IP addresses usually set in the project configuration files and may be checked also from the shell with following commands. Connect a serial console program (such as puTTY) to the board, and enter this command to the Zephyr shell:

```
shell> net iface

Interface 0xa800e580 (Ethernet)
===============================
Link addr : 00:00:5E:00:53:00
MTU       : 1500
IPv6 unicast addresses (max 2):
        fe80::200:5eff:fe00:5300 autoconf preferred infinite
        2001:db8::1 manual preferred infinite
...
IPv4 unicast addresses (max 1):
        192.0.2.1 manual preferred infinite
```

This command shows that one IPv4 address and two IPv6 addresses have been assigned to the board. We can use either IPv4 or IPv6 for network connection depending on the board network configuration.

Next step is to assign IP addresses to the new Linux host interface, in the following steps `enx00005e005301` is the name of the interface on my Linux system.

Setting IPv4 address and routing

```
# ip address add dev enx00005e005301 192.0.2.2
# ip link set enx00005e005301 up
# ip route add 192.0.2.0/24 dev enx00005e005301
```

Setting IPv6 address and routing

```
# ip address add dev enx00005e005301 2001:db8::2
# ip link set enx00005e005301 up
# ip -6 route add 2001:db8::/64 dev enx00005e005301
```

Testing connection From the host we can test the connection by pinging Zephyr IP address of the board with:

```
$ ping 192.0.2.1
PING 192.0.2.1 (192.0.2.1) 56(84) bytes of data.
64 bytes from 192.0.2.1: icmp_seq=1 ttl=64 time=2.30 ms
64 bytes from 192.0.2.1: icmp_seq=2 ttl=64 time=1.43 ms
64 bytes from 192.0.2.1: icmp_seq=3 ttl=64 time=2.45 ms
...
```

Networking with QEMU User

- *Introduction*
- *Using SLIRP with Zephyr*
- *Limitations*

This page is intended to serve as a starting point for anyone interested in using QEMU SLIRP with Zephyr.

Introduction SLIRP is a network backend which provides the complete TCP/IP stack within QEMU and uses that stack to implement a virtual NAT'd network. As there are no dependencies on the host, SLIRP is simple to setup.

By default, QEMU uses the `10.0.2.X/24` network and runs a gateway at `10.0.2.2`. All traffic intended for the host network has to travel through this gateway, which will filter out packets based on the QEMU command line parameters. This gateway also functions as a DHCP server for all GOS, allowing them to be automatically assigned with an IP address starting from `10.0.2.15`.

More details about User Networking can be obtained from here: https://wiki.qemu.org/Documentation/Networking#User_Networking_.28SLIRP.29

Using SLIRP with Zephyr In order to use SLIRP with Zephyr, the user has to set the Kconfig option to enable User Networking.

```
CONFIG_NET_QEMU_USER=y
```

Once this configuration option is enabled, all QEMU launches will use SLIRP. In the default configuration, Zephyr only enables User Networking, and does not pass any arguments to it. This means that the Guest will only be able to communicate to the QEMU gateway, and any data intended for the host machine will be dropped by QEMU.

In general, QEMU User Networking can take in a lot of arguments including,

- Information about host/guest port forwarding. This must be provided to create a communication channel between the guest and host.
- Information about network to use. This may be valuable if the user does not want to use the default `10.0.2.X` network.
- Tell QEMU to start DHCP server at user-defined IP address.
- ID and other information.

As this information varies with every use case, it is difficult to come up with good defaults that work for all. Therefore, Zephyr Implementation offloads this to the user, and expects that they will provide arguments based on requirements. For this, there is a Kconfig string which can be populated by the user.

```
CONFIG_NET_QEMU_USER_EXTRA_ARGS="net=192.168.0.0/24,hostfwd=tcp::8080-:8080"
```

This option is appended as-is to the QEMU command line. Therefore, any problems with this command line will be reported by QEMU only. Here's what this particular example will do,

- Make QEMU use the `192.168.0.0/24` network instead of the default.
- Enable forwarding of any TCP data received from port 8080 of host to port 8080 of guest, and vice versa.

Limitations If the user does not have any specific networking requirements other than the ability to access a web page from the guest, user networking (slirp) is a good choice. However, it has several limitations

- There is a lot of overhead so the performance is poor.
- The guest is not directly accessible from the host or the external network.
- In general, ICMP traffic does not work (so you cannot use ping within a guest).
- As port mappings need to be defined before launching qemu, clients which use dynamically generated ports cannot communicate with external network.
- There is a bug in the SLIRP implementation which filters out all IPv6 packets from the guest. See https://bugs.launchpad.net/qemu/+bug/1724590 for details. Therefore, IPv6 will not work with User Networking.

Networking with multiple Zephyr instances

- *Prerequisites*
- *Basic Setup*
  - *Step 1 - Create configuration files*
  - *Step 2 - Create Ethernet interfaces*
  - *Step 3 - Setup network bridging*
  - *Step 4 - Start Zephyr instances*

This page describes how to set up a virtual network between multiple Zephyr instances. The Zephyr instances could be running inside QEMU or could be native_sim board processes. The Linux host can be used to route network traffic between these systems.

Prerequisites On the Linux Host, fetch the Zephyr `net-tools` project, which is located in a separate Git repository:

```
git clone https://github.com/zephyrproject-rtos/net-tools
```

Basic Setup For the steps below, you will need five terminal windows:

- Terminal #1 and #2 are terminal windows with net-tools being the current directory (`cd net-tools`)
- Terminal #3, where you setup bridging in Linux host
- Terminal #4 and #5 are your usual Zephyr development terminal, with the Zephyr environment initialized.

As there are multiple ways to setup the Zephyr network, the example below uses `qemu_x86` board with `e1000` Ethernet controller and native_sim board to simplify the setup instructions. You can use other QEMU boards and drivers if needed, see *Networking with QEMU Ethernet* for details. You can also use two or more native_sim board Zephyr instances and connect them together.

Step 1 - Create configuration files Before starting QEMU with network connectivity, a network interfaces for each Zephyr instance should be created in the host system. The default setup for creating network interface cannot be used here as that is for connecting one Zephyr instance to Linux host.

For Zephyr instance #1, create file called `zephyr1.conf` to `net-tools` project, or to some other suitable directory.

```
# Configuration file for setting IP addresses for a network interface.
INTERFACE="$1"
HWADDR="00:00:5e:00:53:11"
IPV6_ADDR_1="2001:db8:100::2"
IPV6_ROUTE_1="2001:db8:100::/64"
IPV4_ADDR_1="198.51.100.2/24"
IPV4_ROUTE_1="198.51.100.0/24"
ip link set dev $INTERFACE up
ip link set dev $INTERFACE address $HWADDR
ip -6 address add $IPV6_ADDR_1 dev $INTERFACE nodad
ip -6 route add $IPV6_ROUTE_1 dev $INTERFACE
ip address add $IPV4_ADDR_1 dev $INTERFACE
ip route add $IPV4_ROUTE_1 dev $INTERFACE > /dev/null 2>&1
```

For Zephyr instance #2, create file called `zephyr2.conf` to `net-tools` project, or to some other suitable directory.

```
# Configuration file for setting IP addresses for a network interface.
INTERFACE="$1"
HWADDR="00:00:5e:00:53:22"
IPV6_ADDR_1="2001:db8:200::2"
IPV6_ROUTE_1="2001:db8:200::/64"
IPV4_ADDR_1="203.0.113.2/24"
IPV4_ROUTE_1="203.0.113.0/24"
ip link set dev $INTERFACE up
ip link set dev $INTERFACE address $HWADDR
ip -6 address add $IPV6_ADDR_1 dev $INTERFACE nodad
```

(continues on next page)

(continued from previous page)

```
ip -6 route add $IPV6_ROUTE_1 dev $INTERFACE
ip address add $IPV4_ADDR_1 dev $INTERFACE
ip route add $IPV4_ROUTE_1 dev $INTERFACE > /dev/null 2>&1
```

Step 2 - Create Ethernet interfaces The following `net-setup.sh` commands should be typed in net-tools directory (`cd net-tools`).

In terminal #1, type:

```
./net-setup.sh -c zephyr1.conf -i zeth.1
```

In terminal #2, type:

```
./net-setup.sh -c zephyr2.conf -i zeth.2
```

Step 3 - Setup network bridging In terminal #3, type:

```
sudo brctl addbr zeth-br
sudo brctl addif zeth-br zeth.1
sudo brctl addif zeth-br zeth.2
sudo ifconfig zeth-br up
```

Step 4 - Start Zephyr instances In this example we start sockets-echo-server and sockets-echo-client sample applications. You can use other applications too as needed.

In terminal #4, if you are using QEMU, type this:

```
west build -d build/server -b qemu_x86 -t run \
   samples/net/sockets/echo_server -- \
   -DEXTRA_CONF_FILE=overlay-e1000.conf \
   -DCONFIG_NET_CONFIG_MY_IPV4_ADDR=\"198.51.100.1\" \
   -DCONFIG_NET_CONFIG_PEER_IPV4_ADDR=\"203.0.113.1\" \
   -DCONFIG_NET_CONFIG_MY_IPV6_ADDR=\"2001:db8:100::1\" \
   -DCONFIG_NET_CONFIG_PEER_IPV6_ADDR=\"2001:db8:200::1\" \
   -DCONFIG_NET_CONFIG_MY_IPV4_GW=\"203.0.113.1\" \
   -DCONFIG_ETH_QEMU_IFACE_NAME=\"zeth.1\" \
   -DCONFIG_ETH_QEMU_EXTRA_ARGS=\"mac=00:00:5e:00:53:01\"
```

or if you want to use native_sim board, type this:

```
west build -d build/server -b native_sim -t run \
   samples/net/sockets/echo_server -- \
   -DCONFIG_NET_CONFIG_MY_IPV4_ADDR=\"198.51.100.1\" \
   -DCONFIG_NET_CONFIG_PEER_IPV4_ADDR=\"203.0.113.1\" \
   -DCONFIG_NET_CONFIG_MY_IPV6_ADDR=\"2001:db8:100::1\" \
   -DCONFIG_NET_CONFIG_PEER_IPV6_ADDR=\"2001:db8:200::1\" \
   -DCONFIG_NET_CONFIG_MY_IPV4_GW=\"203.0.113.1\" \
   -DCONFIG_ETH_NATIVE_POSIX_DRV_NAME=\"zeth.1\" \
   -DCONFIG_ETH_NATIVE_POSIX_MAC_ADDR=\"00:00:5e:00:53:01\" \
   -DCONFIG_ETH_NATIVE_POSIX_RANDOM_MAC=n
```

In terminal #5, if you are using QEMU, type this:

```
west build -d build/client -b qemu_x86 -t run \
   samples/net/sockets/echo_client -- \
   -DEXTRA_CONF_FILE=overlay-e1000.conf \
   -DCONFIG_NET_CONFIG_MY_IPV4_ADDR=\"203.0.113.1\" \
```

(continues on next page)

(continued from previous page)

```
   -DCONFIG_NET_CONFIG_PEER_IPV4_ADDR=\"198.51.100.1\" \
   -DCONFIG_NET_CONFIG_MY_IPV6_ADDR=\"2001:db8:200::1\" \
   -DCONFIG_NET_CONFIG_PEER_IPV6_ADDR=\"2001:db8:100::1\" \
   -DCONFIG_NET_CONFIG_MY_IPV4_GW=\"198.51.100.1\" \
   -DCONFIG_ETH_QEMU_IFACE_NAME=\"zeth.2\" \
   -DCONFIG_ETH_QEMU_EXTRA_ARGS=\"mac=00:00:5e:00:53:02\"
```

or if you want to use native_sim board, type this:

```
west build -d build/client -b native_sim -t run \
   samples/net/sockets/echo_client -- \
   -DCONFIG_NET_CONFIG_MY_IPV4_ADDR=\"203.0.113.1\" \
   -DCONFIG_NET_CONFIG_PEER_IPV4_ADDR=\"198.51.100.1\" \
   -DCONFIG_NET_CONFIG_MY_IPV6_ADDR=\"2001:db8:200::1\" \
   -DCONFIG_NET_CONFIG_PEER_IPV6_ADDR=\"2001:db8:100::1\" \
   -DCONFIG_NET_CONFIG_MY_IPV4_GW=\"198.51.100.1\" \
   -DCONFIG_ETH_NATIVE_POSIX_DRV_NAME=\"zeth.2\" \
   -DCONFIG_ETH_NATIVE_POSIX_MAC_ADDR=\"00:00:5e:00:53:02\" \
   -DCONFIG_ETH_NATIVE_POSIX_RANDOM_MAC=n
```

Also if you have firewall enabled in your host, you need to allow traffic between `zeth.1`, `zeth.2` and `zeth-br` interfaces.

Networking with QEMU and IEEE 802.15.4

- *Basic Setup*
  - *Step 1 - Compile and start echo-server*
  - *Step 2 - Compile and start echo-client*

This page describes how to set up a virtual network between two QEMUs that are connected together via UART and are running IEEE 802.15.4 link layer between them. Note that this only works in Linux host.

Basic Setup For the steps below, you will need two terminal windows:

- Terminal #1 is terminal window with `echo-server` Zephyr sample application.
- Terminal #2 is terminal window with `echo-client` Zephyr sample application.

If you want to capture the transferred network data, you must compile the `monitor_15_4` program in `net-tools` directory.

Open a terminal window and type:

```
cd $ZEPHYR_BASE/../net-tools
make monitor_15_4
```

Step 1 - Compile and start echo-server In terminal #1, type:

```
west build -b qemu_x86 -d build/server samples/net/sockets/echo_server -- -DEXTRA_CONF_
↪FILE=overlay-qemu_802154.conf
west build -t server -d build/server
```

If you want to capture the network traffic between the two QEMUs, type:

```
west build -b qemu_x86 -d build/server samples/net/sockets/echo_server -- -G'Unix Makefiles
↪' -DEXTRA_CONF_FILE=overlay-qemu_802154.conf -DPCAP=capture.pcap
west build -t server -d build/server
```

Note that the `make` must be used for `server` target if packet capture option is set in command line. The `build/server/capture.pcap` file will contain the transferred data.

Step 2 - Compile and start echo-client In terminal #2, type:

```
west build -b qemu_x86 -d build/client samples/net/sockets/echo_client -- -DEXTRA_CONF_
↪FILE=overlay-qemu_802154.conf
west build -t client -d build/client
```

You should see data passed between the two QEMUs. Exit QEMU by pressing `CTRL+A x`.

Networking with Arm FVP User Mode

- *Introduction*
- *Using Arm FVP User Mode Networking with Zephyr*
- *Limitations*

This page is intended to serve as a starting point for anyone interested in using Arm FVP user mode networking with Zephyr.

Introduction User mode networking emulates a built-in IP router and DHCP server, and routes TCP and UDP traffic between the guest and host. It uses the user mode socket layer of the host to communicate with other hosts. This allows the use of a significant number of IP network services without requiring administrative privileges, or the installation of a separate driver on the host on which the model is running.

By default, Arm FVP uses the `172.20.51.0/24` network and runs a gateway at `172.20.51.254`. This gateway also functions as a DHCP server for the GOS, allowing it to be automatically assigned with an IP address `172.20.51.1`.

More details about Arm FVP user mode networking can be obtained from here: https://developer.arm.com/documentation/100964/latest/Introduction-to-Fast-Models/User-mode-networking

Using Arm FVP User Mode Networking with Zephyr Arm FVP user mode networking can be enabled in any applications and it doesn't need any configurations on the host system. This feature has been enabled in DHCPv4 client sample. See dhcpv4-client sample application.

Limitations

- You can use TCP and UDP over IP, but not ICMP (ping).
- User mode networking does not support forwarding UDP ports on the host to the model.
- You can only use DHCP within the private network.
- You can only make inward connections by mapping TCP ports on the host to the model. This is common to all implementations that provide host connectivity using NAT.
- Operations that require privileged source ports, for example NFS in its default configuration, do not work.

- If setup fails, or the parameter syntax is incorrect, there is no error reporting.

While developing networking software, it is usually necessary to connect and exchange data with the host system like a Linux desktop computer. Depending on what board is used for development, the following options are possible:

- QEMU using SLIP (Serial Line Internet Protocol).
  - Here IP packets are exchanged between Zephyr and the host system via serial port. This is the legacy way of transferring data. It is also quite slow so use it only when necessary. See *Networking with QEMU* for details.
- QEMU using built-in Ethernet driver.
  - Here IP packets are exchanged between Zephyr and the host system via QEMU's built-in Ethernet driver. Not all QEMU boards support built-in Ethernet so in some cases, you might need to use the SLIP method for host connectivity. See *Networking with QEMU Ethernet* for details.
- QEMU using SLIRP (Qemu User Networking).
  - QEMU User Networking is implemented using "slirp", which provides a full TCP/IP stack within QEMU and uses that stack to implement a virtual NAT'd network. As this support is built into QEMU, it can be used with any model and requires no admin privileges on the host machine, unlike TAP. However, it has several limitations including performance which makes it less valuable for practical purposes. See *Networking with QEMU User* for details.
- Arm FVP (User Mode Networking).
  - User mode networking emulates a built-in IP router and DHCP server, and routes TCP and UDP traffic between the guest and host. It uses the user mode socket layer of the host to communicate with other hosts. This allows the use of a significant number of IP network services without requiring administrative privileges, or the installation of a separate driver on the host on which the model is running. See *Networking with Arm FVP User Mode* for details.
- native_sim board.
  - The Zephyr instance can be executed as a user space process in the host system. This is the most convenient way to debug the Zephyr system as one can attach host debugger directly to the running Zephyr instance. This requires that there is an adaptation driver in Zephyr for interfacing with the host system. Two possible network drivers can be used for this purpose, a TAP virtual Ethernet driver and an offloaded sockets driver. See *Networking with native_sim board* for details.
- USB device networking.
  - Here, the Zephyr instance is run on a real board and the connectivity to the host system is done via USB. See *USB Device Networking* for details.
- Connecting multiple Zephyr instances together.
  - If you have multiple Zephyr instances, either QEMU or native_sim ones, and want to create a connection between them, see *Networking with multiple Zephyr instances* for details.
- Simulating IEEE 802.15.4 network between two QEMUs.
  - Here, two Zephyr instances are running and there is IEEE 802.15.4 link layer run over an UART between them. See *Networking with QEMU and IEEE 802.15.4* for details.

6.3.5 Monitor Network Traffic

- *Host Configuration*
- *Zephyr Configuration*
- *Wireshark Configuration*

It is useful to be able to monitor the network traffic especially when debugging a connectivity issues or when developing new protocol support in Zephyr. This page describes how to set up a way to capture network traffic so that user is able to use Wireshark or similar tool in remote host to see the network packets sent or received by a Zephyr device.

See also the net-capture sample application from the Zephyr source distribution for configuration options that need to be enabled.

Host Configuration

The instructions here describe how to setup a Linux host to capture Zephyr network RX and TX traffic. Similar instructions should work also in other operating systems. On the Linux Host, fetch the Zephyr `net-tools` project, which is located in a separate Git repository:

```
git clone https://github.com/zephyrproject-rtos/net-tools
```

The `net-tools` project provides a configure file to setup IP-to-IP tunnel interface so that we can transfer monitoring data from Zephyr to host.

In terminal #1, type:

```
./net-setup.sh -c zeth-tunnel.conf
```

This script will create following IPIP tunnel interfaces:

Interface name	Description
`zeth-ip6ip`	IPv6-over-IPv4 tunnel
`zeth-ipip`	IPv4-over-IPv4 tunnel
`zeth-ipip6`	IPv4-over-IPv6 tunnel
`zeth-ip6ip6`	IPv6-over-IPv6 tunnel

Zephyr will send captured network packets to one of these interfaces. The actual interface will depend on how the capturing is configured. You can then use Wireshark to monitor the proper network interface.

After the tunneling interfaces have been created, you can use for example `net-capture.py` script from `net-tools` project to print or save the captured network packets. The `net-capture.py` provides an UDP listener, it can print the captured data to screen and optionally can also save the data to a pcap file.

```
$ ./net-capture.py -i zeth-ip6ip -w capture.pcap
[20210408Z14:33:08.959589] Ether / IP / ICMP 192.0.2.1 > 192.0.2.2 echo-request 0 / Raw
[20210408Z14:33:08.976178] Ether / IP / ICMP 192.0.2.2 > 192.0.2.1 echo-reply 0 / Raw
[20210408Z14:33:16.176303] Ether / IPv6 / ICMPv6 Echo Request (id: 0x9feb seq: 0x0)
[20210408Z14:33:16.195326] Ether / IPv6 / ICMPv6 Echo Reply (id: 0x9feb seq: 0x0)
[20210408Z14:33:21.194979] Ether / IPv6 / ICMPv6ND_NS / ICMPv6 Neighbor Discovery Option -␣
↪Source Link-Layer Address 02:00:5e:00:53:3b
[20210408Z14:33:21.217528] Ether / IPv6 / ICMPv6ND_NA / ICMPv6 Neighbor Discovery Option -␣
↪Destination Link-Layer Address 00:00:5e:00:53:ff
[20210408Z14:34:10.245408] Ether / IPv6 / UDP 2001:db8::2:47319 > 2001:db8::1:4242 / Raw
[20210408Z14:34:10.266542] Ether / IPv6 / UDP 2001:db8::1:4242 > 2001:db8::2:47319 / Raw
```

The net-capture.py has following command line options:

```
Listen captured network data from Zephyr and save it optionally to pcap file.
./net-capture.py \
     -i | --interface <network interface>
             Listen this interface for the data
     [-p | --port <UDP port>]
             UDP port (default is 4242) where the capture data is received
     [-q | --quiet]
             Do not print packet information
     [-t | --type <L2 type of the data>]
             Scapy L2 type name of the UDP payload, default is Ether
     [-w | --write <pcap file name>]
             Write the received data to file in PCAP format
```

Instead of the net-capture.py script, you can for example use netcat to provide an UDP listener so that the host will not send port unreachable message to Zephyr:

```
nc -l -u 2001:db8:200::2 4242 > /dev/null
```

The IP address above is the inner tunnel endpoint, and can be changed and it depends on how the Zephyr is configured. Zephyr will send UDP packets containing the captured network packets to the configured IP tunnel, so we need to terminate the network connection like this.

Zephyr Configuration

In this example, we use the native_sim board. You can also use any other board that supports networking.

In terminal #3, type:

```
west build -b native_sim samples/net/capture -- -DCONFIG_NATIVE_UART_AUTOATTACH_DEFAULT_
↪CMD=\""gnome-terminal -- screen %s"\"
```

To see the Zephyr console and shell, start Zephyr instance like this:

```
build/zephyr/zephyr.exe -attach_uart
```

Any other application can be used too, just make sure that suitable configuration options are enabled (see samples/net/capture/prj.conf file for examples).

The network capture can be configured automatically if needed, but currently the capture sample application does not do that. User has to use net-shell to setup and enable the monitoring.

The network packet monitoring needs to be setup first. The net-shell has net capture setup command for doing that. The command syntax is

```
net capture setup <remote-ip-addr> <local-ip-addr> <peer-ip-addr>
     <remote> is the (outer) endpoint IP address
     <local> is the (inner) local IP address
     <peer> is the (inner) peer IP address
     Local and Peer IP addresses can have UDP port number in them (optional)
     like 198.0.51.2:9000 or [2001:db8:100::2]:4242
```

In Zephyr console, type:

```
net capture setup 192.0.2.2 2001:db8:200::1 2001:db8:200::2
```

This command will create the tunneling interface. The 192.0.2.2 is the remote host where the tunnel is terminated. The address is used to select the local network interface where the tunneling interface is attached to. The 2001:db8:200::1 tells the local IP address for the tunnel, the

2001:db8:200::2 is the peer IP address where the captured network packets are sent. The port numbers for UDP packet can be given in the setup command like this for IPv6-over-IPv4 tunnel

```
net capture setup 192.0.2.2 [2001:db8:200::1]:9999 [2001:db8:200::2]:9998
```

and like this for IPv4-over-IPv4 tunnel

```
net capture setup 192.0.2.2 198.51.100.1:9999 198.51.100.2:9998
```

If the port number is omitted, then 4242 UDP port is used as a default.

The current monitoring configuration can be checked like this:

```
uart:~$ net capture
Network packet capture disabled
                Capture  Tunnel
Device          iface    iface   Local                   Peer
NET_CAPTURE0    -        1      [2001:db8:200::1]:4242  [2001:db8:200::2]:4242
```

which will print the current configuration. As we have not yet enabled monitoring, the Capture iface is not set.

Then we need to enable the network packet monitoring like this:

```
net capture enable 2
```

The 2 tells the network interface which traffic we want to capture. In this example, the 2 is the native_sim board Ethernet interface. Note that we send the network traffic to the same interface that we are monitoring in this example. The monitoring system avoids to capture already captured network traffic as that would lead to recursion. You can use net iface command to see what network interfaces are available. Note that you cannot capture traffic from the tunnel interface as that would cause recursion loop. The captured network traffic can be sent to some other network interface if configured so. Just set the <remote-ip-addr> option properly in net capture setup so that the IP tunnel is attached to desired network interface. The capture status can be checked again like this:

```
uart:~$ net capture
Network packet capture enabled
                Capture  Tunnel
Device          iface    iface   Local                   Peer
NET_CAPTURE0    2        1      [2001:db8:200::1]:4242  [2001:db8:200::2]:4242
```

After enabling the monitoring, the system will send captured (either received or sent) network packets to the tunnel interface for further processing.

The monitoring can be disabled like this:

```
net capture disable
```

which will turn currently running monitoring off. The monitoring setup can be cleared like this:

```
net capture cleanup
```

It is not necessary to use net-shell for configuring the monitoring. The *network capture API* functions can be called by the application if needed.

Wireshark Configuration

The Wireshark tool can be used to monitor the captured network traffic in a useful way.

You can monitor either the tunnel interfaces or the zeth interface. In order to see the actual captured data inside an UDP packet, see Wireshark decapsulate UDP document for instructions.

6.3.6 Networking APIs

Zephyr provides support for the standard BSD socket APIs (defined in include/zephyr/net/socket.h) for the applications to use. See *BSD socket API* for more details.

Apart of the standard API, Zephyr provides a set of custom networking APIs and libraries for the application to use. See the list below for details.

Note

The legacy connectivity API in include/zephyr/net/net_context.h should not be used by applications.

Network APIs

BSD Sockets

- *Overview*
- *Secure Sockets*
  - *TLS credentials subsystem*
  - *Secure Socket Creation*
  - *Secure Sockets options*
- *Socket offloading*
  - *Offloaded socket creation*
  - *Dealing with multiple offloaded interfaces*
- *API Reference*
  - *BSD Sockets*
  - *TLS Credentials*

Overview Zephyr offers an implementation of a subset of the BSD Sockets API (a part of the POSIX standard). This API allows to reuse existing programming experience and port existing simple networking applications to Zephyr.

Here are the key requirements and concepts which governed BSD Sockets compatible API implementation for Zephyr:

- Has minimal overhead, similar to the requirement for other Zephyr subsystems.
- Is namespaced by default, to avoid name conflicts with well-known names like `close()`, which may be part of libc or other POSIX compatibility libraries. If enabled by `CONFIG_POSIX_API`, it will also expose native POSIX names.

BSD Sockets compatible API is enabled using `CONFIG_NET_SOCKETS` config option and implements the following operations: `socket()`, `close()`, `recv()`, `recvfrom()`, `send()`, `sendto()`, `connect()`, `bind()`, `listen()`, `accept()`, `fcntl()` (to set non-blocking mode), `getsockopt()`, `setsockopt()`, `poll()`, `select()`, `getaddrinfo()`, `getnameinfo()`.

Based on the namespacing requirements above, these operations are by default exposed as functions with `zsock_` prefix, e.g. *`zsock_socket()`* and *`zsock_close()`*. If the config option `CONFIG_POSIX_API` is defined, all the functions will be also exposed as aliases without the prefix.

This includes the functions like `close()` and `fcntl()` (which may conflict with functions in libc or other libraries, for example, with the filesystem libraries).

Another entailment of the design requirements above is that the Zephyr API aggressively employs the short-read/short-write property of the POSIX API whenever possible (to minimize complexity and overheads). POSIX allows for calls like `recv()` and `send()` to actually process (receive or send) less data than requested by the user (on `SOCK_STREAM` type sockets). For example, a call `recv(sock, 1000, 0)` may return 100, meaning that only 100 bytes were read (short read), and the application needs to retry call(s) to receive the remaining 900 bytes.

The BSD Sockets API uses file descriptors to represent sockets. File descriptors are small integers, consecutively assigned from zero, shared among sockets, files, special devices (like stdin/stdout), etc. Internally, there is a table mapping file descriptors to internal object pointers. The file descriptor table is used by the BSD Sockets API even if the rest of the POSIX subsystem (filesystem, stdin/stdout) is not enabled.

See sockets-echo-server and sockets-echo-client sample applications to learn how to create a simple server or client BSD socket based application.

Secure Sockets Zephyr provides an extension of standard POSIX socket API, allowing to create and configure sockets with TLS protocol types, facilitating secure communication. Secure functions for the implementation are provided by mbedTLS library. Secure sockets implementation allows use of both TLS and DTLS protocols with standard socket calls. See *net_ip_protocol_secure* type for supported secure protocol versions.

To enable secure sockets, set the `CONFIG_NET_SOCKETS_SOCKOPT_TLS` option. To enable DTLS support, use `CONFIG_NET_SOCKETS_ENABLE_DTLS` option.

TLS credentials subsystem TLS credentials must be registered in the system before they can be used with secure sockets. See *tls_credential_add()* for more information.

When a specific TLS credential is registered in the system, it is assigned with numeric value of type *sec_tag_t*, called a tag. This value can be used later on to reference the credential during secure socket configuration with socket options.

The following TLS credential types can be registered in the system:

- `TLS_CREDENTIAL_CA_CERTIFICATE`
- `TLS_CREDENTIAL_SERVER_CERTIFICATE`
- `TLS_CREDENTIAL_PRIVATE_KEY`
- `TLS_CREDENTIAL_PSK`
- `TLS_CREDENTIAL_PSK_ID`

An example registration of CA certificate (provided in `ca_certificate` array) looks like this:

```
ret = tls_credential_add(CA_CERTIFICATE_TAG, TLS_CREDENTIAL_CA_CERTIFICATE,
                         ca_certificate, sizeof(ca_certificate));
```

By default certificates in DER format are supported. PEM support can be enabled in mbedTLS settings.

Secure Socket Creation A secure socket can be created by specifying secure protocol type, for instance:

```
sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TLS_1_2);
```

Once created, it can be configured with socket options. For instance, the CA certificate and hostname can be set:

```
sec_tag_t sec_tag_opt[] = {
        CA_CERTIFICATE_TAG,
};

ret = setsockopt(sock, SOL_TLS, TLS_SEC_TAG_LIST,
                 sec_tag_opt, sizeof(sec_tag_opt));
```

```
char host[] = "google.com";

ret = setsockopt(sock, SOL_TLS, TLS_HOSTNAME, host, sizeof(host));
```

Once configured, socket can be used just like a regular TCP socket.

Several samples in Zephyr use secure sockets for communication. For a sample use see e.g. echo-server sample application or HTTP GET sample application.

Secure Sockets options Secure sockets offer the following options for socket management:

Related code samples

HTTP Client
Implement an HTTP(S) client that issues a variety of HTTP requests.

HTTP GET using plain sockets
Implement an HTTP(S) client using plain BSD sockets.

group `secure_sockets_options`

Socket options for TLS

`SOL_TLS`

Protocol level for TLS.

Here, the same socket protocol level for TLS as in Linux was used.

`TLS_SEC_TAG_LIST`

Socket option to select TLS credentials to use.

It accepts and returns an array of sec_tag_t that indicate which TLS credentials should be used with specific socket.

`TLS_HOSTNAME`

Write-only socket option to set hostname.

It accepts a string containing the hostname (may be NULL to disable hostname verification). By default, hostname check is enforced for TLS clients.

`TLS_CIPHERSUITE_LIST`

Socket option to select ciphersuites to use.

It accepts and returns an array of integers with IANA assigned ciphersuite identifiers. If not set, socket will allow all ciphersuites available in the system (mbedTLS default behavior).

`TLS_CIPHERSUITE_USED`

Read-only socket option to read a ciphersuite chosen during TLS handshake.

It returns an integer containing an IANA assigned ciphersuite identifier of chosen ciphersuite.

`TLS_PEER_VERIFY`

Write-only socket option to set peer verification level for TLS connection.

This option accepts an integer with a peer verification level, compatible with mbedTLS values:

- 0 - none
- 1 - optional
- 2 - required

If not set, socket will use mbedTLS defaults (none for servers, required for clients).

`TLS_DTLS_ROLE`

Write-only socket option to set role for DTLS connection.

This option is irrelevant for TLS connections, as for them role is selected based on *connect()*/listen() usage. By default, DTLS will assume client role. This option accepts an integer with a TLS role, compatible with mbedTLS values:

- 0 - client
- 1 - server

`TLS_ALPN_LIST`

Socket option for setting the supported Application Layer Protocols.

It accepts and returns a const char array of NULL terminated strings representing the supported application layer protocols listed during the TLS handshake.

`TLS_DTLS_HANDSHAKE_TIMEOUT_MIN`

Socket option to set DTLS min handshake timeout.

The timeout starts at min, and upon retransmission the timeout is doubled util max is reached. Min and max arguments are separate options. The time unit is ms.

`TLS_DTLS_HANDSHAKE_TIMEOUT_MAX`

Socket option to set DTLS max handshake timeout.

The timeout starts at min, and upon retransmission the timeout is doubled util max is reached. Min and max arguments are separate options. The time unit is ms.

`TLS_CERT_NOCOPY`

Socket option for preventing certificates from being copied to the mbedTLS heap if possible.

The option is only effective for DER certificates and is ignored for PEM certificates.

`TLS_NATIVE`

TLS socket option to use with offloading.

The option instructs the network stack only to offload underlying TCP/UDP communication. The TLS/DTLS operation is handled by a native TLS/DTLS socket implementation from Zephyr.

Note, that this option is only applicable if socket dispatcher is used (CONFIG_NET_SOCKETS_OFFLOAD_DISPATCHER is enabled). In such case, it should be the first socket option set on a newly created socket. After that, the application may use SO_BINDTODEVICE to choose the dedicated network interface for the underlying TCP/UDP socket.

`TLS_SESSION_CACHE`

Socket option to control TLS session caching on a socket.

Accepted values:

- 0 - Disabled.
- 1 - Enabled.

`TLS_SESSION_CACHE_PURGE`

Write-only socket option to purge session cache immediately.

This option accepts any value.

`TLS_DTLS_CID`

Write-only socket option to control DTLS CID.

The option accepts an integer, indicating the setting. Accepted values for the option are: 0, 1 and 2. Effective when set before connecting to the socket.

- 0 - DTLS CID will be disabled.
- 1 - DTLS CID will be enabled, and a 0 length CID value to be sent to the peer.
- 2 - DTLS CID will be enabled, and the most recent value set with TLS_DTLS_CID_VALUE will be sent to the peer. Otherwise, a random value will be used.

`TLS_DTLS_CID_STATUS`

Read-only socket option to get DTLS CID status.

The option accepts a pointer to an integer, indicating the setting upon return. Returned values for the option are:

- 0 - DTLS CID is disabled.
- 1 - DTLS CID is received on the downlink.
- 2 - DTLS CID is sent to the uplink.
- 3 - DTLS CID is used in both directions.

`TLS_DTLS_CID_VALUE`

Socket option to set or get the value of the DTLS connection ID to be used for the DTLS session.

The option accepts a byte array, holding the CID value.

`TLS_DTLS_PEER_CID_VALUE`

Read-only socket option to get the value of the DTLS connection ID received from the peer.

The option accepts a pointer to a byte array, holding the CID value upon return. The optlen returned will be 0 if the peer did not provide a connection ID, otherwise will contain the length of the CID value.

`TLS_DTLS_HANDSHAKE_ON_CONNECT`
: Socket option to configure DTLS socket behavior on *connect()*.

  If set, DTLS *connect()* will execute the handshake with the configured peer. This is the default behavior. Otherwise, DTLS *connect()* will only configure peer address (as with regular UDP socket) and will not attempt to execute DTLS handshake. The handshake will take place in consecutive *send()*/recv() call.

`TLS_PEER_VERIFY_NONE`
: Peer verification disabled.

`TLS_PEER_VERIFY_OPTIONAL`
: Peer verification optional.

`TLS_PEER_VERIFY_REQUIRED`
: Peer verification required.

`TLS_DTLS_ROLE_CLIENT`
: Client role in a DTLS session.

`TLS_DTLS_ROLE_SERVER`
: Server role in a DTLS session.

`TLS_CERT_NOCOPY_NONE`
: Cert duplicated in heap.

`TLS_CERT_NOCOPY_OPTIONAL`
: Cert not copied in heap if DER.

`TLS_SESSION_CACHE_DISABLED`
: Disable TLS session caching.

`TLS_SESSION_CACHE_ENABLED`
: Enable TLS session caching.

`TLS_DTLS_CID_DISABLED`
: CID is disabled

`TLS_DTLS_CID_SUPPORTED`
: CID is supported.

`TLS_DTLS_CID_ENABLED`
: CID is enabled

`TLS_DTLS_CID_STATUS_DISABLED`
: CID is disabled.

`TLS_DTLS_CID_STATUS_DOWNLINK`
: CID is in use by us.

`TLS_DTLS_CID_STATUS_UPLINK`
: CID is in use by peer.

`TLS_DTLS_CID_STATUS_BIDIRECTIONAL`
: CID is in use by us and peer.

Socket offloading Zephyr allows to register custom socket implementations (called offloaded sockets). This allows for seamless integration for devices which provide an external IP stack and expose socket-like API.

Socket offloading can be enabled with `CONFIG_NET_SOCKETS_OFFLOAD` option. A network driver that wants to register a new socket implementation should use `NET_SOCKET_OFFLOAD_REGISTER` macro. The macro accepts the following parameters:

- **`socket_name`**
  An arbitrary name for the socket implementation.
- **`prio`**
  Socket implementation's priority. The higher the priority, the earlier this particular implementation will be processed when creating a new socket. Lower numeric value indicates higher priority.
- **`_family`**
  Socket family implemented by the offloaded socket. `AF_UNSPEC` indicates any family.
- **`_is_supported`**
  A filtering function, used to verify whether a particular socket family, type and protocol are supported by the offloaded socket implementation.
- **`_handler`**
  A function compatible with *`socket()`* API, used to create an offloaded socket.

Every offloaded socket implementation should also implement a set of socket APIs, specified in `socket_op_vtable` struct.

The function registered for socket creation should allocate a new file descriptor using `zvfs_reserve_fd()` function. Any additional actions, specific to the creation of a particular offloaded socket implementation, should take place after the file descriptor is allocated. As a final step, if the offloaded socket was created successfully, the file descriptor should be finalized with `zvfs_finalize_typed_fd()`, or `zvfs_finalize_fd()` functions. The finalize function allows to register a `socket_op_vtable` structure implementing socket APIs for an offloaded socket along with an optional socket context data pointer.

Finally, when an offloaded network interface is initialized, it should indicate that the interface is offloaded with *`net_if_socket_offload_set()`* function. The function registers the function used to create an offloaded socket (the same as the one provided in `NET_SOCKET_OFFLOAD_REGISTER`) at the network interface.

Offloaded socket creation When application creates a new socket with *`socket()`* function, the network stack iterates over all registered socket implementations (native and offloaded). Higher priority socket implementations are processed first. For each registered socket implementation, an address family is verified, and if it matches (or the socket was registered as `AF_UNSPEC`), the corresponding `_is_supported` function is called to verify the remaining socket parameters. The first implementation that fulfills the socket requirements (i. e. `_is_supported` returns true) will create a new socket with its `_handler` function.

The above indicates the importance of the socket priority. If multiple socket implementations support the same set of socket family/type/protocol, the first implementation processed by the system will create a socket. Therefore it's important to give the highest priority to the implementation that should be the system default.

The socket priority for native socket implementation is configured with Kconfig. Use CONFIG_NET_SOCKETS_TLS_PRIORITY to set the priority for the native TLS sockets. Use CONFIG_NET_SOCKETS_PRIORITY_DEFAULT to set the priority for the remaining native sockets.

Dealing with multiple offloaded interfaces As the *socket()* function does not allow to specify which network interface should be used by a socket, it's not possible to choose a specific implementation in case multiple offloaded socket implementations, supporting the same type of sockets, are available. The same problem arises when both native and offloaded sockets are available in the system.

To address this problem, a special socket implementation (called socket dispatcher) was introduced. The sole reason for this module is to postpone the socket creation for until the first operation on a socket is performed. This leaves an opening to use SO_BINDTODEVICE socket option, to bind a socket to a particular network interface (and thus offloaded socket implementation). The socket dispatcher can be enabled with CONFIG_NET_SOCKETS_OFFLOAD_DISPATCHER Kconfig option.

When enabled, the application can specify the network interface to use with *setsockopt()* function:

```
/* A "dispatcher" socket is created */
sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

struct ifreq ifreq = {
   .ifr_name = "SimpleLink"
};

/* The socket is "dispatched" to a particular network interface
 * (offloaded or not).
 */
setsockopt(sock, SOL_SOCKET, SO_BINDTODEVICE, &ifreq, sizeof(ifreq));
```

Similarly, if TLS is supported by both native and offloaded sockets, TLS_NATIVE socket option can be used to indicate that a native TLS socket should be created. The underlying socket can then be bound to a particular network interface:

```
/* A "dispatcher" socket is created */
sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TLS_1_2);

int tls_native = 1;

/* The socket is "dispatched" to a native TLS socket implmeentation.
 * The underlying socket is a "dispatcher" socket now.
 */
setsockopt(sock, SOL_TLS, TLS_NATIVE, &tls_native, sizeof(tls_native));

struct ifreq ifreq = {
   .ifr_name = "SimpleLink"
};

/* The underlying socket is "dispatched" to a particular network interface
 * (offloaded or not).
 */
setsockopt(sock, SOL_SOCKET, SO_BINDTODEVICE, &ifreq, sizeof(ifreq));
```

In case no SO_BINDTODEVICE socket option is used on a socket, the socket will be dispatched according to the default priority and filtering rules on a first socket API call.

API Reference

Related code samples

AWS IoT Core MQTT
Connect to AWS IoT Core and publish messages using MQTT.

Asynchronous echo server using poll()
Implement an asynchronous IPv4/IPv6 TCP echo server using BSD sockets and poll()

Asynchronous echo server using select()
Implement an asynchronous IPv4/IPv6 TCP echo server using BSD sockets and select()

Dumb HTTP server
Implement a simple, portable, HTTP server using BSD sockets.

Dumb HTTP server (multi-threaded)
Implement a simple HTTP server supporting simultaneous connections using BSD sockets.

Echo client (advanced)
Implement a client that sends IP packets, waits for data to be sent back, and verifies it.

Echo server (advanced)
Implement a UDP/TCP server that sends received packets back to the sender.

Echo server (service)
Implements a simple IPv4/IPv6 TCP echo server using BSD sockets and socket service API.

Echo server (simple)
Implements a simple IPv4/IPv6 TCP echo server using BSD sockets.

HTTP Client
Implement an HTTP(S) client that issues a variety of HTTP requests.

HTTP GET using plain sockets
Implement an HTTP(S) client using plain BSD sockets.

Large HTTP download
Download a large file from a web server using BSD sockets.

Microsoft Azure IoT Hub MQTT
Connect to Azure IoT Hub and publish messages using MQTT.

Modbus TCP server
Implement a Modbus TCP server exposing Modbus commands to control LEDs.

Modbus TCP-to-serial gateway
Implement a gateway between an Ethernet TCP-IP network and a Modbus serial line.

Network management socket
Listen to network management events using a network management socket.

Packet socket
Use raw packet sockets over Ethernet.

SNTP client
Use SNTP to get the current time from the host.

SocketCAN
Send and receive raw CAN frames using BSD sockets API.

Socketpair
Implement communication between threads using socket pairs.

TCP sample for TTCN-3 based sanity check
Use TTCN-3 to validate the functionality of the TCP stack.

TagoIO HTTP Post
Send random temperature values to TagoIO IoT Cloud Platform using HTTP.

UDP sender using SO_TXTIME
Control the transmission time of a packet using SO_TXTIME socket option.

Video TCP server sink
Capture video frames and send them over the network to a TCP client.

WebSocket Client
Implement a Websocket client that connects to a Websocket server.

mDNS responder
Listen and respond to mDNS queries.

BSD Sockets

group `bsd_sockets`

BSD Sockets compatible API.

Socket APIs available if CONFIG_NET_SOCKETS_POSIX_NAMES is enabled

static inline int **socket**(int family, int type, int proto)
POSIX wrapper for *zsock_socket*.

static inline int **socketpair**(int family, int type, int proto, int sv[2])
POSIX wrapper for *zsock_socketpair*.

static inline int **close**(int sock)
POSIX wrapper for *zsock_close*.

static inline int **shutdown**(int sock, int how)
POSIX wrapper for *zsock_shutdown*.

static inline int **bind**(int sock, const struct *sockaddr* *addr, *socklen_t* addrlen)
POSIX wrapper for *zsock_bind*.

static inline int **connect**(int sock, const struct *sockaddr* *addr, *socklen_t* addrlen)
POSIX wrapper for *zsock_connect*.

static inline int **listen**(int sock, int backlog)
POSIX wrapper for *zsock_listen*.

static inline int **accept**(int sock, struct *sockaddr* *addr, *socklen_t* *addrlen)
POSIX wrapper for *zsock_accept*.

static inline ssize_t **send**(int sock, const void *buf, size_t len, int flags)
POSIX wrapper for *zsock_send*.

static inline ssize_t **recv**(int sock, void *buf, size_t max_len, int flags)
POSIX wrapper for *zsock_recv*.

static inline ssize_t **sendto**(int sock, const void *buf, size_t len, int flags, const struct *sockaddr* *dest_addr, *socklen_t* addrlen)
POSIX wrapper for *zsock_sendto*.

static inline ssize_t **sendmsg**(int sock, const struct *msghdr* *message, int flags)
POSIX wrapper for *zsock_sendmsg*.

static inline ssize_t **recvfrom**(int sock, void *buf, size_t max_len, int flags, struct *sockaddr* *src_addr, *socklen_t* *addrlen)

POSIX wrapper for *zsock_recvfrom*.

static inline ssize_t **recvmsg**(int sock, struct *msghdr* *msg, int flags)

POSIX wrapper for *zsock_recvmsg*.

static inline int **poll**(struct *zsock_pollfd* *fds, int nfds, int timeout)

POSIX wrapper for *zsock_poll*.

static inline int **getsockopt**(int sock, int level, int optname, void *optval, *socklen_t* *optlen)

POSIX wrapper for *zsock_getsockopt*.

static inline int **setsockopt**(int sock, int level, int optname, const void *optval, *socklen_t* optlen)

POSIX wrapper for *zsock_setsockopt*.

static inline int **getpeername**(int sock, struct *sockaddr* *addr, *socklen_t* *addrlen)

POSIX wrapper for *zsock_getpeername*.

static inline int **getsockname**(int sock, struct *sockaddr* *addr, *socklen_t* *addrlen)

POSIX wrapper for *zsock_getsockname*.

static inline int **getaddrinfo**(const char *host, const char *service, const struct *zsock_addrinfo* *hints, struct *zsock_addrinfo* **res)

POSIX wrapper for *zsock_getaddrinfo*.

static inline void **freeaddrinfo**(struct *zsock_addrinfo* *ai)

POSIX wrapper for *zsock_freeaddrinfo*.

static inline const char ***gai_strerror**(int errcode)

POSIX wrapper for *zsock_gai_strerror*.

static inline int **getnameinfo**(const struct *sockaddr* *addr, *socklen_t* addrlen, char *host, *socklen_t* hostlen, char *serv, *socklen_t* servlen, int flags)

POSIX wrapper for *zsock_getnameinfo*.

static inline int **gethostname**(char *buf, size_t len)

POSIX wrapper for *zsock_gethostname*.

static inline int **inet_pton**(*sa_family_t* family, const char *src, void *dst)

POSIX wrapper for *zsock_inet_pton*.

static inline char ***inet_ntop**(*sa_family_t* family, const void *src, char *dst, size_t size)

POSIX wrapper for *zsock_inet_ntop*.

`pollfd`

POSIX wrapper for *zsock_pollfd*.

`addrinfo`

POSIX wrapper for *zsock_addrinfo*.

`POLLIN`

POSIX wrapper for *ZSOCK_POLLIN*.

`POLLOUT`

POSIX wrapper for *ZSOCK_POLLOUT*.

`POLLERR`
: POSIX wrapper for *ZSOCK_POLLERR*.

`POLLHUP`
: POSIX wrapper for *ZSOCK_POLLHUP*.

`POLLNVAL`
: POSIX wrapper for *ZSOCK_POLLNVAL*.

`MSG_PEEK`
: POSIX wrapper for *ZSOCK_MSG_PEEK*.

`MSG_CTRUNC`
: POSIX wrapper for *ZSOCK_MSG_CTRUNC*.

`MSG_TRUNC`
: POSIX wrapper for *ZSOCK_MSG_TRUNC*.

`MSG_DONTWAIT`
: POSIX wrapper for *ZSOCK_MSG_DONTWAIT*.

`MSG_WAITALL`
: POSIX wrapper for *ZSOCK_MSG_WAITALL*.

`SHUT_RD`
: POSIX wrapper for *ZSOCK_SHUT_RD*.

`SHUT_WR`
: POSIX wrapper for *ZSOCK_SHUT_WR*.

`SHUT_RDWR`
: POSIX wrapper for *ZSOCK_SHUT_RDWR*.

`EAI_BADFLAGS`
: POSIX wrapper for *DNS_EAI_BADFLAGS*.

`EAI_NONAME`
: POSIX wrapper for *DNS_EAI_NONAME*.

`EAI_AGAIN`
: POSIX wrapper for *DNS_EAI_AGAIN*.

`EAI_FAIL`
: POSIX wrapper for *DNS_EAI_FAIL*.

`EAI_NODATA`
: POSIX wrapper for *DNS_EAI_NODATA*.

`EAI_MEMORY`
: POSIX wrapper for *DNS_EAI_MEMORY*.

`EAI_SYSTEM`
: POSIX wrapper for *DNS_EAI_SYSTEM*.

`EAI_SERVICE`
: POSIX wrapper for *DNS_EAI_SERVICE*.

`EAI_SOCKTYPE`
: POSIX wrapper for *DNS_EAI_SOCKTYPE*.

`EAI_FAMILY`
: POSIX wrapper for *DNS_EAI_FAMILY*.

Options for poll()

`ZSOCK_POLLIN`
: zsock_poll: Poll for readability

`ZSOCK_POLLPRI`
: zsock_poll: Poll for exceptional condition

`ZSOCK_POLLOUT`
: zsock_poll: Poll for writability

`ZSOCK_POLLERR`
: zsock_poll: Poll results in error condition (output value only)

`ZSOCK_POLLHUP`
: zsock_poll: Poll detected closed connection (output value only)

`ZSOCK_POLLNVAL`
: zsock_poll: Invalid socket (output value only)

Options for sending and receiving data

`ZSOCK_MSG_PEEK`
: zsock_recv: Read data without removing it from socket input queue

`ZSOCK_MSG_CTRUNC`
: zsock_recvmsg: Control data buffer too small.

`ZSOCK_MSG_TRUNC`
: zsock_recv: return the real length of the datagram, even when it was longer than the passed buffer

`ZSOCK_MSG_DONTWAIT`
: zsock_recv/zsock_send: Override operation to non-blocking

`ZSOCK_MSG_WAITALL`
: zsock_recv: block until the full amount of data can be returned

Options for shutdown() function

`ZSOCK_SHUT_RD`
: zsock_shutdown: Shut down for reading

`ZSOCK_SHUT_WR`
: zsock_shutdown: Shut down for writing

`ZSOCK_SHUT_RDWR`
: zsock_shutdown: Shut down for both reading and writing

Flags for getaddrinfo() hints

`AI_PASSIVE`
: Address for *bind()* (vs for *connect()*)

`AI_CANONNAME`
: Fill in ai_canonname.

`AI_NUMERICHOST`
: Assume host address is in numeric notation, don't DNS lookup.

`AI_V4MAPPED`
: May return IPv4 mapped address for IPv6

`AI_ALL`
: May return both native IPv6 and mapped IPv4 address for IPv6.

`AI_ADDRCONFIG`
: IPv4/IPv6 support depends on local system config.

`AI_NUMERICSERV`
: Assume service (port) is numeric.

`AI_EXTFLAGS`
: Extra flags present (see RFC 5014)

Flags for getnameinfo()

NI_NUMERICHOST
: *zsock_getnameinfo()*: Resolve to numeric address.

NI_NUMERICSERV
: *zsock_getnameinfo()*: Resolve to numeric port number.

NI_NOFQDN
: *zsock_getnameinfo()*: Return only hostname instead of FQDN

NI_NAMEREQD
: *zsock_getnameinfo()*: Dummy option for compatibility

NI_DGRAM
: *zsock_getnameinfo()*: Dummy option for compatibility

NI_MAXHOST
: *zsock_getnameinfo()*: Max supported hostname length

Network interface name description

IFNAMSIZ
: Network interface name length.

Socket level options (SOL_SOCKET)

SOL_SOCKET
: Socket-level option.

SO_DEBUG
: Recording debugging information (ignored, for compatibility)

SO_REUSEADDR
: address reuse

SO_TYPE
: Type of the socket.

SO_ERROR
: Async error.

SO_DONTROUTE
: Bypass normal routing and send directly to host (ignored, for compatibility)

SO_BROADCAST
: Transmission of broadcast messages is supported (ignored, for compatibility)

`SO_SNDBUF`
: Size of socket send buffer.

`SO_RCVBUF`
: Size of socket recv buffer.

`SO_KEEPALIVE`
: Enable sending keep-alive messages on connections.

`SO_OOBINLINE`
: Place out-of-band data into receive stream (ignored, for compatibility)

`SO_PRIORITY`
: Socket priority.

`SO_LINGER`
: Socket lingers on close (ignored, for compatibility)

`SO_REUSEPORT`
: Allow multiple sockets to reuse a single port.

`SO_RCVLOWAT`
: Receive low watermark (ignored, for compatibility)

`SO_SNDLOWAT`
: Send low watermark (ignored, for compatibility)

`SO_RCVTIMEO`
: Receive timeout Applies to receive functions like *recv()*, but not to *connect()*

`SO_SNDTIMEO`
: Send timeout.

`SO_BINDTODEVICE`
: Bind a socket to an interface.

`SO_ACCEPTCONN`
: Socket accepts incoming connections (ignored, for compatibility)

`SO_TIMESTAMPING`
: Timestamp TX RX or both packets.

  Supports multiple timestamp sources.

`SO_PROTOCOL`
: Protocol used with the socket.

`SO_DOMAIN`
: Domain used with SOCKET.

`SO_SOCKS5`
: Enable SOCKS5 for Socket.

`SO_TXTIME`
: Socket TX time (when the data should be sent)

`SCM_TXTIME`
: Socket TX time (same as SO_TXTIME)

`SOF_TIMESTAMPING_RX_HARDWARE`
: Timestamp generation flags.

  Request RX timestamps generated by network adapter.

`SOF_TIMESTAMPING_TX_HARDWARE`
: Request TX timestamps generated by network adapter.

  This can be enabled via socket option or control messages.

TCP level options (IPPROTO_TCP)

`TCP_NODELAY`
: Disable TCP buffering (ignored, for compatibility)

`TCP_KEEPIDLE`
: Start keepalives after this period (seconds)

`TCP_KEEPINTVL`
: Interval between keepalives (seconds)

`TCP_KEEPCNT`
: Number of keepalives before dropping connection.

IPv4 level options (IPPROTO_IP)

`IP_TOS`
: Set or receive the Type-Of-Service value for an outgoing packet.

`IP_TTL`
: Set or receive the Time-To-Live value for an outgoing packet.

`IP_PKTINFO`
: Pass an IP_PKTINFO ancillary message that contains a pktinfo structure that supplies some information about the incoming packet.

`IP_MULTICAST_TTL`
: Set IPv4 multicast TTL value.

`IP_ADD_MEMBERSHIP`
: Join IPv4 multicast group.

`IP_DROP_MEMBERSHIP`
: Leave IPv4 multicast group.

IPv6 level options (IPPROTO_IPV6)

`IPV6_UNICAST_HOPS`
: Set the unicast hop limit for the socket.

`IPV6_MULTICAST_HOPS`
: Set the multicast hop limit for the socket.

`IPV6_ADD_MEMBERSHIP`
: Join IPv6 multicast group.

`IPV6_DROP_MEMBERSHIP`
: Leave IPv6 multicast group.

`IPV6_V6ONLY`
: Don't support IPv4 access.

`IPV6_RECVPKTINFO`
: Pass an IPV6_RECVPKTINFO ancillary message that contains a *in6_pktinfo* structure that supplies some information about the incoming packet.

  See RFC 3542.

`IPV6_ADDR_PREFERENCES`
: RFC5014: Source address selection.

`IPV6_PREFER_SRC_TMP`
: Prefer temporary address as source.

`IPV6_PREFER_SRC_PUBLIC`
: Prefer public address as source.

`IPV6_PREFER_SRC_PUBTMP_DEFAULT`
: Either public or temporary address is selected as a default source depending on the output interface configuration (this is the default value).

  This is Linux specific option not found in the RFC.

`IPV6_PREFER_SRC_COA`
: Prefer Care-of address as source.

  Ignored in Zephyr.

IPV6_PREFER_SRC_HOME
: Prefer Home address as source.

  Ignored in Zephyr.

IPV6_PREFER_SRC_CGA
: Prefer CGA (Cryptographically Generated Address) address as source.

  Ignored in Zephyr.

IPV6_PREFER_SRC_NONCGA
: Prefer non-CGA address as source.

  Ignored in Zephyr.

IPV6_TCLASS
: Set or receive the traffic class value for an outgoing packet.

Backlog size for listen()

SOMAXCONN
: listen: The maximum backlog queue length

Defines

ZSOCK_FD_SETSIZE
: Number of file descriptors which can be added to *zsock_fd_set*.

Typedefs

typedef struct *zsock_fd_set* **zsock_fd_set**
: Socket file descriptor set.

Functions

void ***zsock_get_context_object**(int sock)
: Obtain a file descriptor's associated net context.

  With CONFIG_USERSPACE enabled, the kernel's object permission system must apply to socket file descriptors. When a socket is opened, by default only the caller has permission, access by other threads will fail unless they have been specifically granted permission.

  This is achieved by tagging data structure definitions that implement the underlying object associated with a network socket file descriptor with '__net_socket'. All pointers to instances of these will be known to the kernel as kernel objects with type K_OBJ_NET_SOCKET.

  This API is intended for threads that need to grant access to the object associated with a particular file descriptor to another thread. The returned pointer represents the underlying K_OBJ_NET_SOCKET and may be passed to APIs like *k_object_access_grant()*.

In a system like Linux which has the notion of threads running in processes in a shared virtual address space, this sort of management is unnecessary as the scope of file descriptors is implemented at the process level.

However in Zephyr the file descriptor scope is global, and MPU-based systems are not able to implement a process-like model due to the lack of memory virtualization hardware. They use discrete object permissions and memory domains instead to define thread access scope.

User threads will have no direct access to the returned object and will fault if they try to access its memory; the pointer can only be used to make permission assignment calls, which follow exactly the rules for other kernel objects like device drivers and IPC.

Parameters

- **sock** – file descriptor

Returns

pointer to associated network socket object, or NULL if the file descriptor wasn't valid or the caller had no access permission

int **zsock_socket**(int family, int type, int proto)

Create a network socket.

See POSIX.1-2017 article for normative description. This function is also exposed as `socket()` if `CONFIG_POSIX_API` is defined.

If CONFIG_USERSPACE is enabled, the caller will be granted access to the context object associated with the returned file descriptor.

See also

zsock_get_context_object()

int **zsock_socketpair**(int family, int type, int proto, int *sv)

Create an unnamed pair of connected sockets.

See POSIX.1-2017 article for normative description. This function is also exposed as `socketpair()` if `CONFIG_POSIX_API` is defined.

int **zsock_close**(int sock)

Close a network socket.

Close a network socket. This function is also exposed as `close()` if `CONFIG_POSIX_API` is defined (in which case it may conflict with generic POSIX `close()` function).

int **zsock_shutdown**(int sock, int how)

Shutdown socket send/receive operations.

See POSIX.1-2017 article for normative description, but currently this function has no effect in Zephyr and provided solely for compatibility with existing code. This function is also exposed as `shutdown()` if `CONFIG_POSIX_API` is defined.

int **zsock_bind**(int sock, const struct *sockaddr* *addr, *socklen_t* addrlen)

Bind a socket to a local network address.

See POSIX.1-2017 article for normative description. This function is also exposed as `bind()` if `CONFIG_POSIX_API` is defined.

int **zsock_connect**(int sock, const struct *sockaddr* *addr, *socklen_t* addrlen)

Connect a socket to a peer network address.

See POSIX.1-2017 article for normative description. This function is also exposed as `connect()` if `CONFIG_POSIX_API` is defined.

int **zsock_listen**(int sock, int backlog)

Set up a STREAM socket to accept peer connections.

See POSIX.1-2017 article for normative description. This function is also exposed as `listen()` if `CONFIG_POSIX_API` is defined.

int **zsock_accept**(int sock, struct *sockaddr* *addr, *socklen_t* *addrlen)

Accept a connection on listening socket.

See POSIX.1-2017 article for normative description. This function is also exposed as `accept()` if `CONFIG_POSIX_API` is defined.

ssize_t **zsock_sendto**(int sock, const void *buf, size_t len, int flags, const struct *sockaddr* *dest_addr, *socklen_t* addrlen)

Send data to an arbitrary network address.

See POSIX.1-2017 article for normative description. This function is also exposed as `sendto()` if `CONFIG_POSIX_API` is defined.

static inline ssize_t **zsock_send**(int sock, const void *buf, size_t len, int flags)

Send data to a connected peer.

See POSIX.1-2017 article for normative description. This function is also exposed as `send()` if `CONFIG_POSIX_API` is defined.

ssize_t **zsock_sendmsg**(int sock, const struct *msghdr* *msg, int flags)

Send data to an arbitrary network address.

See POSIX.1-2017 article for normative description. This function is also exposed as `sendmsg()` if `CONFIG_POSIX_API` is defined.

ssize_t **zsock_recvfrom**(int sock, void *buf, size_t max_len, int flags, struct *sockaddr* *src_addr, *socklen_t* *addrlen)

Receive data from an arbitrary network address.

See POSIX.1-2017 article for normative description. This function is also exposed as `recvfrom()` if `CONFIG_POSIX_API` is defined.

ssize_t **zsock_recvmsg**(int sock, struct *msghdr* *msg, int flags)

Receive a message from an arbitrary network address.

See POSIX.1-2017 article for normative description. This function is also exposed as `recvmsg()` if `CONFIG_POSIX_API` is defined.

static inline ssize_t **zsock_recv**(int sock, void *buf, size_t max_len, int flags)

Receive data from a connected peer.

See POSIX.1-2017 article for normative description. This function is also exposed as `recv()` if `CONFIG_POSIX_API` is defined.

int **zsock_fcntl_impl**(int sock, int cmd, int flags)

Control blocking/non-blocking mode of a socket.

This functions allow to (only) configure a socket for blocking or non-blocking operation (O_NONBLOCK). This function is also exposed as `fcntl()` if `CONFIG_POSIX_API` is defined (in which case it may conflict with generic POSIX `fcntl()` function).

int **zsock_ioctl_impl**(int sock, unsigned long request, va_list ap)

Control underlying socket parameters.

See POSIX.1-2017 article for normative description. This function enables querying or manipulating underlying socket parameters. Currently supported @p request values include `ZFD_IOCTL_FIONBIO`, and `ZFD_IOCTL_FIONREAD`, to set non-blocking mode, and

query the number of bytes available to read, respectively. This function is also exposed as `ioctl()` if `CONFIG_POSIX_API` is defined (in which case it may conflict with generic POSIX `ioctl()` function).

int **zsock_poll**(struct *zsock_pollfd* *fds, int nfds, int timeout)

Efficiently poll multiple sockets for events.

See POSIX.1-2017 article for normative description. This function is also exposed as `poll()` if `CONFIG_POSIX_API` is defined (in which case it may conflict with generic POSIX `poll()` function).

int **zsock_getsockopt**(int sock, int level, int optname, void *optval, *socklen_t* *optlen)

Get various socket options.

See POSIX.1-2017 article for normative description. In Zephyr this function supports a subset of socket options described by POSIX, but also some additional options available in Linux (some options are dummy and provided to ease porting of existing code). This function is also exposed as `getsockopt()` if `CONFIG_POSIX_API` is defined.

int **zsock_setsockopt**(int sock, int level, int optname, const void *optval, *socklen_t* optlen)

Set various socket options.

See POSIX.1-2017 article for normative description. In Zephyr this function supports a subset of socket options described by POSIX, but also some additional options available in Linux (some options are dummy and provided to ease porting of existing code). This function is also exposed as `setsockopt()` if `CONFIG_POSIX_API` is defined.

int **zsock_getpeername**(int sock, struct *sockaddr* *addr, *socklen_t* *addrlen)

Get peer name.

See POSIX.1-2017 article for normative description. This function is also exposed as `getpeername()` if `CONFIG_POSIX_API` is defined.

int **zsock_getsockname**(int sock, struct *sockaddr* *addr, *socklen_t* *addrlen)

Get socket name.

See POSIX.1-2017 article for normative description. This function is also exposed as `getsockname()` if `CONFIG_POSIX_API` is defined.

int **zsock_gethostname**(char *buf, size_t len)

Get local host name.

See POSIX.1-2017 article for normative description. This function is also exposed as `gethostname()` if `CONFIG_POSIX_API` is defined.

static inline char *__zsock_inet_ntop__(*sa_family_t* family, const void *src, char *dst, size_t size)

Convert network address from internal to numeric ASCII form.

See POSIX.1-2017 article for normative description. This function is also exposed as `inet_ntop()` if `CONFIG_POSIX_API` is defined.

int **zsock_inet_pton**(*sa_family_t* family, const char *src, void *dst)

Convert network address from numeric ASCII form to internal representation.

See POSIX.1-2017 article for normative description. This function is also exposed as `inet_pton()` if `CONFIG_POSIX_API` is defined.

int **zsock_getaddrinfo**(const char *host, const char *service, const struct *zsock_addrinfo* *hints, struct *zsock_addrinfo* **res)

Resolve a domain name to one or more network addresses.

See POSIX.1-2017 article for normative description. This function is also exposed as `getaddrinfo()` if `CONFIG_POSIX_API` is defined.

void **zsock_freeaddrinfo**(struct *zsock_addrinfo* *ai)

Free results returned by *zsock_getaddrinfo()*

See POSIX.1-2017 article for normative description. This function is also exposed as `freeaddrinfo()` if `CONFIG_POSIX_API` is defined.

const char ***zsock_gai_strerror**(int errcode)

Convert *zsock_getaddrinfo()* error code to textual message.

See POSIX.1-2017 article for normative description. This function is also exposed as `gai_strerror()` if `CONFIG_POSIX_API` is defined.

int **zsock_getnameinfo**(const struct *sockaddr* *addr, *socklen_t* addrlen, char *host, *socklen_t* hostlen, char *serv, *socklen_t* servlen, int flags)

Resolve a network address to a domain name or ASCII address.

See POSIX.1-2017 article for normative description. This function is also exposed as `getnameinfo()` if `CONFIG_POSIX_API` is defined.

int **zsock_select**(int nfds, *zsock_fd_set* *readfds, *zsock_fd_set* *writefds, *zsock_fd_set* *exceptfds, struct zsock_timeval *timeout)

Legacy function to poll multiple sockets for events.

See POSIX.1-2017 article for normative description. This function is provided to ease porting of existing code and not recommended for usage due to its inefficiency, use zsock_poll() instead. In Zephyr this function works only with sockets, not arbitrary file descriptors. This function is also exposed as `select()` if `CONFIG_POSIX_API` is defined (in which case it may conflict with generic POSIX `select()` function).

void **ZSOCK_FD_ZERO**(*zsock_fd_set* *set)

Initialize (clear) fd_set.

See POSIX.1-2017 article for normative description. This function is also exposed as `FD_ZERO()` if `CONFIG_POSIX_API` is defined.

int **ZSOCK_FD_ISSET**(int fd, *zsock_fd_set* *set)

Check whether socket is a member of fd_set.

See POSIX.1-2017 article for normative description. This function is also exposed as `FD_ISSET()` if `CONFIG_POSIX_API` is defined.

void **ZSOCK_FD_CLR**(int fd, *zsock_fd_set* *set)

Remove socket from fd_set.

See POSIX.1-2017 article for normative description. This function is also exposed as `FD_CLR()` if `CONFIG_POSIX_API` is defined.

void **ZSOCK_FD_SET**(int fd, *zsock_fd_set* *set)

Add socket to fd_set.

See POSIX.1-2017 article for normative description. This function is also exposed as `FD_SET()` if `CONFIG_POSIX_API` is defined.

struct **zsock_addrinfo**

#include <socket.h> Definition used when querying address information.

A linked list of these descriptors is returned by *getaddrinfo()*. The struct is also passed as hints when calling the *getaddrinfo()* function.

Public Members

struct *zsock_addrinfo* ***ai_next**
Pointer to next address entry.

int **ai_flags**
Additional options.

int **ai_family**
Address family of the returned addresses.

int **ai_socktype**
Socket type, for example SOCK_STREAM or SOCK_DGRAM.

int **ai_protocol**
Protocol for addresses, 0 means any protocol.

int **ai_eflags**
Extended flags for special usage.

socklen_t **ai_addrlen**
Length of the socket address.

struct *sockaddr* ***ai_addr**
Pointer to the address.

char ***ai_canonname**
Optional official name of the host.

struct **ifreq**
#include <*socket.h*> Interface description structure.

Public Members

char **ifr_name**[Z_DEVICE_MAX_NAME_LEN]
Network interface name.

struct **in_pktinfo**
#include <*socket.h*> Incoming IPv4 packet information.
Used as ancillary data when calling *recvmsg()* and IP_PKTINFO socket option is set.

Public Members

unsigned int **ipi_ifindex**
Network interface index.

struct *in_addr* **ipi_spec_dst**
Local address.

struct *in_addr* **ipi_addr**
Header Destination address.

struct **ip_mreqn**
#include <*socket.h*> Struct used when joining or leaving a IPv4 multicast group.

Public Members

struct *in_addr* **imr_multiaddr**
IP multicast group address.

struct *in_addr* **imr_address**
IP address of local interface.

int **imr_ifindex**
Network interface index.

struct **ipv6_mreq**
#include <*socket.h*> Struct used when joining or leaving a IPv6 multicast group.

Public Members

struct *in6_addr* **ipv6mr_multiaddr**
IPv6 multicast address of group.

int **ipv6mr_ifindex**
Network interface index of the local IPv6 address.

struct **in6_pktinfo**
#include <*socket.h*> Incoming IPv6 packet information.

Used as ancillary data when calling *recvmsg()* and IPV6_RECVPKTINFO socket option is set.

Public Members

struct *in6_addr* **ipi6_addr**
Destination IPv6 address.

unsigned int **ipi6_ifindex**
Receive interface index.

struct **zsock_pollfd**
#include <*socket_poll.h*> Definition of the monitored socket/file descriptor.

An array of these descriptors is passed as an argument to *poll()*.

Public Members

int `fd`
Socket descriptor.

short `events`
Requested events.

short `revents`
Returned events.

struct `zsock_fd_set`
#include <*socket_select.h*> Socket file descriptor set.

Related code samples

AWS IoT Core MQTT
Connect to AWS IoT Core and publish messages using MQTT.

Dumb HTTP server (multi-threaded)
Implement a simple HTTP server supporting simultaneous connections using BSD sockets.

Echo client (advanced)
Implement a client that sends IP packets, waits for data to be sent back, and verifies it.

Echo server (advanced)
Implement a UDP/TCP server that sends received packets back to the sender.

HTTP Client
Implement an HTTP(S) client that issues a variety of HTTP requests.

HTTP GET using plain sockets
Implement an HTTP(S) client using plain BSD sockets.

HTTP Server
Implement an HTTP(s) Server demonstrating various resource types.

Large HTTP download
Download a large file from a web server using BSD sockets.

Microsoft Azure IoT Hub MQTT
Connect to Azure IoT Hub and publish messages using MQTT.

TagoIO HTTP Post
Send random temperature values to TagoIO IoT Cloud Platform using HTTP.

TLS Credentials

group `tls_credentials`
TLS credentials management.

Typedefs

typedef int **sec_tag_t**

Secure tag, a reference to TLS credential.

Secure tag can be used to reference credential after it was registered in the system.

> **Note**
>
> Some TLS credentials come in pairs:
>
> - TLS_CREDENTIAL_SERVER_CERTIFICATE with TLS_CREDENTIAL_PRIVATE_KEY,
> - TLS_CREDENTIAL_PSK with TLS_CREDENTIAL_PSK_ID. Such pairs of credentials must be assigned the same secure tag to be correctly handled in the system.

> **Note**
>
> Negative values are reserved for internal use.

Enums

enum **tls_credential_type**

TLS credential types.

Values:

enumerator **TLS_CREDENTIAL_NONE**

Unspecified credential.

enumerator **TLS_CREDENTIAL_CA_CERTIFICATE**

A trusted CA certificate.

Use this to authenticate remote servers. Used with certificate-based ciphersuites.

enumerator **TLS_CREDENTIAL_SERVER_CERTIFICATE**

A public server certificate.

Use this to register your own server certificate. Should be registered together with a corresponding private key. Used with certificate-based ciphersuites.

enumerator **TLS_CREDENTIAL_PRIVATE_KEY**

Private key.

Should be registered together with a corresponding public certificate. Used with certificate-based ciphersuites.

enumerator **TLS_CREDENTIAL_PSK**

Pre-shared key.

Should be registered together with a corresponding PSK identity. Used with PSK-based ciphersuites.

enumerator TLS_CREDENTIAL_PSK_ID

Pre-shared key identity.

Should be registered together with a corresponding PSK. Used with PSK-based ciphersuites.

Functions

int **tls_credential_add**(*sec_tag_t* tag, enum *tls_credential_type* type, const void *cred, size_t credlen)

Add a TLS credential.

This function adds a TLS credential, that can be used by TLS/DTLS for authentication.

Parameters

- **tag** – A security tag that credential will be referenced with.
- **type** – A TLS/DTLS credential type.
- **cred** – A TLS/DTLS credential.
- **credlen** – A TLS/DTLS credential length.

Return values

- **0** – TLS credential successfully added.
- **-EACCES** – Access to the TLS credential subsystem was denied.
- **-ENOMEM** – Not enough memory to add new TLS credential.
- **-EEXIST** – TLS credential of specific tag and type already exists.

int **tls_credential_get**(*sec_tag_t* tag, enum *tls_credential_type* type, void *cred, size_t *credlen)

Get a TLS credential.

This function gets an already registered TLS credential, referenced by tag secure tag of type.

Parameters

- **tag** – A security tag of requested credential.
- **type** – A TLS/DTLS credential type of requested credential.
- **cred** – A buffer for TLS/DTLS credential.
- **credlen** – A buffer size on input. TLS/DTLS credential length on output.

Return values

- **0** – TLS credential successfully obtained.
- **-EACCES** – Access to the TLS credential subsystem was denied.
- **-ENOENT** – Requested TLS credential was not found.
- **-EFBIG** – Requested TLS credential does not fit in the buffer provided.

int **tls_credential_delete**(*sec_tag_t* tag, enum *tls_credential_type* type)

Delete a TLS credential.

This function removes a TLS credential, referenced by tag secure tag of type.

Parameters

- **tag** – A security tag corresponding to removed credential.

- `type` – A TLS/DTLS credential type of removed credential.

Return values

- `0` – TLS credential successfully deleted.
- `-EACCES` – Access to the TLS credential subsystem was denied.
- `-ENOENT` – Requested TLS credential was not found.

IPv4/IPv6 Primitives and Helpers

- *Overview*
- *API Reference*

Overview Miscellaneous defines and helper functions for IP addresses and IP protocols.

API Reference

group `ip_4_6`

IPv4/IPv6 primitives and helpers.

Defines

`PF_UNSPEC`

Unspecified protocol family.

`PF_INET`

IP protocol family version 4.

`PF_INET6`

IP protocol family version 6.

`PF_PACKET`

Packet family.

`PF_CAN`

Controller Area Network.

`PF_NET_MGMT`

Network management info.

`PF_LOCAL`

Inter-process communication

`PF_UNIX`

Inter-process communication

AF_UNSPEC
: Unspecified address family.

AF_INET
: IP protocol family version 4.

AF_INET6
: IP protocol family version 6.

AF_PACKET
: Packet family.

AF_CAN
: Controller Area Network.

AF_NET_MGMT
: Network management info.

AF_LOCAL
: Inter-process communication

AF_UNIX
: Inter-process communication

ntohs(x)
: Convert 16-bit value from network to host byte order.

  Parameters

  - x – The network byte order value to convert.

  Returns
  Host byte order value.

ntohl(x)
: Convert 32-bit value from network to host byte order.

  Parameters

  - x – The network byte order value to convert.

  Returns
  Host byte order value.

ntohll(x)
: Convert 64-bit value from network to host byte order.

  Parameters

  - x – The network byte order value to convert.

  Returns
  Host byte order value.

htons(x)
: Convert 16-bit value from host to network byte order.

  Parameters

  - x – The host byte order value to convert.

Returns
Network byte order value.

`htonl`(x)
Convert 32-bit value from host to network byte order.

Parameters

- `x` – The host byte order value to convert.

Returns
Network byte order value.

`htonll`(x)
Convert 64-bit value from host to network byte order.

Parameters

- `x` – The host byte order value to convert.

Returns
Network byte order value.

`NET_IPV6_ADDR_SIZE`
Binary size of the IPv6 address.

`NET_IPV4_ADDR_SIZE`
Binary size of the IPv4 address.

`CMSG_FIRSTHDR`(msghdr)
Returns a pointer to the first cmsghdr in the ancillary data buffer associated with the passed msghdr.

It returns NULL if there isn't enough space for a cmsghdr in the buffer.

`CMSG_NXTHDR`(msghdr, cmsg)
Returns the next valid cmsghdr after the passed cmsghdr.

It returns NULL when there isn't enough space left in the buffer.

`CMSG_DATA`(cmsg)
Returns a pointer to the data portion of a cmsghdr.

The pointer returned cannot be assumed to be suitably aligned for accessing arbitrary payload data types. Applications should not cast it to a pointer type matching the payload, but should instead use memcpy(3) to copy data to or from a suitably declared object.

`CMSG_SPACE`(length)
Returns the number of bytes an ancillary element with payload of the passed data length occupies.

`CMSG_LEN`(length)
Returns the value to store in the cmsg_len member of the cmsghdr structure, taking into account any necessary alignment.

It takes the data length as an argument.

`IN6ADDR_ANY_INIT`
IPv6 address initializer.

`IN6ADDR_LOOPBACK_INIT`
IPv6 loopback address initializer.

INADDR_ANY
: IPv4 any address.

INADDR_ANY_INIT
: IPv4 address initializer.

INADDR_LOOPBACK_INIT
: IPv6 loopback address initializer.

INET_ADDRSTRLEN
: Max length of the IPv4 address as a string.

  Defined by POSIX.

INET6_ADDRSTRLEN
: Max length of the IPv6 address as a string.

  Takes into account possible mapped IPv4 addresses.

NET_MAX_PRIORITIES
: How many priority values there are.

net_ipaddr_copy(dest, src)
: Copy an IPv4 or IPv6 address.

  Parameters

  - dest – Destination IP address.
  - src – Source IP address.

  Returns
  Destination address.

Typedefs

typedef unsigned short int sa_family_t
: Socket address family type.

typedef size_t socklen_t
: Length of a socket address.

Enums

enum net_ip_protocol
: Protocol numbers from IANA/BSD.

  Values:

  enumerator IPPROTO_IP = 0
  : IP protocol (pseudo-val for *setsockopt()*

enumerator IPPROTO_ICMP = 1
ICMP protocol

enumerator IPPROTO_IGMP = 2
IGMP protocol

enumerator IPPROTO_IPIP = 4
IPIP tunnels

enumerator IPPROTO_TCP = 6
TCP protocol

enumerator IPPROTO_UDP = 17
UDP protocol

enumerator IPPROTO_IPV6 = 41
IPv6 protocol

enumerator IPPROTO_ICMPV6 = 58
ICMPv6 protocol.

enumerator IPPROTO_RAW = 255
RAW IP packets

enum net_ip_protocol_secure
Protocol numbers for TLS protocols.
Values:

enumerator IPPROTO_TLS_1_0 = 256
TLS 1.0 protocol.

enumerator IPPROTO_TLS_1_1 = 257
TLS 1.1 protocol.

enumerator IPPROTO_TLS_1_2 = 258
TLS 1.2 protocol.

enumerator IPPROTO_DTLS_1_0 = 272
DTLS 1.0 protocol.

enumerator IPPROTO_DTLS_1_2 = 273
DTLS 1.2 protocol.

enum net_sock_type
Socket type.
Values:

enumerator SOCK_STREAM = 1
Stream socket type

enumerator SOCK_DGRAM
Datagram socket type.

enumerator SOCK_RAW
RAW socket type

enum net_ip_mtu
IP Maximum Transfer Unit.

Values:

enumerator NET_IPV6_MTU = 1280
IPv6 MTU length.

We must be able to receive this size IPv6 packet without fragmentation.

enumerator NET_IPV4_MTU = 576
IPv4 MTU length.

We must be able to receive this size IPv4 packet without fragmentation.

enum net_priority
Network packet priority settings described in IEEE 802.1Q Annex I.1.

Values:

enumerator NET_PRIORITY_BK = 1
Background (lowest)

enumerator NET_PRIORITY_BE = 0
Best effort (default)

enumerator NET_PRIORITY_EE = 2
Excellent effort

enumerator NET_PRIORITY_CA = 3
Critical applications

enumerator NET_PRIORITY_VI = 4
Video, < 100 ms latency and jitter.

enumerator NET_PRIORITY_VO = 5
Voice, < 10 ms latency and jitter

enumerator NET_PRIORITY_IC = 6
Internetwork control

enumerator NET_PRIORITY_NC = 7
Network control (highest)

enum net_addr_state

What is the current state of the network address.

Values:

enumerator NET_ADDR_ANY_STATE = -1

Default (invalid) address type.

enumerator NET_ADDR_TENTATIVE = 0

Tentative address

enumerator NET_ADDR_PREFERRED

Preferred address

enumerator NET_ADDR_DEPRECATED

Deprecated address

enum net_addr_type

How the network address is assigned to network interface.

Values:

enumerator NET_ADDR_ANY = 0

Default value.

This is not a valid value.

enumerator NET_ADDR_AUTOCONF

Auto configured address.

enumerator NET_ADDR_DHCP

Address is from DHCP.

enumerator NET_ADDR_MANUAL

Manually set address.

enumerator NET_ADDR_OVERRIDABLE

Manually set address which is overridable by DHCP.

Functions

static inline bool net_ipv6_is_addr_loopback(struct *in6_addr* *addr)

Check if the IPv6 address is a loopback address (::1).

Parameters

- addr – IPv6 address

Returns

True if address is a loopback address, False otherwise.

static inline bool **net_ipv6_is_addr_mcast**(const struct *in6_addr* *addr)

Check if the IPv6 address is a multicast address.

Parameters

- **addr** – IPv6 address

Returns

True if address is multicast address, False otherwise.

struct *net_if_addr* ***net_if_ipv6_addr_lookup**(const struct *in6_addr* *addr, struct *net_if* **iface)

static inline bool **net_ipv6_is_my_addr**(struct *in6_addr* *addr)

Check if IPv6 address is found in one of the network interfaces.

Parameters

- **addr** – IPv6 address

Returns

True if address was found, False otherwise.

struct *net_if_mcast_addr* ***net_if_ipv6_maddr_lookup**(const struct *in6_addr* *addr, struct *net_if* **iface)

static inline bool **net_ipv6_is_my_maddr**(struct *in6_addr* *maddr)

Check if IPv6 multicast address is found in one of the network interfaces.

Parameters

- **maddr** – Multicast IPv6 address

Returns

True if address was found, False otherwise.

static inline bool **net_ipv6_is_prefix**(const uint8_t *addr1, const uint8_t *addr2, uint8_t length)

Check if two IPv6 addresses are same when compared after prefix mask.

Parameters

- **addr1** – First IPv6 address.
- **addr2** – Second IPv6 address.
- **length** – Prefix length (max length is 128).

Returns

True if IPv6 prefixes are the same, False otherwise.

static inline bool **net_ipv4_is_addr_loopback**(struct *in_addr* *addr)

Check if the IPv4 address is a loopback address (127.0.0.0/8).

Parameters

- **addr** – IPv4 address

Returns

True if address is a loopback address, False otherwise.

static inline bool **net_ipv4_is_addr_unspecified**(const struct *in_addr* *addr)

Check if the IPv4 address is unspecified (all bits zero)

Parameters

- **addr** – IPv4 address.

Returns

True if the address is unspecified, false otherwise.

static inline bool **net_ipv4_is_addr_mcast**(const struct *in_addr* *addr)

Check if the IPv4 address is a multicast address.

Parameters

- **addr** – IPv4 address

Returns

True if address is multicast address, False otherwise.

static inline bool **net_ipv4_is_ll_addr**(const struct *in_addr* *addr)

Check if the given IPv4 address is a link local address.

Parameters

- **addr** – A valid pointer on an IPv4 address

Returns

True if it is, false otherwise.

static inline bool **net_ipv4_is_private_addr**(const struct *in_addr* *addr)

Check if the given IPv4 address is from a private address range.

See https://en.wikipedia.org/wiki/Reserved_IP_addresses for details.

Parameters

- **addr** – A valid pointer on an IPv4 address

Returns

True if it is, false otherwise.

static inline void **net_ipv4_addr_copy_raw**(uint8_t *dest, const uint8_t *src)

Copy an IPv4 address raw buffer.

Parameters

- **dest** – Destination IP address.
- **src** – Source IP address.

static inline void **net_ipv6_addr_copy_raw**(uint8_t *dest, const uint8_t *src)

Copy an IPv6 address raw buffer.

Parameters

- **dest** – Destination IP address.
- **src** – Source IP address.

static inline bool **net_ipv4_addr_cmp**(const struct *in_addr* *addr1, const struct *in_addr* *addr2)

Compare two IPv4 addresses.

Parameters

- **addr1** – Pointer to IPv4 address.
- **addr2** – Pointer to IPv4 address.

Returns

True if the addresses are the same, false otherwise.

static inline bool **net_ipv4_addr_cmp_raw**(const uint8_t *addr1, const uint8_t *addr2)

Compare two raw IPv4 address buffers.

Parameters

- **addr1** – Pointer to IPv4 address buffer.
- **addr2** – Pointer to IPv4 address buffer.

Returns
True if the addresses are the same, false otherwise.

static inline bool **net_ipv6_addr_cmp**(const struct *in6_addr* *addr1, const struct *in6_addr* *addr2)

Compare two IPv6 addresses.

Parameters

- **addr1** – Pointer to IPv6 address.
- **addr2** – Pointer to IPv6 address.

Returns
True if the addresses are the same, false otherwise.

static inline bool **net_ipv6_addr_cmp_raw**(const uint8_t *addr1, const uint8_t *addr2)

Compare two raw IPv6 address buffers.

Parameters

- **addr1** – Pointer to IPv6 address buffer.
- **addr2** – Pointer to IPv6 address buffer.

Returns
True if the addresses are the same, false otherwise.

static inline bool **net_ipv6_is_ll_addr**(const struct *in6_addr* *addr)

Check if the given IPv6 address is a link local address.

Parameters

- **addr** – A valid pointer on an IPv6 address

Returns
True if it is, false otherwise.

static inline bool **net_ipv6_is_sl_addr**(const struct *in6_addr* *addr)

Check if the given IPv6 address is a site local address.

Parameters

- **addr** – A valid pointer on an IPv6 address

Returns
True if it is, false otherwise.

static inline bool **net_ipv6_is_ula_addr**(const struct *in6_addr* *addr)

Check if the given IPv6 address is a unique local address.

Parameters

- **addr** – A valid pointer on an IPv6 address

Returns
True if it is, false otherwise.

static inline bool **net_ipv6_is_global_addr**(const struct *in6_addr* *addr)

Check if the given IPv6 address is a global address.

Parameters

- **addr** – A valid pointer on an IPv6 address

Returns
True if it is, false otherwise.

static inline bool **net_ipv6_is_private_addr**(const struct *in6_addr* *addr)

Check if the given IPv6 address is from a private/local address range.

See https://en.wikipedia.org/wiki/Reserved_IP_addresses for details.

Parameters

- **addr** – A valid pointer on an IPv6 address

Returns

True if it is, false otherwise.

const struct *in6_addr* ***net_ipv6_unspecified_address**(void)

Return pointer to any (all bits zeros) IPv6 address.

Returns

Any IPv6 address.

const struct *in_addr* ***net_ipv4_unspecified_address**(void)

Return pointer to any (all bits zeros) IPv4 address.

Returns

Any IPv4 address.

const struct *in_addr* ***net_ipv4_broadcast_address**(void)

Return pointer to broadcast (all bits ones) IPv4 address.

Returns

Broadcast IPv4 address.

bool **net_if_ipv4_addr_mask_cmp**(struct *net_if* *iface, const struct *in_addr* *addr)

static inline bool **net_ipv4_addr_mask_cmp**(struct *net_if* *iface, const struct *in_addr* *addr)

Check if the given address belongs to same subnet that has been configured for the interface.

Parameters

- **iface** – A valid pointer on an interface
- **addr** – IPv4 address

Returns

True if address is in same subnet, false otherwise.

bool **net_if_ipv4_is_addr_bcast**(struct *net_if* *iface, const struct *in_addr* *addr)

static inline bool **net_ipv4_is_addr_bcast**(struct *net_if* *iface, const struct *in_addr* *addr)

Check if the given IPv4 address is a broadcast address.

Parameters

- **iface** – Interface to use. Must be a valid pointer to an interface.
- **addr** – IPv4 address

Returns

True if address is a broadcast address, false otherwise.

struct *net_if_addr* ***net_if_ipv4_addr_lookup**(const struct *in_addr* *addr, struct *net_if* **iface)

static inline bool **net_ipv4_is_my_addr**(const struct *in_addr* *addr)

Check if the IPv4 address is assigned to any network interface in the system.

Parameters

- **addr** – A valid pointer on an IPv4 address

Returns
True if IPv4 address is found in one of the network interfaces, False otherwise.

static inline bool **net_ipv6_is_addr_unspecified**(const struct *in6_addr* *addr)

Check if the IPv6 address is unspecified (all bits zero)

Parameters

- **addr** – IPv6 address.

Returns
True if the address is unspecified, false otherwise.

static inline bool **net_ipv6_is_addr_solicited_node**(const struct *in6_addr* *addr)

Check if the IPv6 address is solicited node multicast address FF02:0:0:0:0:1:FFXX:XXXX defined in RFC 3513.

Parameters

- **addr** – IPv6 address.

Returns
True if the address is solicited node address, false otherwise.

static inline bool **net_ipv6_is_addr_mcast_scope**(const struct *in6_addr* *addr, int scope)

Check if the IPv6 address is a given scope multicast address (FFyx::).

Parameters

- **addr** – IPv6 address
- **scope** – Scope to check

Returns
True if the address is in given scope multicast address, false otherwise.

static inline bool **net_ipv6_is_same_mcast_scope**(const struct *in6_addr* *addr_1, const struct *in6_addr* *addr_2)

Check if the IPv6 addresses have the same multicast scope (FFyx::).

Parameters

- **addr_1** – IPv6 address 1
- **addr_2** – IPv6 address 2

Returns
True if both addresses have same multicast scope, false otherwise.

static inline bool **net_ipv6_is_addr_mcast_global**(const struct *in6_addr* *addr)

Check if the IPv6 address is a global multicast address (FFxE::/16).

Parameters

- **addr** – IPv6 address.

Returns
True if the address is global multicast address, false otherwise.

static inline bool **net_ipv6_is_addr_mcast_iface**(const struct *in6_addr* *addr)

Check if the IPv6 address is a interface scope multicast address (FFx1::).

Parameters

- **addr** – IPv6 address.

Returns
True if the address is a interface scope multicast address, false otherwise.

static inline bool **net_ipv6_is_addr_mcast_link**(const struct *in6_addr* *addr)

Check if the IPv6 address is a link local scope multicast address (FFx2::).

Parameters

- **addr** – IPv6 address.

Returns

True if the address is a link local scope multicast address, false otherwise.

static inline bool **net_ipv6_is_addr_mcast_mesh**(const struct *in6_addr* *addr)

Check if the IPv6 address is a mesh-local scope multicast address (FFx3::).

Parameters

- **addr** – IPv6 address.

Returns

True if the address is a mesh-local scope multicast address, false otherwise.

static inline bool **net_ipv6_is_addr_mcast_site**(const struct *in6_addr* *addr)

Check if the IPv6 address is a site scope multicast address (FFx5::).

Parameters

- **addr** – IPv6 address.

Returns

True if the address is a site scope multicast address, false otherwise.

static inline bool **net_ipv6_is_addr_mcast_org**(const struct *in6_addr* *addr)

Check if the IPv6 address is an organization scope multicast address (FFx8::).

Parameters

- **addr** – IPv6 address.

Returns

True if the address is an organization scope multicast address, false otherwise.

static inline bool **net_ipv6_is_addr_mcast_group**(const struct *in6_addr* *addr, const struct *in6_addr* *group)

Check if the IPv6 address belongs to certain multicast group.

Parameters

- **addr** – IPv6 address.
- **group** – Group id IPv6 address, the values must be in network byte order

Returns

True if the IPv6 multicast address belongs to given multicast group, false otherwise.

static inline bool **net_ipv6_is_addr_mcast_all_nodes_group**(const struct *in6_addr* *addr)

Check if the IPv6 address belongs to the all nodes multicast group.

Parameters

- **addr** – IPv6 address

Returns

True if the IPv6 multicast address belongs to the all nodes multicast group, false otherwise

static inline bool **net_ipv6_is_addr_mcast_iface_all_nodes**(const struct *in6_addr* *addr)

Check if the IPv6 address is a interface scope all nodes multicast address (FF01::1).

Parameters

- **addr** – IPv6 address.

Returns
True if the address is a interface scope all nodes multicast address, false otherwise.

static inline bool **net_ipv6_is_addr_mcast_link_all_nodes**(const struct *in6_addr* *addr)

Check if the IPv6 address is a link local scope all nodes multicast address (FF02::1).

Parameters

- **addr** – IPv6 address.

Returns
True if the address is a link local scope all nodes multicast address, false otherwise.

static inline void **net_ipv6_addr_create_solicited_node**(const struct *in6_addr* *src, struct *in6_addr* *dst)

Create solicited node IPv6 multicast address FF02:0:0:0:0:1:FFXX:XXXX defined in RFC 3513.

Parameters

- **src** – IPv6 address.
- **dst** – IPv6 address.

static inline void **net_ipv6_addr_create**(struct *in6_addr* *addr, uint16_t addr0, uint16_t addr1, uint16_t addr2, uint16_t addr3, uint16_t addr4, uint16_t addr5, uint16_t addr6, uint16_t addr7)

Construct an IPv6 address from eight 16-bit words.

Parameters

- **addr** – IPv6 address
- **addr0** – 16-bit word which is part of the address
- **addr1** – 16-bit word which is part of the address
- **addr2** – 16-bit word which is part of the address
- **addr3** – 16-bit word which is part of the address
- **addr4** – 16-bit word which is part of the address
- **addr5** – 16-bit word which is part of the address
- **addr6** – 16-bit word which is part of the address
- **addr7** – 16-bit word which is part of the address

static inline void **net_ipv6_addr_create_ll_allnodes_mcast**(struct *in6_addr* *addr)

Create link local allnodes multicast IPv6 address.

Parameters

- **addr** – IPv6 address

static inline void **net_ipv6_addr_create_ll_allrouters_mcast**(struct *in6_addr* *addr)

Create link local allrouters multicast IPv6 address.

Parameters

- **addr** – IPv6 address

static inline void **net_ipv6_addr_create_v4_mapped**(const struct *in_addr* *addr4, struct *in6_addr* *addr6)

Create IPv4 mapped IPv6 address.

Parameters

- **addr4** – IPv4 address
- **addr6** – IPv6 address to be created

static inline bool **net_ipv6_addr_is_v4_mapped**(const struct *in6_addr* *addr)

Is the IPv6 address an IPv4 mapped one.

The v4 mapped addresses look like ::ffff:a.b.c.d

Parameters

- **addr** – IPv6 address

Returns

True if IPv6 address is a IPv4 mapped address, False otherwise.

static inline void **net_ipv6_addr_create_iid**(struct *in6_addr* *addr, struct *net_linkaddr* *lladdr)

Create IPv6 address interface identifier.

Parameters

- **addr** – IPv6 address
- **lladdr** – Link local address

static inline bool **net_ipv6_addr_based_on_ll**(const struct *in6_addr* *addr, const struct *net_linkaddr* *lladdr)

Check if given address is based on link layer address.

Returns

True if it is, False otherwise

static inline struct *sockaddr_in6* ***net_sin6**(const struct *sockaddr* *addr)

Get *sockaddr_in6* from sockaddr.

This is a helper so that the code calling this function can be made shorter.

Parameters

- **addr** – Socket address

Returns

Pointer to IPv6 socket address

static inline struct *sockaddr_in* ***net_sin**(const struct *sockaddr* *addr)

Get *sockaddr_in* from sockaddr.

This is a helper so that the code calling this function can be made shorter.

Parameters

- **addr** – Socket address

Returns

Pointer to IPv4 socket address

static inline struct sockaddr_in6_ptr *net_sin6_ptr(const struct sockaddr_ptr *addr)

Get sockaddr_in6_ptr from sockaddr_ptr.

This is a helper so that the code calling this function can be made shorter.

Parameters

- addr – Socket address

Returns

Pointer to IPv6 socket address

static inline struct sockaddr_in_ptr *net_sin_ptr(const struct sockaddr_ptr *addr)

Get sockaddr_in_ptr from sockaddr_ptr.

This is a helper so that the code calling this function can be made shorter.

Parameters

- addr – Socket address

Returns

Pointer to IPv4 socket address

static inline struct sockaddr_ll_ptr *net_sll_ptr(const struct sockaddr_ptr *addr)

Get sockaddr_ll_ptr from sockaddr_ptr.

This is a helper so that the code calling this function can be made shorter.

Parameters

- addr – Socket address

Returns

Pointer to linklayer socket address

static inline struct sockaddr_can_ptr *net_can_ptr(const struct sockaddr_ptr *addr)

Get sockaddr_can_ptr from sockaddr_ptr.

This is a helper so that the code needing this functionality can be made shorter.

Parameters

- addr – Socket address

Returns

Pointer to CAN socket address

int net_addr_pton(*sa_family_t* family, const char *src, void *dst)

Convert a string to IP address.

Note

This function doesn't do precise error checking, do not use for untrusted strings.

Parameters

- family – IP address family (AF_INET or AF_INET6)
- src – IP address in a null terminated string
- dst – Pointer to struct *in_addr* if family is AF_INET or pointer to struct *in6_addr* if family is AF_INET6

Returns

0 if ok, < 0 if error

char *net_addr_ntop(*sa_family_t* family, const void *src, char *dst, size_t size)

Convert IP address to string form.

Parameters

- family – IP address family (AF_INET or AF_INET6)
- src – Pointer to struct *in_addr* if family is AF_INET or pointer to struct *in6_addr* if family is AF_INET6
- dst – Buffer for IP address as a null terminated string
- size – Number of bytes available in the buffer

Returns

dst pointer if ok, NULL if error

bool net_ipaddr_parse(const char *str, size_t str_len, struct *sockaddr* *addr)

Parse a string that contains either IPv4 or IPv6 address and optional port, and store the information in user supplied sockaddr struct.

Syntax of the IP address string: 192.0.2.1:80 192.0.2.42

[2001:db8::2] 2001:db::42 Note that the str_len parameter is used to restrict the amount of characters that are checked. If the string does not contain port number, then the port number in sockaddr is not modified.

Parameters

- str – String that contains the IP address.
- str_len – Length of the string to be parsed.
- addr – Pointer to user supplied struct sockaddr.

Returns

True if parsing could be done, false otherwise.

int net_port_set_default(struct *sockaddr* *addr, uint16_t default_port)

Set the default port in the sockaddr structure.

If the port is already set, then do nothing.

Parameters

- addr – Pointer to user supplied struct sockaddr.
- default_port – Default port number to set.

Returns

0 if ok, <0 if error

static inline int32_t net_tcp_seq_cmp(uint32_t seq1, uint32_t seq2)

Compare TCP sequence numbers.

This function compares TCP sequence numbers, accounting for wraparound effects.

Parameters

- seq1 – First sequence number
- seq2 – Seconds sequence number

Returns

< 0 if seq1 < seq2, 0 if seq1 == seq2, > 0 if seq > seq2

static inline bool net_tcp_seq_greater(uint32_t seq1, uint32_t seq2)

Check that one TCP sequence number is greater.

This is convenience function on top of *net_tcp_seq_cmp()*.

Parameters

- `seq1` – First sequence number
- `seq2` – Seconds sequence number

Returns
True if seq > seq2

int `net_bytes_from_str`(uint8_t *buf, int buf_len, const char *src)

Convert a string of hex values to array of bytes.

The syntax of the string is “ab:02:98:fa:42:01”

Parameters

- `buf` – Pointer to memory where the bytes are written.
- `buf_len` – Length of the memory area.
- `src` – String of bytes.

Returns
0 if ok, <0 if error

int `net_tx_priority2tc`(enum *net_priority* prio)

Convert Tx network packet priority to traffic class so we can place the packet into correct Tx queue.

Parameters

- `prio` – Network priority

Returns
Tx traffic class that handles that priority network traffic.

int `net_rx_priority2tc`(enum *net_priority* prio)

Convert Rx network packet priority to traffic class so we can place the packet into correct Rx queue.

Parameters

- `prio` – Network priority

Returns
Rx traffic class that handles that priority network traffic.

static inline enum *net_priority* `net_vlan2priority`(uint8_t priority)

Convert network packet VLAN priority to network packet priority so we can place the packet into correct queue.

Parameters

- `priority` – VLAN priority

Returns
Network priority

static inline uint8_t `net_priority2vlan`(enum *net_priority* priority)

Convert network packet priority to network packet VLAN priority.

Parameters

- `priority` – Packet priority

Returns
VLAN priority (PCP)

const char *net_family2str(*sa_family_t* family)

Return network address family value as a string.

This is only usable for debugging.

Parameters

- family – Network address family code

Returns

Network address family as a string, or NULL if family is unknown.

static inline int net_ipv6_pe_add_filter(struct *in6_addr* *addr, bool is_denylist)

Add IPv6 prefix as a privacy extension filter.

Note that the filters can either allow or deny listing.

Parameters

- addr – IPv6 prefix
- is_denylist – Tells if this filter is for allowing or denying listing.

Returns

0 if ok, <0 if error

static inline int net_ipv6_pe_del_filter(struct *in6_addr* *addr)

Delete IPv6 prefix from privacy extension filter list.

Parameters

- addr – IPv6 prefix

Returns

0 if ok, <0 if error

struct in6_addr

#include <net_ip.h> IPv6 address struct.

Public Members

uint8_t s6_addr[16]

IPv6 address buffer.

uint16_t s6_addr16[8]

In big endian.

uint32_t s6_addr32[4]

In big endian.

struct in_addr

#include <net_ip.h> IPv4 address struct.

Public Members

uint8_t s4_addr[4]

IPv4 address buffer.

uint16_t **s4_addr16**[2]
In big endian.

uint32_t **s4_addr32**[1]
In big endian.

uint32_t **s_addr**
In big endian, for POSIX compatibility.

struct **sockaddr_in6**
#include <*net_ip.h*> Socket address struct for IPv6.

Public Members

sa_family_t **sin6_family**
AF_INET6

uint16_t **sin6_port**
Port number

struct *in6_addr* **sin6_addr**
IPv6 address

uint8_t **sin6_scope_id**
Interfaces for a scope.

struct **sockaddr_in**
#include <*net_ip.h*> Socket address struct for IPv4.

Public Members

sa_family_t **sin_family**
AF_INET

uint16_t **sin_port**
Port number

struct *in_addr* **sin_addr**
IPv4 address.

struct **sockaddr_ll**
#include <*net_ip.h*> Socket address struct for packet socket.

Public Members

sa_family_t **sll_family**
Always AF_PACKET

uint16_t **sll_protocol**
Physical-layer protocol

int **sll_ifindex**
Interface number

uint16_t **sll_hatype**
ARP hardware type

uint8_t **sll_pkttype**
Packet type

uint8_t **sll_halen**
Length of address

uint8_t **sll_addr**[8]
Physical-layer address, big endian.

struct **iovec**
#include <net_ip.h> IO vector array element.

Public Members

void ***iov_base**
Pointer to data.

size_t **iov_len**
Length of the data.

struct **msghdr**
#include <net_ip.h> Message struct.

Public Members

void ***msg_name**
Optional socket address, big endian.

socklen_t **msg_namelen**
Size of socket address.

struct *iovec* ***msg_iov**
Scatter/gather array.

size_t **msg_iovlen**
Number of elements in msg_iov.

void ***msg_control**
Ancillary data.

size_t **msg_controllen**
Ancillary data buffer len.

int **msg_flags**
Flags on received message.

struct **cmsghdr**
#include <net_ip.h> Control message ancillary data.

Public Members

socklen_t **cmsg_len**
Number of bytes, including header.

int **cmsg_level**
Originating protocol.

int **cmsg_type**
Protocol-specific type.

z_max_align_t **cmsg_data**[]
Flexible array member to force alignment of cmsghdr.

struct **sockaddr**
#include <net_ip.h> Generic sockaddr struct.

Must be cast to proper type.

Public Members

sa_family_t **sa_family**
Address family.

struct **net_tuple**
#include <net_ip.h> IPv6/IPv4 network connection tuple.

Public Members

struct net_addr ***remote_addr**
IPv6/IPv4 remote address.

struct net_addr *`local_addr`
: IPv6/IPv4 local address

uint16_t `remote_port`
: UDP/TCP remote port

uint16_t `local_port`
: UDP/TCP local port

enum *net_ip_protocol* `ip_proto`
: IP protocol

DNS Resolve

- *Overview*
- *Sample usage*
- *API Reference*

Overview The DNS resolver implements a basic DNS resolver according to IETF RFC1035 on Domain Implementation and Specification. Supported DNS answers are IPv4/IPv6 addresses and CNAME.

If a CNAME is received, the DNS resolver will create another DNS query. The number of additional queries is controlled by the `CONFIG_DNS_RESOLVER_ADDITIONAL_QUERIES` Kconfig variable.

The multicast DNS (mDNS) client resolver support can be enabled by setting `CONFIG_MDNS_RESOLVER` Kconfig option. See IETF RFC6762 for more details about mDNS.

The link-local multicast name resolution (LLMNR) client resolver support can be enabled by setting the `CONFIG_LLMNR_RESOLVER` Kconfig variable. See IETF RFC4795 for more details about LLMNR.

For more information about DNS configuration variables, see: subsys/net/lib/dns/Kconfig. The DNS resolver API can be found at include/zephyr/net/dns_resolve.h.

Sample usage See dns-resolve sample application for details.

Related code samples

AWS IoT Core MQTT
: Connect to AWS IoT Core and publish messages using MQTT.

DNS resolve
: Resolve an IP address for a given hostname.

TagoIO HTTP Post
: Send random temperature values to TagoIO IoT Cloud Platform using HTTP.

API Reference

group `dns_resolve`

DNS resolving library.

Defines

`DNS_MAX_NAME_SIZE`

Max size of the resolved name.

Typedefs

typedef void (*`dns_resolve_cb_t`)(enum *dns_resolve_status* status, struct *dns_addrinfo* *info, void *user_data)

DNS resolve callback.

The DNS resolve callback is called after a successful DNS resolving. The resolver can call this callback multiple times, one for each resolved address.

Param status
The status of the query: DNS_EAI_INPROGRESS returned for each resolved address DNS_EAI_ALLDONE mark end of the resolving, info is set to NULL in this case DNS_EAI_CANCELED if the query was canceled manually or timeout happened DNS_EAI_FAIL if the name cannot be resolved by the server DNS_EAI_NODATA if there is no such name other values means that an error happened.

Param info
Query results are stored here.

Param user_data
The user data given in *dns_resolve_name()* call.

Enums

enum `dns_query_type`

DNS query type enum.

Values:

enumerator `DNS_QUERY_TYPE_A` = 1

IPv4 query.

enumerator `DNS_QUERY_TYPE_AAAA` = 28

IPv6 query.

enum `dns_resolve_status`

Status values for the callback.

Values:

enumerator `DNS_EAI_BADFLAGS` = -1

Invalid value for ai_flags field.

enumerator DNS_EAI_NONAME = -2
NAME or SERVICE is unknown.

enumerator DNS_EAI_AGAIN = -3
Temporary failure in name resolution.

enumerator DNS_EAI_FAIL = -4
Non-recoverable failure in name res.

enumerator DNS_EAI_NODATA = -5
No address associated with NAME.

enumerator DNS_EAI_FAMILY = -6
ai_family not supported

enumerator DNS_EAI_SOCKTYPE = -7
ai_socktype not supported

enumerator DNS_EAI_SERVICE = -8
SRV not supported for ai_socktype.

enumerator DNS_EAI_ADDRFAMILY = -9
Address family for NAME not supported.

enumerator DNS_EAI_MEMORY = -10
Memory allocation failure.

enumerator DNS_EAI_SYSTEM = -11
System error returned in errno.

enumerator DNS_EAI_OVERFLOW = -12
Argument buffer overflow.

enumerator DNS_EAI_INPROGRESS = -100
Processing request in progress.

enumerator DNS_EAI_CANCELED = -101
Request canceled.

enumerator DNS_EAI_NOTCANCELED = -102
Request not canceled.

enumerator DNS_EAI_ALLDONE = -103
All requests done.

enumerator DNS_EAI_IDN_ENCODE = -105
IDN encoding failed.

Functions

int **dns_resolve_init**(struct *dns_resolve_context* *ctx, const char *dns_servers_str[], const struct *sockaddr* *dns_servers_sa[])

Init DNS resolving context.

This function sets the DNS server address and initializes the DNS context that is used by the actual resolver. DNS server addresses can be specified either in textual form, or as struct sockaddr (or both). Note that the recommended way to resolve DNS names is to use the *dns_get_addr_info()* API. In that case user does not need to call *dns_resolve_init()* as the DNS servers are already setup by the system.

Parameters

- **ctx** – DNS context. If the context variable is allocated from the stack, then the variable needs to be valid for the whole duration of the resolving. Caller does not need to fill the variable beforehand or edit the context afterwards.
- **dns_servers_str** – DNS server addresses using textual strings. The array is NULL terminated. The port number can be given in the string. Syntax for the server addresses with or without port numbers: IPv4 : 10.0.9.1 IPv4 + port : 10.0.9.1:5353 IPv6 : 2001:db8::22:42 IPv6 + port : [2001:db8::22:42]:5353
- **dns_servers_sa** – DNS server addresses as struct sockaddr. The array is NULL terminated. Port numbers are optional in struct sockaddr, the default will be used if set to 0.

Returns

0 if ok, <0 if error.

int **dns_resolve_init_default**(struct *dns_resolve_context* *ctx)

Init DNS resolving context with default Kconfig options.

Parameters

- **ctx** – DNS context.

Returns

0 if ok, <0 if error.

int **dns_resolve_close**(struct *dns_resolve_context* *ctx)

Close DNS resolving context.

This releases DNS resolving context and marks the context unusable. Caller must call the *dns_resolve_init()* again to make context usable.

Parameters

- **ctx** – DNS context

Returns

0 if ok, <0 if error.

int **dns_resolve_reconfigure**(struct *dns_resolve_context* *ctx, const char *servers_str[], const struct *sockaddr* *servers_sa[])

Reconfigure DNS resolving context.

Reconfigures DNS context with new server list.

Parameters

- **ctx** – DNS context

- **servers_str** – DNS server addresses using textual strings. The array is NULL terminated. The port number can be given in the string. Syntax for the server addresses with or without port numbers: IPv4 : 10.0.9.1 IPv4 + port : 10.0.9.1:5353 IPv6 : 2001:db8::22:42 IPv6 + port : [2001:db8::22:42]:5353
- **servers_sa** – DNS server addresses as struct sockaddr. The array is NULL terminated. Port numbers are optional in struct sockaddr, the default will be used if set to 0.

Returns
0 if ok, <0 if error.

int **dns_resolve_cancel**(struct *dns_resolve_context* *ctx, uint16_t dns_id)

Cancel a pending DNS query.

This releases DNS resources used by a pending query.

Parameters

- **ctx** – DNS context
- **dns_id** – DNS id of the pending query

Returns
0 if ok, <0 if error.

int **dns_resolve_cancel_with_name**(struct *dns_resolve_context* *ctx, uint16_t dns_id, const char *query_name, enum *dns_query_type* query_type)

Cancel a pending DNS query using id, name and type.

This releases DNS resources used by a pending query.

Parameters

- **ctx** – DNS context
- **dns_id** – DNS id of the pending query
- **query_name** – Name of the resource we are trying to query (hostname)
- **query_type** – Type of the query (A or AAAA)

Returns
0 if ok, <0 if error.

int **dns_resolve_name**(struct *dns_resolve_context* *ctx, const char *query, enum *dns_query_type* type, uint16_t *dns_id, *dns_resolve_cb_t* cb, void *user_data, int32_t timeout)

Resolve DNS name.

This function can be used to resolve e.g., IPv4 or IPv6 address. Note that this is asynchronous call, the function will return immediately and system will call the callback after resolving has finished or timeout has occurred. We might send the query to multiple servers (if there are more than one server configured), but we only use the result of the first received response.

Parameters

- **ctx** – DNS context
- **query** – What the caller wants to resolve.
- **type** – What kind of data the caller wants to get.
- **dns_id** – DNS id is returned to the caller. This is needed if one wishes to cancel the query. This can be set to NULL if there is no need to cancel the query.

- **cb** – Callback to call after the resolving has finished or timeout has happened.
- **user_data** – The user data.
- **timeout** – The timeout value for the query. Possible values: SYS_FOREVER_MS: the query is tried forever, user needs to cancel it manually if it takes too long time to finish >0: start the query and let the system timeout it after specified ms

Returns
0 if resolving was started ok, < 0 otherwise

struct *dns_resolve_context* ***dns_resolve_get_default**(void)

Get default DNS context.

The system level DNS context uses DNS servers that are defined in project config file. If no DNS servers are defined by the user, then resolving DNS names using default DNS context will do nothing. The configuration options are described in subsys/net/lib/dns/Kconfig file.

Returns
Default DNS context.

static inline int **dns_get_addr_info**(const char *query, enum *dns_query_type* type, uint16_t *dns_id, *dns_resolve_cb_t* cb, void *user_data, int32_t timeout)

Get IP address info from DNS.

This function can be used to resolve e.g., IPv4 or IPv6 address. Note that this is asynchronous call, the function will return immediately and system will call the callback after resolving has finished or timeout has occurred. We might send the query to multiple servers (if there are more than one server configured), but we only use the result of the first received response. This variant uses system wide DNS servers.

Parameters

- **query** – What the caller wants to resolve.
- **type** – What kind of data the caller wants to get.
- **dns_id** – DNS id is returned to the caller. This is needed if one wishes to cancel the query. This can be set to NULL if there is no need to cancel the query.
- **cb** – Callback to call after the resolving has finished or timeout has happened.
- **user_data** – The user data.
- **timeout** – The timeout value for the connection. Possible values: SYS_FOREVER_MS: the query is tried forever, user needs to cancel it manually if it takes too long time to finish >0: start the query and let the system timeout it after specified ms

Returns
0 if resolving was started ok, < 0 otherwise

static inline int **dns_cancel_addr_info**(uint16_t dns_id)

Cancel a pending DNS query.

This releases DNS resources used by a pending query.

Parameters

- **dns_id** – DNS id of the pending query

Returns
0 if ok, <0 if error.

struct **dns_addrinfo**

#include <dns_resolve.h> Address info struct is passed to callback that gets all the results.

Public Members

struct *sockaddr* **ai_addr**

IP address information.

socklen_t **ai_addrlen**

Length of the ai_addr field.

uint8_t **ai_family**

Address family of the address information.

char **ai_canonname**[20 + 1]

Canonical name of the address.

struct **dns_resolve_context**

#include <dns_resolve.h> DNS resolve context structure.

Public Members

struct *k_mutex* **lock**

Prevent concurrent access.

k_timeout_t **buf_timeout**

This timeout is also used when a buffer is required from the buffer pools.

enum dns_resolve_context_state **state**

Is this context in use.

struct **dns_pending_query**

#include <dns_resolve.h> Result callbacks.

We have multiple callbacks here so that it is possible to do multiple queries at the same time.

Contents of this structure can be inspected and changed only when the lock is held.

Public Members

struct *k_work_delayable* **timer**

Timeout timer.

struct *dns_resolve_context* ***ctx**

Back pointer to ctx, needed in timeout handler.

dns_resolve_cb_t **cb**

Result callback.

A null value indicates the slot is not in use.

void ***user_data**

User data.

k_timeout_t **timeout**

TX timeout.

const char ***query**

String containing the thing to resolve like www.example.com.

This is set to a non-null value when the query is started, and is not used thereafter.

If the query completed at a point where the work item was still pending the pointer is cleared to indicate that the query is complete, but release of the query slot will be deferred until a request for a slot determines that the work item has been released.

enum *dns_query_type* **query_type**

Query type.

uint16_t **id**

DNS id of this query.

uint16_t **query_hash**

Hash of the DNS name + query type we are querying.

This hash is calculated so we can match the response that we are receiving. This is needed mainly for mDNS which is setting the DNS id to 0, which means that the id alone cannot be used to find correct pending query.

struct **dns_server**

#include <*dns_resolve.h*> List of configured DNS servers.

Public Members

struct *sockaddr* **dns_server**

DNS server information.

int **sock**

Connection to the DNS server.

uint8_t **is_mdns**

Is this server mDNS one.

uint8_t **is_llmnr**

Is this server LLMNR one.

Network Management

- *Overview*
- *Requesting a defined procedure*
- *Listening to network events*
- *Defining a network management procedure*
- *Signaling a network event*
- *API Reference*

Overview The Network Management APIs allow applications, as well as network layer code itself, to call defined network routines at any level in the IP stack, or receive notifications on relevant network events. For example, by using these APIs, application code can request a scan be done on a Wi-Fi- or Bluetooth-based network interface, or request notification if a network interface IP address changes.

The Network Management API implementation is designed to save memory by eliminating code at build time for management routines that are not used. Distinct and statically defined APIs for network management procedures are not used. Instead, defined procedure handlers are registered by using a *NET_MGMT_REGISTER_REQUEST_HANDLER* macro. Procedure requests are done through a single *net_mgmt()* API that invokes the registered handler for the corresponding request.

The current implementation is experimental and may change and improve in future releases.

Requesting a defined procedure All network management requests are of the form `net_mgmt(mgmt_request, ...)`. The `mgmt_request` parameter is a bit mask that tells which stack layer is targeted, if a `net_if` object is implied, and the specific management procedure being requested. The available procedure requests depend on what has been implemented in the stack.

To avoid extra cost, all *net_mgmt()* calls are direct. Though this may change in a future release, it will not affect the users of this function.

Listening to network events You can receive notifications on network events by registering a callback function and specifying a set of events used to filter when your callback is invoked. The callback will have to be unique for a pair of layer and code, whereas on the command part it will be a mask of events.

At runtime two functions are available, *net_mgmt_add_event_callback()* for registering the callback function, and *net_mgmt_del_event_callback()* for unregistering a callback. A helper function, *net_mgmt_init_event_callback()*, can be used to ease the initialization of the callback structure.

Additionally *NET_MGMT_REGISTER_EVENT_HANDLER* can be used to register a callback handler at compile time.

When an event occurs that matches a callback's event set, the associated callback function is invoked with the actual event code. This makes it possible for different events to be handled by the same callback function, if desired.

Warning

Event set filtering allows false positives for events that have the same layer and layer code. A callback handler function **must** check the event code (passed as an argument) against the specific network events it will handle, **regardless** of how many events were in the set passed to *net_mgmt_init_event_callback()*.

Note that in order to receive events from multiple layers, one must have multiple listeners registered, one for each layer being listened. The callback handler function can be shared between different layer events.

(False positives can occur for events which have the same layer and layer code.)

An example follows.

```
/*
 * Set of events to handle.
 * See e.g. include/net/net_event.h for some NET_EVENT_xxx values.
 */
#define EVENT_IFACE_SET (NET_EVENT_IF_xxx | NET_EVENT_IF_yyy)
#define EVENT_IPV4_SET (NET_EVENT_IPV4_xxx | NET_EVENT_IPV4_yyy)

struct net_mgmt_event_callback iface_callback;
struct net_mgmt_event_callback ipv4_callback;

void callback_handler(struct net_mgmt_event_callback *cb,
                      uint32_t mgmt_event,
                      struct net_if *iface)
{
        if (mgmt_event == NET_EVENT_IF_xxx) {
                /* Handle NET_EVENT_IF_xxx */
        } else if (mgmt_event == NET_EVENT_IF_yyy) {
                /* Handle NET_EVENT_IF_yyy */
        } else if (mgmt_event == NET_EVENT_IPV4_xxx) {
                /* Handle NET_EVENT_IPV4_xxx */
        } else if (mgmt_event == NET_EVENT_IPV4_yyy) {
                /* Handle NET_EVENT_IPV4_yyy */
        } else {
                /* Spurious (false positive) invocation. */
        }
}

void register_cb(void)
{
        net_mgmt_init_event_callback(&iface_callback, callback_handler,
                                     EVENT_IFACE_SET);
        net_mgmt_init_event_callback(&ipv4_callback, callback_handler,
                                     EVENT_IPV4_SET);
        net_mgmt_add_event_callback(&iface_callback);
        net_mgmt_add_event_callback(&ipv4_callback);
}
```

Or similarly using *NET_MGMT_REGISTER_EVENT_HANDLER*.

Note

The `info` and `info_length` arguments are only usable if `CONFIG_NET_MGMT_EVENT_INFO` is enabled. Otherwise these are `NULL` and zero.

```
/*
 * Set of events to handle.
 */
#define EVENT_IFACE_SET (NET_EVENT_IF_xxx | NET_EVENT_IF_yyy)
#define EVENT_IPV4_SET (NET_EVENT_IPV4_xxx | NET_EVENT_IPV4_yyy)

static void event_handler(uint32_t mgmt_event, struct net_if *iface,
                          void *info, size_t info_length,
                          void *user_data)
{
        if (mgmt_event == NET_EVENT_IF_xxx) {
                /* Handle NET_EVENT_IF_xxx */
        } else if (mgmt_event == NET_EVENT_IF_yyy) {
                /* Handle NET_EVENT_IF_yyy */
        } else if (mgmt_event == NET_EVENT_IPV4_xxx) {
                /* Handle NET_EVENT_IPV4_xxx */
        } else if (mgmt_event == NET_EVENT_IPV4_yyy) {
                /* Handle NET_EVENT_IPV4_yyy */
        } else {
                /* Spurious (false positive) invocation. */
        }
}

NET_MGMT_REGISTER_EVENT_HANDLER(iface_event_handler, EVENT_IFACE_SET,
                                event_handler, NULL);
NET_MGMT_REGISTER_EVENT_HANDLER(ipv4_event_handler, EVENT_IPV4_SET,
                                event_handler, NULL);
```

See include/zephyr/net/net_event.h for available generic core events that can be listened to.

Defining a network management procedure You can provide additional management procedures specific to your stack implementation by defining a handler and registering it with an associated mgmt_request code.

Management request code are defined in relevant places depending on the targeted layer or eventually, if l2 is the layer, on the technology as well. For instance, all IP layer management request code will be found in the include/zephyr/net/net_event.h header file. But in case of an L2 technology, let's say Ethernet, these would be found in include/zephyr/net/ethernet.h

You define your handler modeled with this signature:

```
static int your_handler(uint32_t mgmt_event, struct net_if *iface,
                        void *data, size_t len);
```

and then register it with an associated mgmt_request code:

```
NET_MGMT_REGISTER_REQUEST_HANDLER(<mgmt_request code>, your_handler);
```

This new management procedure could then be called by using:

```
net_mgmt(<mgmt_request code>, ...);
```

Signaling a network event You can signal a specific network event using the *net_mgmt_event_notify()* function and provide the network event code. See include/zephyr/net/net_mgmt.h for details. As for the management request code, event code can be also found on specific L2 technology mgmt headers, for example include/zephyr/net/ieee802154_mgmt.h would be the right place if 802.15.4 L2 is the technology one wants to listen to events.

Related code samples

DHCPv4 client
Start a DHCPv4 client to obtain an IPv4 address from a DHCPv4 server.

DNS resolve
Resolve an IP address for a given hostname.

IPv4 autoconf client
Perform IPv4 autoconfiguration and self-assign a random IPv4 address

Telnet console
Access Zephyr shell over telnet.

API Reference

group `net_mgmt`

Network Management.

Defines

`NET_EVENT_IF_DOWN`
Event emitted when the network interface goes down.

`NET_EVENT_IF_UP`
Event emitted when the network interface goes up.

`NET_EVENT_IF_ADMIN_DOWN`
Event emitted when the network interface is taken down manually.

`NET_EVENT_IF_ADMIN_UP`
Event emitted when the network interface goes up manually.

`NET_EVENT_IPV6_ADDR_ADD`
Event emitted when an IPv6 address is added to the system.

`NET_EVENT_IPV6_ADDR_DEL`
Event emitted when an IPv6 address is removed from the system.

`NET_EVENT_IPV6_MADDR_ADD`
Event emitted when an IPv6 multicast address is added to the system.

`NET_EVENT_IPV6_MADDR_DEL`
Event emitted when an IPv6 multicast address is removed from the system.

`NET_EVENT_IPV6_PREFIX_ADD`
Event emitted when an IPv6 prefix is added to the system.

`NET_EVENT_IPV6_PREFIX_DEL`
Event emitted when an IPv6 prefix is removed from the system.

`NET_EVENT_IPV6_MCAST_JOIN`
: Event emitted when an IPv6 multicast group is joined.

`NET_EVENT_IPV6_MCAST_LEAVE`
: Event emitted when an IPv6 multicast group is left.

`NET_EVENT_IPV6_ROUTER_ADD`
: Event emitted when an IPv6 router is added to the system.

`NET_EVENT_IPV6_ROUTER_DEL`
: Event emitted when an IPv6 router is removed from the system.

`NET_EVENT_IPV6_ROUTE_ADD`
: Event emitted when an IPv6 route is added to the system.

`NET_EVENT_IPV6_ROUTE_DEL`
: Event emitted when an IPv6 route is removed from the system.

`NET_EVENT_IPV6_DAD_SUCCEED`
: Event emitted when an IPv6 duplicate address detection succeeds.

`NET_EVENT_IPV6_DAD_FAILED`
: Event emitted when an IPv6 duplicate address detection fails.

`NET_EVENT_IPV6_NBR_ADD`
: Event emitted when an IPv6 neighbor is added to the system.

`NET_EVENT_IPV6_NBR_DEL`
: Event emitted when an IPv6 neighbor is removed from the system.

`NET_EVENT_IPV6_DHCP_START`
: Event emitted when an IPv6 DHCP client starts.

`NET_EVENT_IPV6_DHCP_BOUND`
: Event emitted when an IPv6 DHCP client address is bound.

`NET_EVENT_IPV6_DHCP_STOP`
: Event emitted when an IPv6 DHCP client is stopped.

`NET_EVENT_IPV6_ADDR_DEPRECATED`
: IPv6 address is deprecated.

`NET_EVENT_IPV6_PE_ENABLED`
: IPv6 Privacy extension is enabled.

`NET_EVENT_IPV6_PE_DISABLED`
: IPv6 Privacy extension is disabled.

NET_EVENT_IPV6_PE_FILTER_ADD
: IPv6 Privacy extension filter is added.

NET_EVENT_IPV6_PE_FILTER_DEL
: IPv6 Privacy extension filter is removed.

NET_EVENT_IPV4_ADDR_ADD
: Event emitted when an IPv4 address is added to the system.

NET_EVENT_IPV4_ADDR_DEL
: Event emitted when an IPv4 address is removed from the system.

NET_EVENT_IPV4_MADDR_ADD
: Event emitted when an IPv4 multicast address is added to the system.

NET_EVENT_IPV4_MADDR_DEL
: Event emitted when an IPv4 multicast address is removed from the system.

NET_EVENT_IPV4_ROUTER_ADD
: Event emitted when an IPv4 router is added to the system.

NET_EVENT_IPV4_ROUTER_DEL
: Event emitted when an IPv4 router is removed from the system.

NET_EVENT_IPV4_DHCP_START
: Event emitted when an IPv4 DHCP client is started.

NET_EVENT_IPV4_DHCP_BOUND
: Event emitted when an IPv4 DHCP client address is bound.

NET_EVENT_IPV4_DHCP_STOP
: Event emitted when an IPv4 DHCP client is stopped.

NET_EVENT_IPV4_MCAST_JOIN
: Event emitted when an IPv4 multicast group is joined.

NET_EVENT_IPV4_MCAST_LEAVE
: Event emitted when an IPv4 multicast group is left.

NET_EVENT_IPV4_ACD_SUCCEED
: Event emitted when an IPv4 address conflict detection succeeds.

NET_EVENT_IPV4_ACD_FAILED
: Event emitted when an IPv4 address conflict detection fails.

NET_EVENT_IPV4_ACD_CONFLICT
: Event emitted when an IPv4 address conflict was detected after the address was confirmed as safe to use.

  It's up to the application to determine on how to act in such case.

NET_EVENT_L4_CONNECTED

Event emitted when the system is considered to be connected.

The connected in this context means that the network interface is up, and the interface has either IPv4 or IPv6 address assigned to it.

NET_EVENT_L4_DISCONNECTED

Event emitted when the system is no longer connected.

Typically this means that network connectivity is lost either by the network interface is going down, or the interface has no longer an IP address etc.

NET_EVENT_L4_IPV4_CONNECTED

Event raised when IPv4 network connectivity is available.

NET_EVENT_L4_IPV4_DISCONNECTED

Event emitted when IPv4 network connectivity is lost.

NET_EVENT_L4_IPV6_CONNECTED

Event emitted when IPv6 network connectivity is available.

NET_EVENT_L4_IPV6_DISCONNECTED

Event emitted when IPv6 network connectivity is lost.

NET_EVENT_DNS_SERVER_ADD

Event emitted when a DNS server is added to the system.

NET_EVENT_DNS_SERVER_DEL

Event emitted when a DNS server is removed from the system.

NET_EVENT_HOSTNAME_CHANGED

Event emitted when the system hostname is changed.

NET_EVENT_CAPTURE_STARTED

Network packet capture is started.

NET_EVENT_CAPTURE_STOPPED

Network packet capture is stopped.

net_mgmt(_mgmt_request, _iface, _data, _len)

Generate a network management event.

Parameters

- **_mgmt_request** – Management event identifier
- **_iface** – Network interface
- **_data** – Any additional data for the event
- **_len** – Length of the additional data.

NET_MGMT_DEFINE_REQUEST_HANDLER(_mgmt_request)

Declare a request handler function for the given network event.

Parameters

- **_mgmt_request** – Management event identifier

NET_MGMT_REGISTER_REQUEST_HANDLER(_mgmt_request, _func)

Create a request handler function for the given network event.

Parameters

- **_mgmt_request** – Management event identifier
- **_func** – Function for handling this event

NET_MGMT_REGISTER_EVENT_HANDLER(_name, _event_mask, _func, _user_data)

Define a static network event handler.

Parameters

- **_name** – Name of the event handler.
- **_event_mask** – A mask of network events on which the passed handler should be called in case those events come. Note that only the command part is treated as a mask, matching one to several commands. Layer and layer code will be made of an exact match. This means that in order to receive events from multiple layers, one must have multiple listeners registered, one for each layer being listened.
- **_func** – The function to be called upon network events being emitted.
- **_user_data** – User data passed to the handler being called on network events.

Typedefs

typedef int (***net_mgmt_request_handler_t**)(uint32_t mgmt_request, struct *net_if* *iface, void *data, size_t len)

Signature which all Net MGMT request handler need to follow.

Param mgmt_request
The exact request value the handler is being called through

Param iface
A valid pointer on struct *net_if* if the request is meant to be tied to a network interface. NULL otherwise.

Param data
A valid pointer on a data understood by the handler. NULL otherwise.

Param len
Length in byte of the memory pointed by data.

typedef void (***net_mgmt_event_handler_t**)(struct *net_mgmt_event_callback* *cb, uint32_t mgmt_event, struct *net_if* *iface)

Define the user's callback handler function signature.

Param cb
Original struct *net_mgmt_event_callback* owning this handler.

Param mgmt_event
The network event being notified.

Param iface
A pointer on a struct *net_if* to which the event belongs to, if it's an event on an iface. NULL otherwise.

typedef void (***net_mgmt_event_static_handler_t**)(uint32_t mgmt_event, struct *net_if* *iface, void *info, size_t info_length, void *user_data)

Define the user's callback handler function signature.

Param mgmt_event
The network event being notified.

Param iface
A pointer on a struct *net_if* to which the event belongs to, if it's an event on an iface. NULL otherwise.

Param info
A valid pointer on a data understood by the handler. NULL otherwise.

Param info_length
Length in bytes of the memory pointed by `info`.

Param user_data
Data provided by the user to the handler.

Functions

static inline void **net_mgmt_init_event_callback**(struct *net_mgmt_event_callback* *cb, *net_mgmt_event_handler_t* handler, uint32_t mgmt_event_mask)

Helper to initialize a struct *net_mgmt_event_callback* properly.

Parameters

- **cb** – A valid application's callback structure pointer.
- **handler** – A valid handler function pointer.
- **mgmt_event_mask** – A mask of relevant events for the handler

void **net_mgmt_add_event_callback**(struct *net_mgmt_event_callback* *cb)

Add a user callback.

Parameters

- **cb** – A valid pointer on user's callback to add.

void **net_mgmt_del_event_callback**(struct *net_mgmt_event_callback* *cb)

Delete a user callback.

Parameters

- **cb** – A valid pointer on user's callback to delete.

void **net_mgmt_event_notify_with_info**(uint32_t mgmt_event, struct *net_if* *iface, const void *info, size_t length)

Used by the system to notify an event.

Note: info and length are disabled if CONFIG_NET_MGMT_EVENT_INFO is not defined.

Parameters

- **mgmt_event** – The actual network event code to notify
- **iface** – a valid pointer on a struct *net_if* if only the event is based on an iface. NULL otherwise.

- **info** – A valid pointer on the information you want to pass along with the event. NULL otherwise. Note the data pointed there is normalized by the related event.
- **length** – size of the data pointed by info pointer.

static inline void **net_mgmt_event_notify**(uint32_t mgmt_event, struct *net_if* *iface)

Used by the system to notify an event without any additional information.

Parameters

- **mgmt_event** – The actual network event code to notify
- **iface** – A valid pointer on a struct *net_if* if only the event is based on an iface. NULL otherwise.

int **net_mgmt_event_wait**(uint32_t mgmt_event_mask, uint32_t *raised_event, struct *net_if* **iface, const void **info, size_t *info_length, *k_timeout_t* timeout)

Used to wait synchronously on an event mask.

Parameters

- **mgmt_event_mask** – A mask of relevant events to wait on.
- **raised_event** – a pointer on a uint32_t to get which event from the mask generated the event. Can be NULL if the caller is not interested in that information.
- **iface** – a pointer on a place holder for the iface on which the event has originated from. This is valid if only the event mask has bit NET_MGMT_IFACE_BIT set relevantly, depending on events the caller wants to listen to.
- **info** – a valid pointer if user wants to get the information the event might bring along. NULL otherwise.
- **info_length** – tells how long the info memory area is. Only valid if the info is not NULL.
- **timeout** – A timeout delay. K_FOREVER can be used to wait indefinitely.

Returns

0 on success, a negative error code otherwise. -ETIMEDOUT will be specifically returned if the timeout kick-in instead of an actual event.

int **net_mgmt_event_wait_on_iface**(struct *net_if* *iface, uint32_t mgmt_event_mask, uint32_t *raised_event, const void **info, size_t *info_length, *k_timeout_t* timeout)

Used to wait synchronously on an event mask for a specific iface.

Parameters

- **iface** – a pointer on a valid network interface to listen event to
- **mgmt_event_mask** – A mask of relevant events to wait on. Listened to events should be relevant to iface events and thus have the bit NET_MGMT_IFACE_BIT set.
- **raised_event** – a pointer on a uint32_t to get which event from the mask generated the event. Can be NULL if the caller is not interested in that information.
- **info** – a valid pointer if user wants to get the information the event might bring along. NULL otherwise.
- **info_length** – tells how long the info memory area is. Only valid if the info is not NULL.

- **timeout** – A timeout delay. K_FOREVER can be used to wait indefinitely.

Returns
0 on success, a negative error code otherwise. -ETIMEDOUT will be specifically returned if the timeout kick-in instead of an actual event.

void **net_mgmt_event_init**(void)

Used by the core of the network stack to initialize the network event processing.

struct **net_event_ipv6_addr**

#include <*net_event.h*> Network Management event information structure Used to pass information on network events like NET_EVENT_IPV6_ADDR_ADD, NET_EVENT_IPV6_ADDR_DEL, NET_EVENT_IPV6_MADDR_ADD and NET_EVENT_IPV6_MADDR_DEL when CONFIG_NET_MGMT_EVENT_INFO enabled and event generator pass the information.

Public Members

struct *in6_addr* **addr**

IPv6 address related to this event.

struct **net_event_ipv6_nbr**

#include <*net_event.h*> Network Management event information structure Used to pass information on network events like NET_EVENT_IPV6_NBR_ADD and NET_EVENT_IPV6_NBR_DEL when CONFIG_NET_MGMT_EVENT_INFO enabled and event generator pass the information.

Note

: idx will be '-1' in case of NET_EVENT_IPV6_NBR_DEL event.

Public Members

struct *in6_addr* **addr**

Neighbor IPv6 address.

int **idx**

Neighbor index in cache.

struct **net_event_ipv6_route**

#include <*net_event.h*> Network Management event information structure Used to pass information on network events like NET_EVENT_IPV6_ROUTE_ADD and NET_EVENT_IPV6_ROUTE_DEL when CONFIG_NET_MGMT_EVENT_INFO enabled and event generator pass the information.

Public Members

struct *in6_addr* **nexthop**

IPv6 address of the next hop.

struct *in6_addr* **addr**

IPv6 address or prefix of the route.

uint8_t **prefix_len**

IPv6 prefix length.

struct **net_event_ipv6_prefix**

#include <*net_event.h*> Network Management event information structure Used to pass information on network events like NET_EVENT_IPV6_PREFIX_ADD and NET_EVENT_IPV6_PREFIX_DEL when CONFIG_NET_MGMT_EVENT_INFO is enabled and event generator pass the information.

Public Members

struct *in6_addr* **addr**

IPv6 prefix.

uint8_t **len**

IPv6 prefix length.

uint32_t **lifetime**

IPv6 prefix lifetime in seconds.

struct **net_event_l4_hostname**

#include <*net_event.h*> Network Management event information structure Used to pass information on NET_EVENT_HOSTNAME_CHANGED event when CONFIG_NET_MGMT_EVENT_INFO is enabled and event generator pass the information.

Public Members

char **hostname**[NET_HOSTNAME_SIZE]

New hostname.

struct **net_event_ipv6_pe_filter**

#include <*net_event.h*> Network Management event information structure Used to pass information on network events like NET_EVENT_IPV6_PE_FILTER_ADD and NET_EVENT_IPV6_PE_FILTER_DEL when CONFIG_NET_MGMT_EVENT_INFO is enabled and event generator pass the information.

This is only available if CONFIG_NET_IPV6_PE_FILTER_PREFIX_COUNT is >0.

Public Members

struct *in6_addr* **prefix**

IPv6 address of privacy extension filter.

bool **is_deny_list**

IPv6 filter deny or allow list.

struct **net_mgmt_event_callback**

#include <net_mgmt.h> Network Management event callback structure Used to register a callback into the network management event part, in order to let the owner of this struct to get network event notification based on given event mask.

Public Members

sys_snode_t **node**

Meant to be used internally, to insert the callback into a list.

So nobody should mess with it.

net_mgmt_event_handler_t **handler**

Actual callback function being used to notify the owner.

struct k_sem ***sync_call**

Semaphore meant to be used internally for the synchronous *net_mgmt_event_wait()* function.

uint32_t **event_mask**

A mask of network events on which the above handler should be called in case those events come.

Note that only the command part is treated as a mask, matching one to several commands. Layer and layer code will be made of an exact match. This means that in order to receive events from multiple layers, one must have multiple listeners registered, one for each layer being listened.

uint32_t **raised_event**

Internal place holder when a synchronous event wait is successfully unlocked on a event.

union net_mgmt_event_callback

A mask of network events on which the above handler should be called in case those events come.

Such mask can be modified whenever necessary by the owner, and thus will affect the handler being called or not.

Network Statistics

- *Overview*
- *API Reference*

Overview Network statistics are collected if `CONFIG_NET_STATISTICS` is set. Individual component statistics for IPv4 or IPv6 can be turned off if those statistics are not needed. See various options in subsys/net/ip/Kconfig.stats file for details.

By default, the system collects network statistics per network interface. This can be controlled by `CONFIG_NET_STATISTICS_PER_INTERFACE` option.

The `CONFIG_NET_STATISTICS_USER_API` option can be set if the application wants to collect statistics for further processing. The network management interface API is used for that. See *Network Management* for details.

The `CONFIG_NET_STATISTICS_ETHERNET` option can be set to collect generic Ethernet statistics. If the `CONFIG_NET_STATISTICS_ETHERNET_VENDOR` option is set, then Ethernet device driver can collect Ethernet device specific statistics. These statistics can then be transferred to application for processing.

If the `CONFIG_NET_SHELL` option is set, then network shell can show statistics information with `net stats` command.

Related code samples

Network statistics
Query and display network statistics from a user application.

Wi-Fi shell
Test Wi-Fi functionality using the Wi-Fi shell module.

API Reference

group `net_stats`

Network statistics library.

Typedefs

typedef uint32_t **`net_stats_t`**

Network statistics counter.

struct **`net_stats_bytes`**

#include <net_stats.h> Number of bytes sent and received.

Public Members

net_stats_t **`sent`**

Number of bytes sent.

net_stats_t **`received`**

Number of bytes received.

struct **`net_stats_pkts`**

#include <net_stats.h> Number of network packets sent and received.

Public Members

net_stats_t **tx**
Number of packets sent.

net_stats_t **rx**
Number of packets received.

struct **net_stats_ip**
#include <*net_stats.h*> IP layer statistics.

Public Members

net_stats_t **recv**
Number of received packets at the IP layer.

net_stats_t **sent**
Number of sent packets at the IP layer.

net_stats_t **forwarded**
Number of forwarded packets at the IP layer.

net_stats_t **drop**
Number of dropped packets at the IP layer.

struct **net_stats_ip_errors**
#include <*net_stats.h*> IP layer error statistics.

Public Members

net_stats_t **vhlerr**
Number of packets dropped due to wrong IP version or header length.

net_stats_t **hblenerr**
Number of packets dropped due to wrong IP length, high byte.

net_stats_t **lblenerr**
Number of packets dropped due to wrong IP length, low byte.

net_stats_t **fragerr**
Number of packets dropped because they were IP fragments.

net_stats_t **chkerr**
Number of packets dropped due to IP checksum errors.

net_stats_t **protoerr**
Number of packets dropped because they were neither ICMP, UDP nor TCP.

struct **net_stats_icmp**

#include <net_stats.h> ICMP statistics.

Public Members

net_stats_t **recv**

Number of received ICMP packets.

net_stats_t **sent**

Number of sent ICMP packets.

net_stats_t **drop**

Number of dropped ICMP packets.

net_stats_t **typeerr**

Number of ICMP packets with a wrong type.

net_stats_t **chkerr**

Number of ICMP packets with a bad checksum.

struct **net_stats_tcp**

#include <net_stats.h> TCP statistics.

Public Members

struct *net_stats_bytes* **bytes**

Amount of received and sent TCP application data.

net_stats_t **resent**

Amount of retransmitted data.

net_stats_t **drop**

Number of dropped packets at the TCP layer.

net_stats_t **recv**

Number of received TCP segments.

net_stats_t **sent**

Number of sent TCP segments.

net_stats_t **seg_drop**

Number of dropped TCP segments.

net_stats_t **chkerr**

Number of TCP segments with a bad checksum.

net_stats_t **ackerr**

Number of received TCP segments with a bad ACK number.

net_stats_t **rsterr**

Number of received bad TCP RST (reset) segments.

net_stats_t **rst**

Number of received TCP RST (reset) segments.

net_stats_t **rexmit**

Number of retransmitted TCP segments.

net_stats_t **conndrop**

Number of dropped connection attempts because too few connections were available.

net_stats_t **connrst**

Number of connection attempts for closed ports, triggering a RST.

struct **net_stats_udp**

#include <net_stats.h> UDP statistics.

Public Members

net_stats_t **drop**

Number of dropped UDP segments.

net_stats_t **recv**

Number of received UDP segments.

net_stats_t **sent**

Number of sent UDP segments.

net_stats_t **chkerr**

Number of UDP segments with a bad checksum.

struct **net_stats_ipv6_nd**

#include <net_stats.h> IPv6 neighbor discovery statistics.

Public Members

net_stats_t **drop**

Number of dropped IPv6 neighbor discovery packets.

net_stats_t **recv**

Number of received IPv6 neighbor discovery packets.

net_stats_t **sent**
Number of sent IPv6 neighbor discovery packets.

struct **net_stats_ipv6_mld**
#include <*net_stats.h*> IPv6 multicast listener daemon statistics.

Public Members

net_stats_t **recv**
Number of received IPv6 MLD queries.

net_stats_t **sent**
Number of sent IPv6 MLD reports.

net_stats_t **drop**
Number of dropped IPv6 MLD packets.

struct **net_stats_ipv4_igmp**
#include <*net_stats.h*> IPv4 IGMP daemon statistics.

Public Members

net_stats_t **recv**
Number of received IPv4 IGMP queries.

net_stats_t **sent**
Number of sent IPv4 IGMP reports.

net_stats_t **drop**
Number of dropped IPv4 IGMP packets.

struct **net_stats_tx_time**
#include <*net_stats.h*> Network packet transfer times for calculating average TX time.

Public Members

uint64_t **sum**
Sum of network packet transfer times.

net_stats_t **count**
Number of network packets transferred.

struct **net_stats_rx_time**
#include <*net_stats.h*> Network packet receive times for calculating average RX time.

Public Members

uint64_t **sum**

Sum of network packet receive times.

net_stats_t **count**

Number of network packets received.

struct **net_stats_tc**

#include <net_stats.h> Traffic class statistics.

Public Members

struct *net_stats_tx_time* **tx_time**

Helper for calculating average TX time statistics.

net_stats_t **pkts**

Number of packets sent for this traffic class.

Number of packets received for this traffic class.

net_stats_t **bytes**

Number of bytes sent for this traffic class.

Number of bytes received for this traffic class.

uint8_t **priority**

Priority of this traffic class.

struct *net_stats_tc* **sent**[NET_TC_TX_STATS_COUNT]

TX statistics for each traffic class.

struct *net_stats_rx_time* **rx_time**

Helper for calculating average RX time statistics.

struct *net_stats_tc* **recv**[NET_TC_RX_STATS_COUNT]

RX statistics for each traffic class.

struct **net_stats_pm**

#include <net_stats.h> Power management statistics.

Public Members

uint64_t **overall_suspend_time**

Total suspend time.

net_stats_t **suspend_count**

How many times we were suspended.

uint32_t **last_suspend_time**
How long the last suspend took.

uint32_t **start_time**
Network interface last suspend start time.

struct **net_stats**
#include <net_stats.h> All network statistics in one struct.

Public Members

net_stats_t **processing_error**
Count of malformed packets or packets we do not have handler for.

struct *net_stats_bytes* **bytes**
This calculates amount of data transferred through all the network interfaces.

struct *net_stats_ip_errors* **ip_errors**
IP layer errors.

struct **net_stats_eth_errors**
#include <net_stats.h> Ethernet error statistics.

Public Members

net_stats_t **rx_length_errors**
Number of RX length errors.

net_stats_t **rx_over_errors**
Number of RX overrun errors.

net_stats_t **rx_crc_errors**
Number of RX CRC errors.

net_stats_t **rx_frame_errors**
Number of RX frame errors.

net_stats_t **rx_no_buffer_count**
Number of RX *net_pkt* allocation errors.

net_stats_t **rx_missed_errors**
Number of RX missed errors.

net_stats_t **rx_long_length_errors**
Number of RX long length errors.

net_stats_t **rx_short_length_errors**
Number of RX short length errors.

net_stats_t **rx_align_errors**
Number of RX buffer align errors.

net_stats_t **rx_dma_failed**
Number of RX DMA failed errors.

net_stats_t **rx_buf_alloc_failed**
Number of RX *net_buf* allocation errors.

net_stats_t **tx_aborted_errors**
Number of TX aborted errors.

net_stats_t **tx_carrier_errors**
Number of TX carrier errors.

net_stats_t **tx_fifo_errors**
Number of TX FIFO errors.

net_stats_t **tx_heartbeat_errors**
Number of TX heartbeat errors.

net_stats_t **tx_window_errors**
Number of TX window errors.

net_stats_t **tx_dma_failed**
Number of TX DMA failed errors.

net_stats_t **uncorr_ecc_errors**
Number of uncorrected ECC errors.

net_stats_t **corr_ecc_errors**
Number of corrected ECC errors.

struct **net_stats_eth_flow**
#include <net_stats.h> Ethernet flow control statistics.

Public Members

net_stats_t **rx_flow_control_xon**
Number of RX XON flow control.

net_stats_t **rx_flow_control_xoff**
Number of RX XOFF flow control.

net_stats_t **tx_flow_control_xon**
Number of TX XON flow control.

net_stats_t **tx_flow_control_xoff**
Number of TX XOFF flow control.

struct **net_stats_eth_csum**
#include <*net_stats.h*> Ethernet checksum statistics.

Public Members

net_stats_t **rx_csum_offload_good**
Number of good RX checksum offloading.

net_stats_t **rx_csum_offload_errors**
Number of failed RX checksum offloading.

struct **net_stats_eth_hw_timestamp**
#include <*net_stats.h*> Ethernet hardware timestamp statistics.

Public Members

net_stats_t **rx_hwtstamp_cleared**
Number of RX hardware timestamp cleared.

net_stats_t **tx_hwtstamp_timeouts**
Number of RX hardware timestamp timeout.

net_stats_t **tx_hwtstamp_skipped**
Number of RX hardware timestamp skipped.

struct **net_stats_eth**
#include <*net_stats.h*> All Ethernet specific statistics.

Public Members

struct *net_stats_bytes* **bytes**
Total number of bytes received and sent.

struct *net_stats_pkts* **pkts**
Total number of packets received and sent.

struct *net_stats_pkts* **broadcast**
Total number of broadcast packets received and sent.

struct *net_stats_pkts* **multicast**
Total number of multicast packets received and sent.

struct *net_stats_pkts* **errors**
Total number of errors in RX and TX.

struct *net_stats_eth_errors* **error_details**
Total number of errors in RX and TX.

struct *net_stats_eth_flow* **flow_control**
Total number of flow control errors in RX and TX.

struct *net_stats_eth_csum* **csum**
Total number of checksum errors in RX and TX.

struct *net_stats_eth_hw_timestamp* **hw_timestamp**
Total number of hardware timestamp errors in RX and TX.

net_stats_t **collisions**
Total number of collisions.

net_stats_t **tx_dropped**
Total number of dropped TX packets.

net_stats_t **tx_timeout_count**
Total number of TX timeout errors.

net_stats_t **tx_restart_queue**
Total number of TX queue restarts.

net_stats_t **unknown_protocol**
Total number of RX unknown protocol packets.

struct **net_stats_ppp**
#include <net_stats.h> All PPP specific statistics.

Public Members

struct *net_stats_bytes* **bytes**
Total number of bytes received and sent.

struct *net_stats_pkts* **pkts**
Total number of packets received and sent.

net_stats_t **drop**
Number of received and dropped PPP frames.

net_stats_t **chkerr**
Number of received PPP frames with a bad checksum.

struct **net_stats_sta_mgmt**
#include <*net_stats.h*> All Wi-Fi management statistics.

Public Members

net_stats_t **beacons_rx**
Number of received beacons.

net_stats_t **beacons_miss**
Number of missed beacons.

struct **net_stats_wifi**
#include <*net_stats.h*> All Wi-Fi specific statistics.

Public Members

struct *net_stats_sta_mgmt* **sta_mgmt**
Total number of beacon errors.

struct *net_stats_bytes* **bytes**
Total number of bytes received and sent.

struct *net_stats_pkts* **pkts**
Total number of packets received and sent.

struct *net_stats_pkts* **broadcast**
Total number of broadcast packets received and sent.

struct *net_stats_pkts* **multicast**
Total number of multicast packets received and sent.

struct *net_stats_pkts* **errors**
Total number of errors in RX and TX.

struct *net_stats_pkts* **unicast**
Total number of unicast packets received and sent.

Network Timeout

- *Overview*
- *Use*

- *API Reference*

Overview Zephyr's network infrastructure mostly uses the millisecond-resolution uptime clock to track timeouts, with both deadlines and durations measured with 32-bit unsigned values. The 32-bit value rolls over at 49 days 17 hours 2 minutes 47.296 seconds.

Timeout processing is often affected by latency, so that the time at which the timeout is checked may be some time after it should have expired. Handling this correctly without arbitrary expectations of maximum latency requires that the maximum delay that can be directly represented be a 31-bit non-negative number (`INT32_MAX`), which overflows at 24 days 20 hours 31 minutes 23.648 seconds.

Most network timeouts are shorter than the delay rollover, but a few protocols allow for delays that are represented as unsigned 32-bit values counting seconds, which corresponds to a 42-bit millisecond count.

The net_timeout API provides a generic timeout mechanism to correctly track the remaining time for these extended-duration timeouts.

Use The simplest use of this API is:

1. Configure a network timeout using *net_timeout_set()*.
2. Use *net_timeout_evaluate()* to determine how long it is until the timeout occurs. Schedule a timeout to occur after this delay.
3. When the timeout callback is invoked, use *net_timeout_evaluate()* again to determine whether the timeout has completed, or whether there is additional time remaining. If the latter, reschedule the callback.
4. While the timeout is running, use *net_timeout_remaining()* to get the number of seconds until the timeout expires. This may be used to explicitly update the timeout, which should be done by canceling any pending callback and restarting from step 1 with the new timeout.

The *net_timeout* contains a `sys_snode_t` that allows multiple timeout instances to be aggregated to share a single kernel timer element. The application must use *net_timeout_evaluate()* on all instances to determine the next timeout event to occur.

net_timeout_deadline() may be used to reconstruct the full-precision deadline of the timeout. This exists primarily for testing but may have use in some applications, as it does allow a millisecond-resolution calculation of remaining time.

API Reference

group **net_timeout**

Network long timeout primitives and helpers.

Defines

`NET_TIMEOUT_MAX_VALUE`

Divisor used to support ms resolution timeouts.

Because delays are processed in work queues which are not invoked synchronously with clock changes we need to be able to detect timeouts after they occur, which requires comparing "deadline" to "now" with enough "slop" to handle any observable latency due to "now" advancing past "deadline".

The simplest solution is to use the native conversion of the well-defined 32-bit unsigned difference to a 32-bit signed difference, which caps the maximum delay at INT32_MAX. This is compatible with the standard mechanism for detecting completion of deadlines that do not overflow their representation.

Functions

void **net_timeout_set**(struct *net_timeout* *timeout, uint32_t lifetime, uint32_t now)

Configure a network timeout structure.

Parameters

- **timeout** – a pointer to the timeout state.
- **lifetime** – the duration of the timeout in seconds.
- **now** – the time at which the timeout started counting down, in milliseconds. This is generally a captured value of *k_uptime_get_32()*.

int64_t **net_timeout_deadline**(const struct *net_timeout* *timeout, int64_t now)

Return the 64-bit system time at which the timeout will complete.

> **Note**
>
> Correct behavior requires invocation of *net_timeout_evaluate()* at its specified intervals.

Parameters

- **timeout** – state a pointer to the timeout state, initialized by *net_timeout_set()* and maintained by *net_timeout_evaluate()*.
- **now** – the full-precision value of *k_uptime_get()* relative to which the deadline will be calculated.

Returns

the value of *k_uptime_get()* at which the timeout will expire.

uint32_t **net_timeout_remaining**(const struct *net_timeout* *timeout, uint32_t now)

Calculate the remaining time to the timeout in whole seconds.

> **Note**
>
> This function rounds the remaining time down, i.e. if the timeout will occur in 3500 milliseconds the value 3 will be returned.

> **Note**
>
> Correct behavior requires invocation of *net_timeout_evaluate()* at its specified intervals.

Parameters

- **timeout** – a pointer to the timeout state
- **now** – the time relative to which the estimate of remaining time should be calculated. This should be recently captured value from *k_uptime_get_32()*.

Return values

- `0` – if the timeout has completed.
- `positive` – the remaining duration of the timeout, in seconds.

uint32_t **net_timeout_evaluate**(struct *net_timeout* *timeout, uint32_t now)

Update state to reflect elapsed time and get new delay.

This function must be invoked periodically to (1) apply the effect of elapsed time on what remains of a total delay that exceeded the maximum representable delay, and (2) determine that either the timeout has completed or that the infrastructure must wait a certain period before checking again for completion.

Parameters

- `timeout` – a pointer to the timeout state
- `now` – the time relative to which the estimate of remaining time should be calculated. This should be recently captured value from *k_uptime_get_32()*.

Return values

- `0` – if the timeout has completed
- `positive` – the maximum delay until the state of this timeout should be re-evaluated, in milliseconds.

struct **net_timeout**

#include <net_timeout.h> Generic struct for handling network timeouts.

Except for the linking node, all access to state from these objects must go through the defined API.

Public Members

sys_snode_t **node**

Used to link multiple timeouts that share a common timer infrastructure.

For examples a set of related timers may use a single delayed work structure, which is always scheduled at the shortest time to a timeout event.

uint32_t **timer_start**

Time at which the timer was last set.

This usually corresponds to the low 32 bits of *k_uptime_get()*.

uint32_t **timer_timeout**

Portion of remaining timeout that does not exceed NET_TIMEOUT_MAX_VALUE.

This value is updated in parallel with timer_start and wrap_counter by *net_timeout_evaluate()*.

uint32_t **wrap_counter**

Timer wrap count.

This tracks multiples of NET_TIMEOUT_MAX_VALUE milliseconds that have yet to pass. It is also updated along with timer_start and wrap_counter by *net_timeout_evaluate()*.

Networking Context The net_context API is not meant for application use. Application should use *BSD Sockets* API instead.

Promiscuous Mode

- *Overview*
- *Sample usage*
- *API Reference*

Overview Promiscuous mode is a mode for a network interface controller that causes it to pass all traffic it receives to the application rather than passing only the frames that the controller is specifically programmed to receive. This mode is normally used for packet sniffing as used to diagnose network connectivity issues by showing an application all the data being transferred over the network. (See the Wikipedia article on promiscuous mode for more information.)

The network promiscuous APIs are used to enable and disable this mode, and to wait for and receive a network data to arrive. Not all network technologies or network device drivers support promiscuous mode.

Sample usage First the promiscuous mode needs to be turned ON by the application like this:

```
ret = net_promisc_mode_on(iface);
if (ret < 0) {
        if (ret == -EALREADY) {
                printf("Promiscuous mode already enabled\n");
        } else {
                printf("Cannot enable promiscuous mode for "
                       "interface %p (%d)\n", iface, ret);
        }
}
```

If there is no error, then the application can start to wait for network data:

```
while (true) {
        pkt = net_promisc_mode_wait_data(K_FOREVER);
        if (pkt) {
                print_info(pkt);
        }

        net_pkt_unref(pkt);
}
```

Finally the promiscuous mode can be turned OFF by the application like this:

```
ret = net_promisc_mode_off(iface);
if (ret < 0) {
        if (ret == -EALREADY) {
                printf("Promiscuous mode already disabled\n");
        } else {
                printf("Cannot disable promiscuous mode for "
                       "interface %p (%d)\n", iface, ret);
        }
}
```

See net-promiscuous-mode for a more comprehensive example.

Related code samples

Promiscuous mode
Enable promiscuous mode on all interfaces and print information about incoming packets.

API Reference

group `promiscuous`

Promiscuous mode support.

Functions

static inline struct *net_pkt* ***net_promisc_mode_wait_data**(*k_timeout_t* timeout)

Start to wait received network packets.

Parameters

- **timeout** – How long to wait before returning.

Returns
Received *net_pkt*, NULL if not received any packet.

static inline int **net_promisc_mode_on**(struct *net_if* *iface)

Enable promiscuous mode for a given network interface.

Parameters

- **iface** – Network interface

Returns
0 if ok, <0 if error

static inline int **net_promisc_mode_off**(struct *net_if* *iface)

Disable promiscuous mode for a given network interface.

Parameters

- **iface** – Network interface

Returns
0 if ok, <0 if error

Simple Network Time Protocol Library

- *Overview*
- *API Reference*

Overview The SNTP library implements IETF RFC4330 (Simple Network Time Protocol v4). SNTP provides a way to synchronize clocks in computer networks.

Related code samples

AWS IoT Core MQTT
Connect to AWS IoT Core and publish messages using MQTT.

SNTP client
Use SNTP to get the current time from the host.

API Reference

group `sntp`

Simple Network Time Protocol API.

Functions

int **`sntp_init`**(struct *sntp_ctx* *ctx, struct *sockaddr* *addr, *socklen_t* addr_len)

Initialize SNTP context.

Parameters

- **`ctx`** – Address of sntp context.
- **`addr`** – IP address of NTP/SNTP server.
- **`addr_len`** – IP address length of NTP/SNTP server.

Returns
0 if ok, <0 if error.

int **`sntp_query`**(struct *sntp_ctx* *ctx, uint32_t timeout, struct *sntp_time* *time)

Perform SNTP query.

Parameters

- **`ctx`** – Address of sntp context.
- **`timeout`** – Timeout of waiting for sntp response (in milliseconds).
- **`time`** – Timestamp including integer and fractional seconds since 1 Jan 1970 (output).

Returns
0 if ok, <0 if error (-ETIMEDOUT if timeout).

void **`sntp_close`**(struct *sntp_ctx* *ctx)

Release SNTP context.

Parameters

- **`ctx`** – Address of sntp context.

int **`sntp_simple`**(const char *server, uint32_t timeout, struct *sntp_time* *ts)

Convenience function to query SNTP in one-shot fashion.

Convenience wrapper which calls *getaddrinfo()*, *sntp_init()*, *sntp_query()*, and *sntp_close()*.

Parameters

- **`server`** – Address of server in format addr[:port]
- **`timeout`** – Query timeout

- **ts** – Timestamp including integer and fractional seconds since 1 Jan 1970 (output).

Returns
0 if ok, <0 if error (-ETIMEDOUT if timeout).

int **sntp_simple_addr**(struct *sockaddr* *addr, *socklen_t* addr_len, uint32_t timeout, struct *sntp_time* *ts)

Convenience function to query SNTP in one-shot fashion using a pre-initialized address struct.

Convenience wrapper which calls *sntp_init()*, *sntp_query()* and *sntp_close()*.

Parameters

- **addr** – IP address of NTP/SNTP server.
- **addr_len** – IP address length of NTP/SNTP server.
- **timeout** – Query timeout
- **ts** – Timestamp including integer and fractional seconds since 1 Jan 1970 (output).

Returns
0 if ok, <0 if error (-ETIMEDOUT if timeout).

struct **sntp_time**

#include <*sntp.h*> Time as returned by SNTP API, fractional seconds since 1 Jan 1970.

Public Members

uint64_t **seconds**

Second value.

uint32_t **fraction**

Fractional seconds value.

struct **sntp_ctx**

#include <*sntp.h*> SNTP context.

Public Members

struct *sntp_time* **expected_orig_ts**

Timestamp when the request was sent from client to server.

This is used to check if the originated timestamp in the server reply matches the one in client request.

SOCKS5 Proxy Support

- *Overview*
- *SOCKS5 API*

- *SOCKS5 Proxy Usage in MQTT*

Overview The SOCKS library implements SOCKS5 support, which allows Zephyr to connect to peer devices via a network proxy.

See this SOCKS5 Wikipedia article for a detailed overview of how SOCKS5 works.

For more information about the protocol itself, see IETF RFC1928 SOCKS Protocol Version 5.

SOCKS5 API The SOCKS5 support is enabled by `CONFIG_SOCKS` Kconfig variable. Application wanting to use the SOCKS5 must set the SOCKS5 proxy host address by calling *`setsockopt()`* like this:

```
static int set_proxy(int sock, const struct sockaddr *proxy_addr,
                     socklen_t proxy_addrlen)
{
    int ret;

    ret = setsockopt(sock, SOL_SOCKET, SO_SOCKS5,
                     proxy_addr, proxy_addrlen);
    if (ret < 0) {
            return -errno;
    }

    return 0;
}
```

SOCKS5 Proxy Usage in MQTT For MQTT client, there is `mqtt_client_set_proxy()` API that the application can call to setup SOCKS5 proxy. See mqtt-publisher sample application for usage example.

Trickle Timer Library

- *Overview*
- *API Reference*

Overview The Trickle timer library implements IETF RFC6206 (Trickle Algorithm).

The Trickle algorithm allows nodes in a lossy shared medium (e.g., low-power and lossy networks) to exchange information in a highly robust, energy efficient, simple, and scalable manner.

API Reference

group `trickle`

Trickle algorithm library.

Typedefs

typedef void (*`net_trickle_cb_t`)(struct *net_trickle* *trickle, bool do_suppress, void *user_data)

Trickle timer callback.

The callback is called after Trickle timeout expires.

Param trickle
The trickle context to use.

Param do_suppress
Is TX allowed (true) or not (false).

Param user_data
The user data given in *net_trickle_start()* call.

Functions

int `net_trickle_create`(struct *net_trickle* *trickle, uint32_t Imin, uint8_t Imax, uint8_t k)

Create a Trickle timer.

Parameters

- `trickle` – Pointer to Trickle struct.
- `Imin` – Imin configuration parameter in ms.
- `Imax` – Max number of doublings.
- `k` – Redundancy constant parameter. See RFC 6206 for details.

Returns
Return 0 if ok and <0 if error.

int `net_trickle_start`(struct *net_trickle* *trickle, *net_trickle_cb_t* cb, void *user_data)

Start a Trickle timer.

Parameters

- `trickle` – Pointer to Trickle struct.
- `cb` – User callback to call at time T within the current trickle interval
- `user_data` – User pointer that is passed to callback.

Returns
Return 0 if ok and <0 if error.

int `net_trickle_stop`(struct *net_trickle* *trickle)

Stop a Trickle timer.

Parameters

- `trickle` – Pointer to Trickle struct.

Returns
Return 0 if ok and <0 if error.

void `net_trickle_consistency`(struct *net_trickle* *trickle)

To be called by the protocol handler when it hears a consistent network transmission.

Parameters

- `trickle` – Pointer to Trickle struct.

void **net_trickle_inconsistency**(struct *net_trickle* *trickle)

To be called by the protocol handler when it hears an inconsistent network transmission.

Parameters

- **trickle** – Pointer to Trickle struct.

static inline bool **net_trickle_is_running**(struct *net_trickle* *trickle)

Check if the Trickle timer is running or not.

Parameters

- **trickle** – Pointer to Trickle struct.

Returns

Return True if timer is running and False if not.

struct **net_trickle**

#include <trickle.h> The variable names are taken directly from RFC 6206 when applicable.

Note that the struct members should not be accessed directly but only via the Trickle API.

Public Members

uint32_t **I**

Current interval size.

uint32_t **Imin**

Min interval size in ms.

uint32_t **Istart**

Start of the interval in ms.

uint32_t **Imax_abs**

Max interval size in ms (not doublings)

uint8_t **Imax**

Max number of doublings.

uint8_t **k**

Redundancy constant.

uint8_t **c**

Consistency counter.

bool **double_to**

Flag telling if the internval is doubled.

struct *k_work_delayable* **timer**

Internal timer struct.

net_trickle_cb_t **cb**
Callback to be called when timer expires.

void ***user_data**
User specific opaque data.

Websocket Client API

- *Overview*
- *Websocket Transport*
- *API Reference*

Overview The Websocket client library allows Zephyr to connect to a Websocket server. The Websocket client API can be used directly by application to establish a Websocket connection to server, or it can be used as a transport for other network protocols like MQTT.

See this Websocket Wikipedia article for a detailed overview of how Websocket works.

For more information about the protocol itself, see IETF RFC6455 The WebSocket Protocol.

Websocket Transport The Websocket API allows it to be used as a transport for other high level protocols like MQTT. The Zephyr MQTT client library can be configured to use Websocket transport by enabling `CONFIG_MQTT_LIB_WEBSOCKET` and `CONFIG_WEBSOCKET_CLIENT` Kconfig options.

First a socket needs to be created and connected to the Websocket server:

```
sock = socket(family, SOCK_STREAM, IPPROTO_TCP);
...
ret = connect(sock, addr, addr_len);
...
```

The Websocket transport socket is then created like this:

```
ws_sock = websocket_connect(sock, &config, timeout, user_data);
```

The Websocket socket can then be used to send or receive data, and the Websocket client API will encapsulate the sent or received data to/from Websocket packet payload. Both the `websocket_xxx()` API or normal BSD socket API functions can be used to send and receive application data.

```
ret = websocket_send_msg(ws_sock, buf_to_send, buf_len,
                         WEBSOCKET_OPCODE_DATA_BINARY, true, true,
                         K_FOREVER);
...
ret = send(ws_sock, buf_to_send, buf_len, 0);
```

If normal BSD socket functions are used, then currently only TEXT data is supported. In order to send BINARY data, the *websocket_send_msg()* must be used.

When done, the Websocket transport socket must be closed. User should handle the lifecycle(close/reuse) of tcp socket after websocket_disconnect.

```
ret = close(ws_sock);
or
ret = websocket_disconnect(ws_sock);
```

Related code samples

WebSocket Client
Implement a Websocket client that connects to a Websocket server.

API Reference

group `websocket`

Websocket API.

Defines

`WEBSOCKET_FLAG_FINAL`

Message type values.

Returned in *websocket_recv_msg()* Final frame

`WEBSOCKET_FLAG_TEXT`

Textual data

`WEBSOCKET_FLAG_BINARY`

Binary data

`WEBSOCKET_FLAG_CLOSE`

Closing connection.

`WEBSOCKET_FLAG_PING`

Ping message

`WEBSOCKET_FLAG_PONG`

Pong message

Typedefs

typedef int (*`websocket_connect_cb_t`)(int ws_sock, struct *http_request* *req, void *user_data)

Callback called after Websocket connection is established.

Param ws_sock
Websocket id

Param req
HTTP handshake request

Param user_data
A valid pointer on some user data or NULL

Return
0 if ok, <0 if there is an error and connection should be aborted

Enums

enum `websocket_opcode`

Websocket option codes.

Values:

enumerator `WEBSOCKET_OPCODE_CONTINUE` = 0x00

Message continues.

enumerator `WEBSOCKET_OPCODE_DATA_TEXT` = 0x01

Textual data.

enumerator `WEBSOCKET_OPCODE_DATA_BINARY` = 0x02

Binary data.

enumerator `WEBSOCKET_OPCODE_CLOSE` = 0x08

Closing connection.

enumerator `WEBSOCKET_OPCODE_PING` = 0x09

Ping message.

enumerator `WEBSOCKET_OPCODE_PONG` = 0x0A

Pong message.

Functions

int `websocket_connect`(int http_sock, struct *websocket_request* *req, int32_t timeout, void *user_data)

Connect to a server that provides Websocket service.

The callback is called after connection is established. The returned value is a new socket descriptor that can be used to send / receive data using the BSD socket API.

Parameters

- `http_sock` – Socket id to the server. Note that this socket is used to do HTTP handshakes etc. The actual Websocket connectivity is done via the returned websocket id. Note that the http_sock must not be closed after this function returns as it is used to deliver the Websocket packets to the Websocket server.
- `req` – Websocket request. User should allocate and fill the request data.
- `timeout` – Max timeout to wait for the connection. The timeout value is in milliseconds. Value SYS_FOREVER_MS means to wait forever.
- `user_data` – User specified data that is passed to the callback.

Returns
Websocket id to be used when sending/receiving Websocket data.

int **websocket_send_msg**(int ws_sock, const uint8_t *payload, size_t payload_len, enum *websocket_opcode* opcode, bool mask, bool final, int32_t timeout)

Send websocket msg to peer.

The function will automatically add websocket header to the message.

Parameters

- **ws_sock** – Websocket id returned by *websocket_connect()*.
- **payload** – Websocket data to send.
- **payload_len** – Length of the data to be sent.
- **opcode** – Operation code (text, binary, ping, pong, close)
- **mask** – Mask the data, see RFC 6455 for details
- **final** – Is this final message for this message send. If final == false, then the first message must have valid opcode and subsequent messages must have opcode WEBSOCKET_OPCODE_CONTINUE. If final == true and this is the only message, then opcode should have proper opcode (text or binary) set.
- **timeout** – How long to try to send the message. The value is in milliseconds. Value SYS_FOREVER_MS means to wait forever.

Returns
<0 if error, >=0 amount of bytes sent

int **websocket_recv_msg**(int ws_sock, uint8_t *buf, size_t buf_len, uint32_t *message_type, uint64_t *remaining, int32_t timeout)

Receive websocket msg from peer.

The function will automatically remove websocket header from the message.

Parameters

- **ws_sock** – Websocket id returned by *websocket_connect()*.
- **buf** – Buffer where websocket data is read.
- **buf_len** – Length of the data buffer.
- **message_type** – Type of the message.
- **remaining** – How much there is data left in the message after this read.
- **timeout** – How long to try to receive the message. The value is in milliseconds. Value SYS_FOREVER_MS means to wait forever.

Return values

- **>=0** – amount of bytes received.
- **-EAGAIN** – on timeout.
- **-ENOTCONN** – on socket close.
- **-errno** – other negative errno value in case of failure.

int **websocket_disconnect**(int ws_sock)

Close websocket.

One must call *websocket_connect()* after this call to re-establish the connection.

Parameters

- **ws_sock** – Websocket id returned by *websocket_connect()*.

Returns
<0 if error, 0 the connection was closed successfully

int **websocket_register**(int http_sock, uint8_t *recv_buf, size_t recv_buf_len)

Register a socket as websocket.

This is called by HTTP server when a connection is upgraded to a websocket connection.

Parameters

- **http_sock** – Underlying socket connection socket.
- **recv_buf** – Temporary receive buffer for websocket parsing. This must point to a memory area that is valid for the duration of the whole websocket session.
- **recv_buf_len** – Length of the temporary receive buffer.

Returns
<0 if error, >=0 the actual websocket to be used by application

int **websocket_unregister**(int ws_sock)

Unregister a websocket.

This is called when we no longer need the underlying "real" socket. This will close first the websocket and then the original socket.

Parameters

- **ws_sock** – Websocket connection socket.

Returns
<0 if error, 0 the websocket connection is now fully closed

struct **websocket_request**

#include <*websocket.h*> Websocket client connection request.

This contains all the data that is needed when doing a Websocket connection request.

Public Members

const char ***host**

Host of the Websocket server when doing HTTP handshakes.

const char ***url**

URL of the Websocket.

http_header_cb_t **optional_headers_cb**

User supplied callback function to call when optional headers need to be sent.

This can be NULL, in which case the optional_headers field in *http_request* is used. The idea of this optional_headers callback is to allow user to send more HTTP header data that is practical to store in allocated memory.

const char ****optional_headers**

A NULL terminated list of any optional headers that should be added to the HTTP request.

May be NULL. If the optional_headers_cb is specified, then this field is ignored.

websocket_connect_cb_t **cb**

User supplied callback function to call when a connection is established.

const struct http_parser_settings ***http_cb**

User supplied list of callback functions if the calling application wants to know the parsing status or the HTTP fields during the handshake.

This is optional parameter and normally not needed but is useful if the caller wants to know something about the fields that the server is sending.

uint8_t ***tmp_buf**

User supplied buffer where HTTP connection data is stored.

size_t **tmp_buf_len**

Length of the user supplied temp buffer.

Network Packet Capture

- *Overview*
- *Cooked Mode Capture*
- *Sample usage*
- *API Reference*

Overview The `net_capture` API allows user to monitor the network traffic in one of the Zephyr network interfaces and send that traffic to external system for analysis. The monitoring can be setup either manually using `net-shell` or automatically by using the `net_capture` API.

Cooked Mode Capture If capturing is enabled and configured, the system will automatically capture network traffic for a given network interface. If you would like to capture network data when there is no network interface involved, then you need to use the cooked mode capture API.

In cooked mode capture, arbitrary network packets can be captured and there does not need to be network interface involved. For example low level HDLC packets in PPP can be captured, as the HDLC L2 layer data is stripped away when using the normal network interface based capture. Also CANBUS or Bluetooth network data could be captured although currently there is no support in the network stack to capture those.

The cooked mode capture works like this:

- An `any` network interface is created. It acts as a sink where the cooked mode captured packets are written by the cooked mode capture API.
- A `cooked` virtual network interface is attached on top of this `any` interface.
- The `cooked` interface must be configured to capture certain L2 packet types using the network interface configuration API.
- When cooked mode capture API is used, the caller must specify what is the layer 2 protocol type of the captured data. The cooked mode capture API is then able to determine what to capture when receiving such a L2 packet.

- The network packet capturing infrastructure is then setup so that the cooked interface is marked as captured network interface. The packets received by the cooked interface via the any interface are then automatically placed to the capture IP tunnel and sent to remote host for analysis.

For example, in the sample capture application, these network interfaces are created:

```
Interface any (0x808ab3c) (Dummy) [1]
================================
Virtual interfaces attached to this : 2
Device    : NET_ANY (0x80849a4)

Interface cooked (0x808ac94) (Virtual) [2]
==================================
Virtual name : Cooked mode capture
Attached  : 1 (Dummy / 0x808ab3c)
Device    : NET_COOKED (0x808497c)

Interface eth0 (0x808adec) (Ethernet) [3]
==================================
Virtual interfaces attached to this : 4
Device    : zeth0 (0x80849b8)
IPv6 unicast addresses (max 4):
     fe80::5eff:fe00:53e6 autoconf preferred infinite
     2001:db8::1 manual preferred infinite
IPv4 unicast addresses (max 2):
     192.0.2.1/255.255.255.0 overridable preferred infinite

Interface net0 (0x808af44) (Virtual) [4]
================================
Virtual name : Capture tunnel
Attached  : 3 (Ethernet / 0x808adec)
Device    : IP_TUNNEL0 (0x8084990)
IPv6 unicast addresses (max 4):
     2001:db8:200::1 manual preferred infinite
     fe80::efed:6dff:fef2:b1df autoconf preferred infinite
     fe80::56da:1eff:fe5e:bc02 autoconf preferred infinite
```

In this example, the `192.0.2.2` is the address of the outer end point of the host that terminates the tunnel. Zephyr uses this address to select the internal interface to use for the tunnel. In this example it is interface 3.

The interface 2 is a virtual interface that runs on top of interface 1. The cooked capture packets are written by the capture API to sink interface 1. The packets propagate to interface 2 because it is linked to the first interface. The `net capture enable 2` net-shell command will cause the packets sent to interface 2 to be written to capture interface 4, which in turn then capsulates the packets and tunnels them to peer via the Ethernet interface 3.

The above IP addresses might change if you change the addresses in the sample samples/net/capture/overlay-tunnel.conf file.

Sample usage See net-capture sample application and *Monitor Network Traffic* for details.

Related code samples

Network packet capture
Capture network packets and send them to a remote host via IPIP tunnel.

API Reference

group `net_capture`

Network packet capture support functions.

Functions

int `net_capture_setup`(const char *remote_addr, const char *my_local_addr, const char *peer_addr, const struct *device* **dev)

Setup network packet capturing support.

Parameters

- `remote_addr` – The value tells the tunnel remote/outer endpoint IP address. The IP address can be either IPv4 or IPv6 address. This address is used to select the network interface where the tunnel is created.
- `my_local_addr` – The local/inner IP address of the tunnel. Can contain also port number which is used as UDP source port.
- `peer_addr` – The peer/inner IP address of the tunnel. Can contain also port number which is used as UDP destination port.
- `dev` – Network capture device. This is returned to the caller.

Returns

0 if ok, <0 if network packet capture setup failed

static inline int `net_capture_cleanup`(const struct *device* *dev)

Cleanup network packet capturing support.

This should be called after the capturing is done and resources can be released.

Parameters

- `dev` – Network capture device. User must allocate using the *net_capture_setup()* function.

Returns

0 if ok, <0 if network packet capture cleanup failed

static inline int `net_capture_enable`(const struct *device* *dev, struct *net_if* *iface)

Enable network packet capturing support.

This creates tunnel network interface where all the captured packets are pushed. The captured network packets are placed in UDP packets that are sent to tunnel peer.

Parameters

- `dev` – Network capture device
- `iface` – Network interface we are starting to capture packets.

Returns

0 if ok, <0 if network packet capture enable failed

static inline bool `net_capture_is_enabled`(const struct *device* *dev)

Is network packet capture enabled or disabled.

Parameters

- `dev` – Network capture device. If set to NULL, then the default capture device is used.

Returns

True if enabled, False if network capture is disabled.

static inline int **net_capture_disable**(const struct *device* *dev)

Disable network packet capturing support.

Parameters

- **dev** – Network capture device

Returns

0 if ok, <0 if network packet capture disable failed

Network Buffer Management

Network Buffer

- *Overview*
- *Creating buffers*
- *Common Operations*
- *Reference Counting*
- *API Reference*

Overview Network buffers are a core concept of how the networking stack (as well as the Bluetooth stack) pass data around. The API for them is defined in include/zephyr/net/buf.h:.

Creating buffers Network buffers are created by first defining a pool of them:

```
NET_BUF_POOL_DEFINE(pool_name, buf_count, buf_size, user_data_size, NULL);
```

The pool is a static variable, so if it's needed to be exported to another module a separate pointer is needed.

Once the pool has been defined, buffers can be allocated from it with:

```
buf = net_buf_alloc(&pool_name, timeout);
```

There is no explicit initialization function for the pool or its buffers, rather this is done implicitly as *net_buf_alloc()* gets called.

If there is a need to reserve space in the buffer for protocol headers to be prepended later, it's possible to reserve this headroom with:

```
net_buf_reserve(buf, headroom);
```

In addition to actual protocol data and generic parsing context, network buffers may also contain protocol-specific context, known as user data. Both the maximum data and user data capacity of the buffers is compile-time defined when declaring the buffer pool.

The buffers have native support for being passed through k_fifo kernel objects. This is a very practical feature when the buffers need to be passed from one thread to another. However, since a net_buf may have a fragment chain attached to it, instead of using the *k_fifo_put()* and *k_fifo_get()* APIs, special *net_buf_put()* and *net_buf_get()* APIs must be used when passing buffers through FIFOs. These APIs ensure that the buffer chains stay intact. The same applies for passing buffers through a singly linked list, in which case the *net_buf_slist_put()* and *net_buf_slist_get()* functions must be used instead of *sys_slist_append()* and *sys_slist_get()*.

Common Operations The network buffer API provides some useful helpers for encoding and decoding data in the buffers. To fully understand these helpers it's good to understand the basic names of operations used with them:

Add
: Add data to the end of the buffer. Modifies the data length value while leaving the actual data pointer intact. Requires that there is enough tailroom in the buffer. Some examples of APIs for adding data:

```
void *net_buf_add(struct net_buf *buf, size_t len);
void *net_buf_add_mem(struct net_buf *buf, const void *mem, size_t len);
uint8_t *net_buf_add_u8(struct net_buf *buf, uint8_t value);
void net_buf_add_le16(struct net_buf *buf, uint16_t value);
void net_buf_add_le32(struct net_buf *buf, uint32_t value);
```

Remove
: Remove data from the end of the buffer. Modifies the data length value while leaving the actual data pointer intact. Some examples of APIs for removing data:

```
void *net_buf_remove_mem(struct net_buf *buf, size_t len);
uint8_t net_buf_remove_u8(struct net_buf *buf);
uint16_t net_buf_remove_le16(struct net_buf *buf);
uint32_t net_buf_remove_le32(struct net_buf *buf);
```

Push
: Prepend data to the beginning of the buffer. Modifies both the data length value as well as the data pointer. Requires that there is enough headroom in the buffer. Some examples of APIs for pushing data:

```
void *net_buf_push(struct net_buf *buf, size_t len);
void *net_buf_push_mem(struct net_buf *buf, const void *mem, size_t len);
void net_buf_push_u8(struct net_buf *buf, uint8_t value);
void net_buf_push_le16(struct net_buf *buf, uint16_t value);
```

Pull
: Remove data from the beginning of the buffer. Modifies both the data length value as well as the data pointer. Some examples of APIs for pulling data:

```
void *net_buf_pull(struct net_buf *buf, size_t len);
void *net_buf_pull_mem(struct net_buf *buf, size_t len);
uint8_t net_buf_pull_u8(struct net_buf *buf);
uint16_t net_buf_pull_le16(struct net_buf *buf);
uint32_t net_buf_pull_le32(struct net_buf *buf);
```

The Add and Push operations are used when encoding data into the buffer, whereas the Remove and Pull operations are used when decoding data from a buffer.

Reference Counting Each network buffer is reference counted. The buffer is initially acquired from a free buffers pool by calling *net_buf_alloc()*, resulting in a buffer with reference count 1. The reference count can be incremented with *net_buf_ref()* or decremented with *net_buf_unref()*. When the count drops to zero the buffer is automatically placed back to the free buffers pool.

API Reference

group `net_buf`

Network buffer library.

Defines

NET_BUF_SIMPLE_DEFINE(_name, _size)

Define a *net_buf_simple* stack variable.

This is a helper macro which is used to define a *net_buf_simple* object on the stack.

Parameters

- **_name** – Name of the *net_buf_simple* object.
- **_size** – Maximum data storage for the buffer.

NET_BUF_SIMPLE_DEFINE_STATIC(_name, _size)

Define a static *net_buf_simple* variable.

This is a helper macro which is used to define a static *net_buf_simple* object.

Parameters

- **_name** – Name of the *net_buf_simple* object.
- **_size** – Maximum data storage for the buffer.

NET_BUF_SIMPLE(_size)

Define a *net_buf_simple* stack variable and get a pointer to it.

This is a helper macro which is used to define a *net_buf_simple* object on the stack and the get a pointer to it as follows:

struct *net_buf_simple* *my_buf = *NET_BUF_SIMPLE(10)*;

After creating the object it needs to be initialized by calling *net_buf_simple_init()*.

Parameters

- **_size** – Maximum data storage for the buffer.

Returns

Pointer to stack-allocated *net_buf_simple* object.

NET_BUF_EXTERNAL_DATA

Flag indicating that the buffer's associated data pointer, points to externally allocated memory.

Therefore once ref goes down to zero, the pointed data will not need to be deallocated. This never needs to be explicitly set or unset by the *net_buf* API user. Such *net_buf* is exclusively instantiated via *net_buf_alloc_with_data()* function. Reference count mechanism however will behave the same way, and ref count going to 0 will free the *net_buf* but no the data pointer in it.

NET_BUF_POOL_HEAP_DEFINE(_name, _count, _ud_size, _destroy)

Define a new pool for buffers using the heap for the data.

Defines a *net_buf_pool* struct and the necessary memory storage (array of structs) for the needed amount of buffers. After this, the buffers can be accessed from the pool through net_buf_alloc. The pool is defined as a static variable, so if it needs to be exported outside the current module this needs to happen with the help of a separate pointer rather than an extern declaration.

The data payload of the buffers will be allocated from the heap using k_malloc, so CONFIG_HEAP_MEM_POOL_SIZE must be set to a positive value. This kind of pool does not support blocking on the data allocation, so the timeout passed to net_buf_alloc will be always treated as K_NO_WAIT when trying to allocate the data. This means that allocation failures, i.e. NULL returns, must always be handled cleanly.

If provided with a custom destroy callback, this callback is responsible for eventually calling *net_buf_destroy()* to complete the process of returning the buffer to the pool.

Parameters

- **_name** – Name of the pool variable.
- **_count** – Number of buffers in the pool.
- **_ud_size** – User data space to reserve per buffer.
- **_destroy** – Optional destroy callback when buffer is freed.

NET_BUF_POOL_FIXED_DEFINE(_name, _count, _data_size, _ud_size, _destroy)

Define a new pool for buffers based on fixed-size data.

Defines a *net_buf_pool* struct and the necessary memory storage (array of structs) for the needed amount of buffers. After this, the buffers can be accessed from the pool through net_buf_alloc. The pool is defined as a static variable, so if it needs to be exported outside the current module this needs to happen with the help of a separate pointer rather than an extern declaration.

The data payload of the buffers will be allocated from a byte array of fixed sized chunks. This kind of pool does not support blocking on the data allocation, so the timeout passed to net_buf_alloc will be always treated as K_NO_WAIT when trying to allocate the data. This means that allocation failures, i.e. NULL returns, must always be handled cleanly.

If provided with a custom destroy callback, this callback is responsible for eventually calling *net_buf_destroy()* to complete the process of returning the buffer to the pool.

Parameters

- **_name** – Name of the pool variable.
- **_count** – Number of buffers in the pool.
- **_data_size** – Maximum data payload per buffer.
- **_ud_size** – User data space to reserve per buffer.
- **_destroy** – Optional destroy callback when buffer is freed.

NET_BUF_POOL_VAR_DEFINE(_name, _count, _data_size, _ud_size, _destroy)

Define a new pool for buffers with variable size payloads.

Defines a *net_buf_pool* struct and the necessary memory storage (array of structs) for the needed amount of buffers. After this, the buffers can be accessed from the pool through net_buf_alloc. The pool is defined as a static variable, so if it needs to be exported outside the current module this needs to happen with the help of a separate pointer rather than an extern declaration.

The data payload of the buffers will be based on a memory pool from which variable size payloads may be allocated.

If provided with a custom destroy callback, this callback is responsible for eventually calling *net_buf_destroy()* to complete the process of returning the buffer to the pool.

Parameters

- **_name** – Name of the pool variable.
- **_count** – Number of buffers in the pool.
- **_data_size** – Total amount of memory available for data payloads.
- **_ud_size** – User data space to reserve per buffer.
- **_destroy** – Optional destroy callback when buffer is freed.

NET_BUF_POOL_DEFINE(_name, _count, _size, _ud_size, _destroy)

Define a new pool for buffers.

Defines a *net_buf_pool* struct and the necessary memory storage (array of structs) for the needed amount of buffers. After this,the buffers can be accessed from the pool through net_buf_alloc. The pool is defined as a static variable, so if it needs to be exported outside the current module this needs to happen with the help of a separate pointer rather than an extern declaration.

If provided with a custom destroy callback this callback is responsible for eventually calling *net_buf_destroy()* to complete the process of returning the buffer to the pool.

Parameters

- **_name** – Name of the pool variable.
- **_count** – Number of buffers in the pool.
- **_size** – Maximum data size for each buffer.
- **_ud_size** – Amount of user data space to reserve.
- **_destroy** – Optional destroy callback when buffer is freed.

Typedefs

typedef struct *net_buf* *(***net_buf_allocator_cb**)(*k_timeout_t* timeout, void *user_data)

Network buffer allocator callback.

The allocator callback is called when net_buf_append_bytes needs to allocate a new *net_buf*.

Param timeout
Affects the action taken should the net buf pool be empty. If K_NO_WAIT, then return immediately. If K_FOREVER, then wait as long as necessary. Otherwise, wait until the specified timeout.

Param user_data
The user data given in net_buf_append_bytes call.

Return
pointer to allocated *net_buf* or NULL on error.

Functions

static inline void **net_buf_simple_init**(struct *net_buf_simple* *buf, size_t reserve_head)

Initialize a *net_buf_simple* object.

This needs to be called after creating a *net_buf_simple* object using the NET_BUF_SIMPLE macro.

Parameters

- **buf** – Buffer to initialize.
- **reserve_head** – Headroom to reserve.

void **net_buf_simple_init_with_data**(struct *net_buf_simple* *buf, void *data, size_t size)

Initialize a *net_buf_simple* object with data.

Initialized buffer object with external data.

Parameters

- **buf** – Buffer to initialize.
- **data** – External data pointer
- **size** – Amount of data the pointed data buffer if able to fit.

static inline void **net_buf_simple_reset**(struct *net_buf_simple* *buf)

Reset buffer.

Reset buffer data so it can be reused for other purposes.

Parameters

- **buf** – Buffer to reset.

void **net_buf_simple_clone**(const struct *net_buf_simple* *original, struct *net_buf_simple* *clone)

Clone buffer state, using the same data buffer.

Initializes a buffer to point to the same data as an existing buffer. Allows operations on the same data without altering the length and offset of the original.

Parameters

- **original** – Buffer to clone.
- **clone** – The new clone.

void ***net_buf_simple_add**(struct *net_buf_simple* *buf, size_t len)

Prepare data to be added at the end of the buffer.

Increments the data length of a buffer to account for more data at the end.

Parameters

- **buf** – Buffer to update.
- **len** – Number of bytes to increment the length with.

Returns

The original tail of the buffer.

void ***net_buf_simple_add_mem**(struct *net_buf_simple* *buf, const void *mem, size_t len)

Copy given number of bytes from memory to the end of the buffer.

Increments the data length of the buffer to account for more data at the end.

Parameters

- **buf** – Buffer to update.
- **mem** – Location of data to be added.
- **len** – Length of data to be added

Returns

The original tail of the buffer.

uint8_t ***net_buf_simple_add_u8**(struct *net_buf_simple* *buf, uint8_t val)

Add (8-bit) byte at the end of the buffer.

Increments the data length of the buffer to account for more data at the end.

Parameters

- **buf** – Buffer to update.
- **val** – byte value to be added.

Returns

Pointer to the value added

void **net_buf_simple_add_le16**(struct *net_buf_simple* *buf, uint16_t val)

Add 16-bit value at the end of the buffer.

Adds 16-bit value in little endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 16-bit value to be added.

void **net_buf_simple_add_be16**(struct *net_buf_simple* *buf, uint16_t val)

Add 16-bit value at the end of the buffer.

Adds 16-bit value in big endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 16-bit value to be added.

void **net_buf_simple_add_le24**(struct *net_buf_simple* *buf, uint32_t val)

Add 24-bit value at the end of the buffer.

Adds 24-bit value in little endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 24-bit value to be added.

void **net_buf_simple_add_be24**(struct *net_buf_simple* *buf, uint32_t val)

Add 24-bit value at the end of the buffer.

Adds 24-bit value in big endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 24-bit value to be added.

void **net_buf_simple_add_le32**(struct *net_buf_simple* *buf, uint32_t val)

Add 32-bit value at the end of the buffer.

Adds 32-bit value in little endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 32-bit value to be added.

void **net_buf_simple_add_be32**(struct *net_buf_simple* *buf, uint32_t val)

Add 32-bit value at the end of the buffer.

Adds 32-bit value in big endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 32-bit value to be added.

void **net_buf_simple_add_le40**(struct *net_buf_simple* *buf, uint64_t val)

Add 40-bit value at the end of the buffer.

Adds 40-bit value in little endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 40-bit value to be added.

void **net_buf_simple_add_be40**(struct *net_buf_simple* *buf, uint64_t val)

Add 40-bit value at the end of the buffer.

Adds 40-bit value in big endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 40-bit value to be added.

void **net_buf_simple_add_le48**(struct *net_buf_simple* *buf, uint64_t val)

Add 48-bit value at the end of the buffer.

Adds 48-bit value in little endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 48-bit value to be added.

void **net_buf_simple_add_be48**(struct *net_buf_simple* *buf, uint64_t val)

Add 48-bit value at the end of the buffer.

Adds 48-bit value in big endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 48-bit value to be added.

void **net_buf_simple_add_le64**(struct *net_buf_simple* *buf, uint64_t val)

Add 64-bit value at the end of the buffer.

Adds 64-bit value in little endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 64-bit value to be added.

void **net_buf_simple_add_be64**(struct *net_buf_simple* *buf, uint64_t val)

Add 64-bit value at the end of the buffer.

Adds 64-bit value in big endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 64-bit value to be added.

void *net_buf_simple_remove_mem(struct *net_buf_simple* *buf, size_t len)

Remove data from the end of the buffer.

Removes data from the end of the buffer by modifying the buffer length.

Parameters

- buf – Buffer to update.
- len – Number of bytes to remove.

Returns
New end of the buffer data.

uint8_t net_buf_simple_remove_u8(struct *net_buf_simple* *buf)

Remove a 8-bit value from the end of the buffer.

Same idea as with *net_buf_simple_remove_mem()*, but a helper for operating on 8-bit values.

Parameters

- buf – A valid pointer on a buffer.

Returns
The 8-bit removed value

uint16_t net_buf_simple_remove_le16(struct *net_buf_simple* *buf)

Remove and convert 16 bits from the end of the buffer.

Same idea as with *net_buf_simple_remove_mem()*, but a helper for operating on 16-bit little endian data.

Parameters

- buf – A valid pointer on a buffer.

Returns
16-bit value converted from little endian to host endian.

uint16_t net_buf_simple_remove_be16(struct *net_buf_simple* *buf)

Remove and convert 16 bits from the end of the buffer.

Same idea as with *net_buf_simple_remove_mem()*, but a helper for operating on 16-bit big endian data.

Parameters

- buf – A valid pointer on a buffer.

Returns
16-bit value converted from big endian to host endian.

uint32_t net_buf_simple_remove_le24(struct *net_buf_simple* *buf)

Remove and convert 24 bits from the end of the buffer.

Same idea as with *net_buf_simple_remove_mem()*, but a helper for operating on 24-bit little endian data.

Parameters

- buf – A valid pointer on a buffer.

Returns
24-bit value converted from little endian to host endian.

uint32_t **net_buf_simple_remove_be24**(struct *net_buf_simple* *buf)

Remove and convert 24 bits from the end of the buffer.

Same idea as with *net_buf_simple_remove_mem()*, but a helper for operating on 24-bit big endian data.

Parameters

- **buf** – A valid pointer on a buffer.

Returns
24-bit value converted from big endian to host endian.

uint32_t **net_buf_simple_remove_le32**(struct *net_buf_simple* *buf)

Remove and convert 32 bits from the end of the buffer.

Same idea as with *net_buf_simple_remove_mem()*, but a helper for operating on 32-bit little endian data.

Parameters

- **buf** – A valid pointer on a buffer.

Returns
32-bit value converted from little endian to host endian.

uint32_t **net_buf_simple_remove_be32**(struct *net_buf_simple* *buf)

Remove and convert 32 bits from the end of the buffer.

Same idea as with *net_buf_simple_remove_mem()*, but a helper for operating on 32-bit big endian data.

Parameters

- **buf** – A valid pointer on a buffer.

Returns
32-bit value converted from big endian to host endian.

uint64_t **net_buf_simple_remove_le40**(struct *net_buf_simple* *buf)

Remove and convert 40 bits from the end of the buffer.

Same idea as with *net_buf_simple_remove_mem()*, but a helper for operating on 40-bit little endian data.

Parameters

- **buf** – A valid pointer on a buffer.

Returns
40-bit value converted from little endian to host endian.

uint64_t **net_buf_simple_remove_be40**(struct *net_buf_simple* *buf)

Remove and convert 40 bits from the end of the buffer.

Same idea as with *net_buf_simple_remove_mem()*, but a helper for operating on 40-bit big endian data.

Parameters

- **buf** – A valid pointer on a buffer.

Returns
40-bit value converted from big endian to host endian.

uint64_t **net_buf_simple_remove_le48**(struct *net_buf_simple* *buf)

Remove and convert 48 bits from the end of the buffer.

Same idea as with *net_buf_simple_remove_mem()*, but a helper for operating on 48-bit little endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns
48-bit value converted from little endian to host endian.

uint64_t **net_buf_simple_remove_be48**(struct *net_buf_simple* *buf)

Remove and convert 48 bits from the end of the buffer.

Same idea as with *net_buf_simple_remove_mem()*, but a helper for operating on 48-bit big endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns
48-bit value converted from big endian to host endian.

uint64_t **net_buf_simple_remove_le64**(struct *net_buf_simple* *buf)

Remove and convert 64 bits from the end of the buffer.

Same idea as with *net_buf_simple_remove_mem()*, but a helper for operating on 64-bit little endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns
64-bit value converted from little endian to host endian.

uint64_t **net_buf_simple_remove_be64**(struct *net_buf_simple* *buf)

Remove and convert 64 bits from the end of the buffer.

Same idea as with *net_buf_simple_remove_mem()*, but a helper for operating on 64-bit big endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns
64-bit value converted from big endian to host endian.

void *__net_buf_simple_push__(struct *net_buf_simple* *buf, size_t len)

Prepare data to be added to the start of the buffer.

Modifies the data pointer and buffer length to account for more data in the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `len` – Number of bytes to add to the beginning.

Returns
The new beginning of the buffer data.

void *__net_buf_simple_push_mem__(struct *net_buf_simple* *buf, const void *mem, size_t len)

Copy given number of bytes from memory to the start of the buffer.

Modifies the data pointer and buffer length to account for more data in the beginning of the buffer.

Parameters

- `buf` – Buffer to update.

- `mem` – Location of data to be added.
- `len` – Length of data to be added.

Returns
The new beginning of the buffer data.

void `net_buf_simple_push_le16`(struct *net_buf_simple* *buf, uint16_t val)
Push 16-bit value to the beginning of the buffer.

Adds 16-bit value in little endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 16-bit value to be pushed to the buffer.

void `net_buf_simple_push_be16`(struct *net_buf_simple* *buf, uint16_t val)
Push 16-bit value to the beginning of the buffer.

Adds 16-bit value in big endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 16-bit value to be pushed to the buffer.

void `net_buf_simple_push_u8`(struct *net_buf_simple* *buf, uint8_t val)
Push 8-bit value to the beginning of the buffer.

Adds 8-bit value the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 8-bit value to be pushed to the buffer.

void `net_buf_simple_push_le24`(struct *net_buf_simple* *buf, uint32_t val)
Push 24-bit value to the beginning of the buffer.

Adds 24-bit value in little endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 24-bit value to be pushed to the buffer.

void `net_buf_simple_push_be24`(struct *net_buf_simple* *buf, uint32_t val)
Push 24-bit value to the beginning of the buffer.

Adds 24-bit value in big endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 24-bit value to be pushed to the buffer.

void `net_buf_simple_push_le32`(struct *net_buf_simple* *buf, uint32_t val)
Push 32-bit value to the beginning of the buffer.

Adds 32-bit value in little endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 32-bit value to be pushed to the buffer.

void **net_buf_simple_push_be32**(struct *net_buf_simple* *buf, uint32_t val)

Push 32-bit value to the beginning of the buffer.

Adds 32-bit value in big endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 32-bit value to be pushed to the buffer.

void **net_buf_simple_push_le40**(struct *net_buf_simple* *buf, uint64_t val)

Push 40-bit value to the beginning of the buffer.

Adds 40-bit value in little endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 40-bit value to be pushed to the buffer.

void **net_buf_simple_push_be40**(struct *net_buf_simple* *buf, uint64_t val)

Push 40-bit value to the beginning of the buffer.

Adds 40-bit value in big endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 40-bit value to be pushed to the buffer.

void **net_buf_simple_push_le48**(struct *net_buf_simple* *buf, uint64_t val)

Push 48-bit value to the beginning of the buffer.

Adds 48-bit value in little endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 48-bit value to be pushed to the buffer.

void **net_buf_simple_push_be48**(struct *net_buf_simple* *buf, uint64_t val)

Push 48-bit value to the beginning of the buffer.

Adds 48-bit value in big endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 48-bit value to be pushed to the buffer.

void **net_buf_simple_push_le64**(struct *net_buf_simple* *buf, uint64_t val)

Push 64-bit value to the beginning of the buffer.

Adds 64-bit value in little endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 64-bit value to be pushed to the buffer.

void **net_buf_simple_push_be64**(struct *net_buf_simple* *buf, uint64_t val)

Push 64-bit value to the beginning of the buffer.

Adds 64-bit value in big endian format to the beginning of the buffer.

Parameters

- **buf** – Buffer to update.
- **val** – 64-bit value to be pushed to the buffer.

void *net_buf_simple_pull(struct *net_buf_simple* *buf, size_t len)

Remove data from the beginning of the buffer.

Removes data from the beginning of the buffer by modifying the data pointer and buffer length.

Parameters

- **buf** – Buffer to update.
- **len** – Number of bytes to remove.

Returns

New beginning of the buffer data.

void *net_buf_simple_pull_mem(struct *net_buf_simple* *buf, size_t len)

Remove data from the beginning of the buffer.

Removes data from the beginning of the buffer by modifying the data pointer and buffer length.

Parameters

- **buf** – Buffer to update.
- **len** – Number of bytes to remove.

Returns

Pointer to the old location of the buffer data.

uint8_t net_buf_simple_pull_u8(struct *net_buf_simple* *buf)

Remove a 8-bit value from the beginning of the buffer.

Same idea as with *net_buf_simple_pull()*, but a helper for operating on 8-bit values.

Parameters

- **buf** – A valid pointer on a buffer.

Returns

The 8-bit removed value

uint16_t net_buf_simple_pull_le16(struct *net_buf_simple* *buf)

Remove and convert 16 bits from the beginning of the buffer.

Same idea as with *net_buf_simple_pull()*, but a helper for operating on 16-bit little endian data.

Parameters

- **buf** – A valid pointer on a buffer.

Returns

16-bit value converted from little endian to host endian.

uint16_t net_buf_simple_pull_be16(struct *net_buf_simple* *buf)

Remove and convert 16 bits from the beginning of the buffer.

Same idea as with *net_buf_simple_pull()*, but a helper for operating on 16-bit big endian data.

Parameters

- **buf** – A valid pointer on a buffer.

Returns

16-bit value converted from big endian to host endian.

uint32_t **net_buf_simple_pull_le24**(struct *net_buf_simple* *buf)

Remove and convert 24 bits from the beginning of the buffer.

Same idea as with *net_buf_simple_pull()*, but a helper for operating on 24-bit little endian data.

Parameters

- **buf** – A valid pointer on a buffer.

Returns
24-bit value converted from little endian to host endian.

uint32_t **net_buf_simple_pull_be24**(struct *net_buf_simple* *buf)

Remove and convert 24 bits from the beginning of the buffer.

Same idea as with *net_buf_simple_pull()*, but a helper for operating on 24-bit big endian data.

Parameters

- **buf** – A valid pointer on a buffer.

Returns
24-bit value converted from big endian to host endian.

uint32_t **net_buf_simple_pull_le32**(struct *net_buf_simple* *buf)

Remove and convert 32 bits from the beginning of the buffer.

Same idea as with *net_buf_simple_pull()*, but a helper for operating on 32-bit little endian data.

Parameters

- **buf** – A valid pointer on a buffer.

Returns
32-bit value converted from little endian to host endian.

uint32_t **net_buf_simple_pull_be32**(struct *net_buf_simple* *buf)

Remove and convert 32 bits from the beginning of the buffer.

Same idea as with *net_buf_simple_pull()*, but a helper for operating on 32-bit big endian data.

Parameters

- **buf** – A valid pointer on a buffer.

Returns
32-bit value converted from big endian to host endian.

uint64_t **net_buf_simple_pull_le40**(struct *net_buf_simple* *buf)

Remove and convert 40 bits from the beginning of the buffer.

Same idea as with *net_buf_simple_pull()*, but a helper for operating on 40-bit little endian data.

Parameters

- **buf** – A valid pointer on a buffer.

Returns
40-bit value converted from little endian to host endian.

uint64_t **net_buf_simple_pull_be40**(struct *net_buf_simple* *buf)

Remove and convert 40 bits from the beginning of the buffer.

Same idea as with *net_buf_simple_pull()*, but a helper for operating on 40-bit big endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns
40-bit value converted from big endian to host endian.

uint64_t `net_buf_simple_pull_le48`(struct *net_buf_simple* *buf)

Remove and convert 48 bits from the beginning of the buffer.

Same idea as with *net_buf_simple_pull()*, but a helper for operating on 48-bit little endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns
48-bit value converted from little endian to host endian.

uint64_t `net_buf_simple_pull_be48`(struct *net_buf_simple* *buf)

Remove and convert 48 bits from the beginning of the buffer.

Same idea as with *net_buf_simple_pull()*, but a helper for operating on 48-bit big endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns
48-bit value converted from big endian to host endian.

uint64_t `net_buf_simple_pull_le64`(struct *net_buf_simple* *buf)

Remove and convert 64 bits from the beginning of the buffer.

Same idea as with *net_buf_simple_pull()*, but a helper for operating on 64-bit little endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns
64-bit value converted from little endian to host endian.

uint64_t `net_buf_simple_pull_be64`(struct *net_buf_simple* *buf)

Remove and convert 64 bits from the beginning of the buffer.

Same idea as with *net_buf_simple_pull()*, but a helper for operating on 64-bit big endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns
64-bit value converted from big endian to host endian.

static inline uint8_t *`net_buf_simple_tail`(const struct *net_buf_simple* *buf)

Get the tail pointer for a buffer.

Get a pointer to the end of the data in a buffer.

Parameters

- `buf` – Buffer.

Returns
Tail pointer for the buffer.

size_t **net_buf_simple_headroom**(const struct *net_buf_simple* *buf)

Check buffer headroom.

Check how much free space there is in the beginning of the buffer.

buf A valid pointer on a buffer

Returns
Number of bytes available in the beginning of the buffer.

size_t **net_buf_simple_tailroom**(const struct *net_buf_simple* *buf)

Check buffer tailroom.

Check how much free space there is at the end of the buffer.

Parameters

- **buf** – A valid pointer on a buffer

Returns
Number of bytes available at the end of the buffer.

uint16_t **net_buf_simple_max_len**(const struct *net_buf_simple* *buf)

Check maximum *net_buf_simple::len* value.

This value is depending on the number of bytes being reserved as headroom.

Parameters

- **buf** – A valid pointer on a buffer

Returns
Number of bytes usable behind the *net_buf_simple::data* pointer.

static inline void **net_buf_simple_save**(const struct *net_buf_simple* *buf, struct *net_buf_simple_state* *state)

Save the parsing state of a buffer.

Saves the parsing state of a buffer so it can be restored later.

Parameters

- **buf** – Buffer from which the state should be saved.
- **state** – Storage for the state.

static inline void **net_buf_simple_restore**(struct *net_buf_simple* *buf, struct *net_buf_simple_state* *state)

Restore the parsing state of a buffer.

Restores the parsing state of a buffer from a state previously stored by *net_buf_simple_save()*.

Parameters

- **buf** – Buffer to which the state should be restored.
- **state** – Stored state.

struct *net_buf_pool* ***net_buf_pool_get**(int id)

Looks up a pool based on its ID.

Parameters

- **id** – Pool ID (e.g. from buf->pool_id).

Returns
Pointer to pool.

int **net_buf_id**(const struct *net_buf* *buf)

Get a zero-based index for a buffer.

This function will translate a buffer into a zero-based index, based on its placement in its buffer pool. This can be useful if you want to associate an external array of metadata contexts with the buffers of a pool.

Parameters

- **buf** – Network buffer.

Returns

Zero-based index for the buffer.

struct *net_buf* ***net_buf_alloc_fixed**(struct *net_buf_pool* *pool, *k_timeout_t* timeout)

Allocate a new fixed buffer from a pool.

Note

Some types of data allocators do not support blocking (such as the HEAP type). In this case it's still possible for *net_buf_alloc()* to fail (return NULL) even if it was given K_FOREVER.

Note

The timeout value will be overridden to K_NO_WAIT if called from the system workqueue.

Parameters

- **pool** – Which pool to allocate the buffer from.
- **timeout** – Affects the action taken should the pool be empty. If K_NO_WAIT, then return immediately. If K_FOREVER, then wait as long as necessary. Otherwise, wait until the specified timeout.

Returns

New buffer or NULL if out of buffers.

static inline struct *net_buf* ***net_buf_alloc**(struct *net_buf_pool* *pool, *k_timeout_t* timeout)

Note

Some types of data allocators do not support blocking (such as the HEAP type). In this case it's still possible for *net_buf_alloc()* to fail (return NULL) even if it was given K_FOREVER.

Note

The timeout value will be overridden to K_NO_WAIT if called from the system workqueue.

Parameters

- **pool** – Which pool to allocate the buffer from.

- **timeout** – Affects the action taken should the pool be empty. If K_NO_WAIT, then return immediately. If K_FOREVER, then wait as long as necessary. Otherwise, wait until the specified timeout.

Returns
New buffer or NULL if out of buffers.

struct *net_buf* ***net_buf_alloc_len**(struct *net_buf_pool* *pool, size_t size, *k_timeout_t* timeout)

Allocate a new variable length buffer from a pool.

Note

Some types of data allocators do not support blocking (such as the HEAP type). In this case it's still possible for *net_buf_alloc()* to fail (return NULL) even if it was given K_FOREVER.

Note

The timeout value will be overridden to K_NO_WAIT if called from the system workqueue.

Parameters

- **pool** – Which pool to allocate the buffer from.
- **size** – Amount of data the buffer must be able to fit.
- **timeout** – Affects the action taken should the pool be empty. If K_NO_WAIT, then return immediately. If K_FOREVER, then wait as long as necessary. Otherwise, wait until the specified timeout.

Returns
New buffer or NULL if out of buffers.

struct *net_buf* ***net_buf_alloc_with_data**(struct *net_buf_pool* *pool, void *data, size_t size, *k_timeout_t* timeout)

Allocate a new buffer from a pool but with external data pointer.

Allocate a new buffer from a pool, where the data pointer comes from the user and not from the pool.

Note

Some types of data allocators do not support blocking (such as the HEAP type). In this case it's still possible for *net_buf_alloc()* to fail (return NULL) even if it was given K_FOREVER.

Note

The timeout value will be overridden to K_NO_WAIT if called from the system workqueue.

Parameters

- **pool** – Which pool to allocate the buffer from.

- **data** – External data pointer
- **size** – Amount of data the pointed data buffer if able to fit.
- **timeout** – Affects the action taken should the pool be empty. If K_NO_WAIT, then return immediately. If K_FOREVER, then wait as long as necessary. Otherwise, wait until the specified timeout.

Returns
New buffer or NULL if out of buffers.

struct *net_buf* ***net_buf_get**(struct k_fifo *fifo, *k_timeout_t* timeout)

Get a buffer from a FIFO.

This function is NOT thread-safe if the buffers in the FIFO contain fragments.

Parameters

- **fifo** – Which FIFO to take the buffer from.
- **timeout** – Affects the action taken should the FIFO be empty. If K_NO_WAIT, then return immediately. If K_FOREVER, then wait as long as necessary. Otherwise, wait until the specified timeout.

Returns
New buffer or NULL if the FIFO is empty.

static inline void **net_buf_destroy**(struct *net_buf* *buf)

Destroy buffer from custom destroy callback.

This helper is only intended to be used from custom destroy callbacks. If no custom destroy callback is given to NET_BUF_POOL_*_DEFINE() then there is no need to use this API.

Parameters

- **buf** – Buffer to destroy.

void **net_buf_reset**(struct *net_buf* *buf)

Reset buffer.

Reset buffer data and flags so it can be reused for other purposes.

Parameters

- **buf** – Buffer to reset.

void **net_buf_simple_reserve**(struct *net_buf_simple* *buf, size_t reserve)

Initialize buffer with the given headroom.

The buffer is not expected to contain any data when this API is called.

Parameters

- **buf** – Buffer to initialize.
- **reserve** – How much headroom to reserve.

void **net_buf_slist_put**(*sys_slist_t* *list, struct *net_buf* *buf)

Put a buffer into a list.

If the buffer contains follow-up fragments this function will take care of inserting them as well into the list.

Parameters

- **list** – Which list to append the buffer to.
- **buf** – Buffer.

struct *net_buf* *net_buf_slist_get(*sys_slist_t* *list)

Get a buffer from a list.

If the buffer had any fragments, these will automatically be recovered from the list as well and be placed to the buffer's fragment list.

Parameters

- `list` – Which list to take the buffer from.

Returns
New buffer or NULL if the FIFO is empty.

void net_buf_put(struct k_fifo *fifo, struct *net_buf* *buf)

Put a buffer to the end of a FIFO.

If the buffer contains follow-up fragments this function will take care of inserting them as well into the FIFO.

Parameters

- `fifo` – Which FIFO to put the buffer to.
- `buf` – Buffer.

void net_buf_unref(struct *net_buf* *buf)

Decrements the reference count of a buffer.

The buffer is put back into the pool if the reference count reaches zero.

Parameters

- `buf` – A valid pointer on a buffer

struct *net_buf* *net_buf_ref(struct *net_buf* *buf)

Increment the reference count of a buffer.

Parameters

- `buf` – A valid pointer on a buffer

Returns
the buffer newly referenced

struct *net_buf* *net_buf_clone(struct *net_buf* *buf, *k_timeout_t* timeout)

Clone buffer.

Duplicate given buffer including any (user) data and headers currently stored.

Parameters

- `buf` – A valid pointer on a buffer
- `timeout` – Affects the action taken should the pool be empty. If K_NO_WAIT, then return immediately. If K_FOREVER, then wait as long as necessary. Otherwise, wait until the specified timeout.

Returns
Cloned buffer or NULL if out of buffers.

static inline void *net_buf_user_data(const struct *net_buf* *buf)

Get a pointer to the user data of a buffer.

Parameters

- `buf` – A valid pointer on a buffer

Returns
Pointer to the user data of the buffer.

int **net_buf_user_data_copy**(struct *net_buf* *dst, const struct *net_buf* *src)

Copy user data from one to another buffer.

Parameters

- **dst** – A valid pointer to a buffer gettings its user data overwritten.
- **src** – A valid pointer to a buffer gettings its user data copied. User data size must be equal to or exceed *dst*.

Returns

0 on success or negative error number on failure.

static inline void **net_buf_reserve**(struct *net_buf* *buf, size_t reserve)

Initialize buffer with the given headroom.

The buffer is not expected to contain any data when this API is called.

Parameters

- **buf** – Buffer to initialize.
- **reserve** – How much headroom to reserve.

static inline void ***net_buf_add**(struct *net_buf* *buf, size_t len)

Prepare data to be added at the end of the buffer.

Increments the data length of a buffer to account for more data at the end.

Parameters

- **buf** – Buffer to update.
- **len** – Number of bytes to increment the length with.

Returns

The original tail of the buffer.

static inline void ***net_buf_add_mem**(struct *net_buf* *buf, const void *mem, size_t len)

Copies the given number of bytes to the end of the buffer.

Increments the data length of the buffer to account for more data at the end.

Parameters

- **buf** – Buffer to update.
- **mem** – Location of data to be added.
- **len** – Length of data to be added

Returns

The original tail of the buffer.

static inline uint8_t ***net_buf_add_u8**(struct *net_buf* *buf, uint8_t val)

Add (8-bit) byte at the end of the buffer.

Increments the data length of the buffer to account for more data at the end.

Parameters

- **buf** – Buffer to update.
- **val** – byte value to be added.

Returns

Pointer to the value added

static inline void **net_buf_add_le16**(struct *net_buf* *buf, uint16_t val)

Add 16-bit value at the end of the buffer.

Adds 16-bit value in little endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 16-bit value to be added.

static inline void **net_buf_add_be16**(struct *net_buf* *buf, uint16_t val)

Add 16-bit value at the end of the buffer.

Adds 16-bit value in big endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 16-bit value to be added.

static inline void **net_buf_add_le24**(struct *net_buf* *buf, uint32_t val)

Add 24-bit value at the end of the buffer.

Adds 24-bit value in little endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 24-bit value to be added.

static inline void **net_buf_add_be24**(struct *net_buf* *buf, uint32_t val)

Add 24-bit value at the end of the buffer.

Adds 24-bit value in big endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 24-bit value to be added.

static inline void **net_buf_add_le32**(struct *net_buf* *buf, uint32_t val)

Add 32-bit value at the end of the buffer.

Adds 32-bit value in little endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 32-bit value to be added.

static inline void **net_buf_add_be32**(struct *net_buf* *buf, uint32_t val)

Add 32-bit value at the end of the buffer.

Adds 32-bit value in big endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 32-bit value to be added.

static inline void **net_buf_add_le40**(struct *net_buf* *buf, uint64_t val)

Add 40-bit value at the end of the buffer.

Adds 40-bit value in little endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 40-bit value to be added.

static inline void **net_buf_add_be40**(struct *net_buf* *buf, uint64_t val)

Add 40-bit value at the end of the buffer.

Adds 40-bit value in big endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 40-bit value to be added.

static inline void **net_buf_add_le48**(struct *net_buf* *buf, uint64_t val)

Add 48-bit value at the end of the buffer.

Adds 48-bit value in little endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 48-bit value to be added.

static inline void **net_buf_add_be48**(struct *net_buf* *buf, uint64_t val)

Add 48-bit value at the end of the buffer.

Adds 48-bit value in big endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 48-bit value to be added.

static inline void **net_buf_add_le64**(struct *net_buf* *buf, uint64_t val)

Add 64-bit value at the end of the buffer.

Adds 64-bit value in little endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 64-bit value to be added.

static inline void **net_buf_add_be64**(struct *net_buf* *buf, uint64_t val)

Add 64-bit value at the end of the buffer.

Adds 64-bit value in big endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 64-bit value to be added.

static inline void *net_buf_remove_mem(struct *net_buf* *buf, size_t len)

Remove data from the end of the buffer.

Removes data from the end of the buffer by modifying the buffer length.

Parameters

- buf – Buffer to update.
- len – Number of bytes to remove.

Returns
New end of the buffer data.

static inline uint8_t net_buf_remove_u8(struct *net_buf* *buf)

Remove a 8-bit value from the end of the buffer.

Same idea as with *net_buf_remove_mem()*, but a helper for operating on 8-bit values.

Parameters

- buf – A valid pointer on a buffer.

Returns
The 8-bit removed value

static inline uint16_t net_buf_remove_le16(struct *net_buf* *buf)

Remove and convert 16 bits from the end of the buffer.

Same idea as with *net_buf_remove_mem()*, but a helper for operating on 16-bit little endian data.

Parameters

- buf – A valid pointer on a buffer.

Returns
16-bit value converted from little endian to host endian.

static inline uint16_t net_buf_remove_be16(struct *net_buf* *buf)

Remove and convert 16 bits from the end of the buffer.

Same idea as with *net_buf_remove_mem()*, but a helper for operating on 16-bit big endian data.

Parameters

- buf – A valid pointer on a buffer.

Returns
16-bit value converted from big endian to host endian.

static inline uint32_t net_buf_remove_be24(struct *net_buf* *buf)

Remove and convert 24 bits from the end of the buffer.

Same idea as with *net_buf_remove_mem()*, but a helper for operating on 24-bit big endian data.

Parameters

- buf – A valid pointer on a buffer.

Returns
24-bit value converted from big endian to host endian.

static inline uint32_t net_buf_remove_le24(struct *net_buf* *buf)

Remove and convert 24 bits from the end of the buffer.

Same idea as with *net_buf_remove_mem()*, but a helper for operating on 24-bit little endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns
24-bit value converted from little endian to host endian.

static inline uint32_t `net_buf_remove_le32`(struct *net_buf* *buf)

Remove and convert 32 bits from the end of the buffer.

Same idea as with *net_buf_remove_mem()*, but a helper for operating on 32-bit little endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns
32-bit value converted from little endian to host endian.

static inline uint32_t `net_buf_remove_be32`(struct *net_buf* *buf)

Remove and convert 32 bits from the end of the buffer.

Same idea as with *net_buf_remove_mem()*, but a helper for operating on 32-bit big endian data.

Parameters

- `buf` – A valid pointer on a buffer

Returns
32-bit value converted from big endian to host endian.

static inline uint64_t `net_buf_remove_le40`(struct *net_buf* *buf)

Remove and convert 40 bits from the end of the buffer.

Same idea as with *net_buf_remove_mem()*, but a helper for operating on 40-bit little endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns
40-bit value converted from little endian to host endian.

static inline uint64_t `net_buf_remove_be40`(struct *net_buf* *buf)

Remove and convert 40 bits from the end of the buffer.

Same idea as with *net_buf_remove_mem()*, but a helper for operating on 40-bit big endian data.

Parameters

- `buf` – A valid pointer on a buffer

Returns
40-bit value converted from big endian to host endian.

static inline uint64_t `net_buf_remove_le48`(struct *net_buf* *buf)

Remove and convert 48 bits from the end of the buffer.

Same idea as with *net_buf_remove_mem()*, but a helper for operating on 48-bit little endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns
48-bit value converted from little endian to host endian.

static inline uint64_t **net_buf_remove_be48**(struct *net_buf* *buf)

Remove and convert 48 bits from the end of the buffer.

Same idea as with *net_buf_remove_mem()*, but a helper for operating on 48-bit big endian data.

Parameters

- **buf** – A valid pointer on a buffer

Returns
48-bit value converted from big endian to host endian.

static inline uint64_t **net_buf_remove_le64**(struct *net_buf* *buf)

Remove and convert 64 bits from the end of the buffer.

Same idea as with *net_buf_remove_mem()*, but a helper for operating on 64-bit little endian data.

Parameters

- **buf** – A valid pointer on a buffer.

Returns
64-bit value converted from little endian to host endian.

static inline uint64_t **net_buf_remove_be64**(struct *net_buf* *buf)

Remove and convert 64 bits from the end of the buffer.

Same idea as with *net_buf_remove_mem()*, but a helper for operating on 64-bit big endian data.

Parameters

- **buf** – A valid pointer on a buffer

Returns
64-bit value converted from big endian to host endian.

static inline void *__net_buf_push__(struct *net_buf* *buf, size_t len)

Prepare data to be added at the start of the buffer.

Modifies the data pointer and buffer length to account for more data in the beginning of the buffer.

Parameters

- **buf** – Buffer to update.
- **len** – Number of bytes to add to the beginning.

Returns
The new beginning of the buffer data.

static inline void *__net_buf_push_mem__(struct *net_buf* *buf, const void *mem, size_t len)

Copies the given number of bytes to the start of the buffer.

Modifies the data pointer and buffer length to account for more data in the beginning of the buffer.

Parameters

- **buf** – Buffer to update.
- **mem** – Location of data to be added.
- **len** – Length of data to be added.

Returns
The new beginning of the buffer data.

static inline void **net_buf_push_u8**(struct *net_buf* *buf, uint8_t val)
Push 8-bit value to the beginning of the buffer.

Adds 8-bit value the beginning of the buffer.

Parameters
- `buf` – Buffer to update.
- `val` – 8-bit value to be pushed to the buffer.

static inline void **net_buf_push_le16**(struct *net_buf* *buf, uint16_t val)
Push 16-bit value to the beginning of the buffer.

Adds 16-bit value in little endian format to the beginning of the buffer.

Parameters
- `buf` – Buffer to update.
- `val` – 16-bit value to be pushed to the buffer.

static inline void **net_buf_push_be16**(struct *net_buf* *buf, uint16_t val)
Push 16-bit value to the beginning of the buffer.

Adds 16-bit value in big endian format to the beginning of the buffer.

Parameters
- `buf` – Buffer to update.
- `val` – 16-bit value to be pushed to the buffer.

static inline void **net_buf_push_le24**(struct *net_buf* *buf, uint32_t val)
Push 24-bit value to the beginning of the buffer.

Adds 24-bit value in little endian format to the beginning of the buffer.

Parameters
- `buf` – Buffer to update.
- `val` – 24-bit value to be pushed to the buffer.

static inline void **net_buf_push_be24**(struct *net_buf* *buf, uint32_t val)
Push 24-bit value to the beginning of the buffer.

Adds 24-bit value in big endian format to the beginning of the buffer.

Parameters
- `buf` – Buffer to update.
- `val` – 24-bit value to be pushed to the buffer.

static inline void **net_buf_push_le32**(struct *net_buf* *buf, uint32_t val)
Push 32-bit value to the beginning of the buffer.

Adds 32-bit value in little endian format to the beginning of the buffer.

Parameters
- `buf` – Buffer to update.
- `val` – 32-bit value to be pushed to the buffer.

static inline void **net_buf_push_be32**(struct *net_buf* *buf, uint32_t val)

Push 32-bit value to the beginning of the buffer.

Adds 32-bit value in big endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 32-bit value to be pushed to the buffer.

static inline void **net_buf_push_le40**(struct *net_buf* *buf, uint64_t val)

Push 40-bit value to the beginning of the buffer.

Adds 40-bit value in little endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 40-bit value to be pushed to the buffer.

static inline void **net_buf_push_be40**(struct *net_buf* *buf, uint64_t val)

Push 40-bit value to the beginning of the buffer.

Adds 40-bit value in big endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 40-bit value to be pushed to the buffer.

static inline void **net_buf_push_le48**(struct *net_buf* *buf, uint64_t val)

Push 48-bit value to the beginning of the buffer.

Adds 48-bit value in little endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 48-bit value to be pushed to the buffer.

static inline void **net_buf_push_be48**(struct *net_buf* *buf, uint64_t val)

Push 48-bit value to the beginning of the buffer.

Adds 48-bit value in big endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 48-bit value to be pushed to the buffer.

static inline void **net_buf_push_le64**(struct *net_buf* *buf, uint64_t val)

Push 64-bit value to the beginning of the buffer.

Adds 64-bit value in little endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 64-bit value to be pushed to the buffer.

static inline void **net_buf_push_be64**(struct *net_buf* *buf, uint64_t val)

Push 64-bit value to the beginning of the buffer.

Adds 64-bit value in big endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 64-bit value to be pushed to the buffer.

static inline void *`net_buf_pull`(struct *net_buf* *buf, size_t len)

Remove data from the beginning of the buffer.

Removes data from the beginning of the buffer by modifying the data pointer and buffer length.

Parameters

- `buf` – Buffer to update.
- `len` – Number of bytes to remove.

Returns

New beginning of the buffer data.

static inline void *`net_buf_pull_mem`(struct *net_buf* *buf, size_t len)

Remove data from the beginning of the buffer.

Removes data from the beginning of the buffer by modifying the data pointer and buffer length.

Parameters

- `buf` – Buffer to update.
- `len` – Number of bytes to remove.

Returns

Pointer to the old beginning of the buffer data.

static inline uint8_t `net_buf_pull_u8`(struct *net_buf* *buf)

Remove a 8-bit value from the beginning of the buffer.

Same idea as with *net_buf_pull()*, but a helper for operating on 8-bit values.

Parameters

- `buf` – A valid pointer on a buffer.

Returns

The 8-bit removed value

static inline uint16_t `net_buf_pull_le16`(struct *net_buf* *buf)

Remove and convert 16 bits from the beginning of the buffer.

Same idea as with *net_buf_pull()*, but a helper for operating on 16-bit little endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns

16-bit value converted from little endian to host endian.

static inline uint16_t `net_buf_pull_be16`(struct *net_buf* *buf)

Remove and convert 16 bits from the beginning of the buffer.

Same idea as with *net_buf_pull()*, but a helper for operating on 16-bit big endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns

16-bit value converted from big endian to host endian.

static inline uint32_t **net_buf_pull_le24**(struct *net_buf* *buf)

Remove and convert 24 bits from the beginning of the buffer.

Same idea as with *net_buf_pull()*, but a helper for operating on 24-bit little endian data.

Parameters

- **buf** – A valid pointer on a buffer.

Returns

24-bit value converted from little endian to host endian.

static inline uint32_t **net_buf_pull_be24**(struct *net_buf* *buf)

Remove and convert 24 bits from the beginning of the buffer.

Same idea as with *net_buf_pull()*, but a helper for operating on 24-bit big endian data.

Parameters

- **buf** – A valid pointer on a buffer.

Returns

24-bit value converted from big endian to host endian.

static inline uint32_t **net_buf_pull_le32**(struct *net_buf* *buf)

Remove and convert 32 bits from the beginning of the buffer.

Same idea as with *net_buf_pull()*, but a helper for operating on 32-bit little endian data.

Parameters

- **buf** – A valid pointer on a buffer.

Returns

32-bit value converted from little endian to host endian.

static inline uint32_t **net_buf_pull_be32**(struct *net_buf* *buf)

Remove and convert 32 bits from the beginning of the buffer.

Same idea as with *net_buf_pull()*, but a helper for operating on 32-bit big endian data.

Parameters

- **buf** – A valid pointer on a buffer

Returns

32-bit value converted from big endian to host endian.

static inline uint64_t **net_buf_pull_le40**(struct *net_buf* *buf)

Remove and convert 40 bits from the beginning of the buffer.

Same idea as with *net_buf_pull()*, but a helper for operating on 40-bit little endian data.

Parameters

- **buf** – A valid pointer on a buffer.

Returns

40-bit value converted from little endian to host endian.

static inline uint64_t **net_buf_pull_be40**(struct *net_buf* *buf)

Remove and convert 40 bits from the beginning of the buffer.

Same idea as with *net_buf_pull()*, but a helper for operating on 40-bit big endian data.

Parameters

- **buf** – A valid pointer on a buffer

Returns

40-bit value converted from big endian to host endian.

static inline uint64_t **net_buf_pull_le48**(struct *net_buf* *buf)

Remove and convert 48 bits from the beginning of the buffer.

Same idea as with *net_buf_pull()*, but a helper for operating on 48-bit little endian data.

Parameters

- **buf** – A valid pointer on a buffer.

Returns

48-bit value converted from little endian to host endian.

static inline uint64_t **net_buf_pull_be48**(struct *net_buf* *buf)

Remove and convert 48 bits from the beginning of the buffer.

Same idea as with *net_buf_pull()*, but a helper for operating on 48-bit big endian data.

Parameters

- **buf** – A valid pointer on a buffer

Returns

48-bit value converted from big endian to host endian.

static inline uint64_t **net_buf_pull_le64**(struct *net_buf* *buf)

Remove and convert 64 bits from the beginning of the buffer.

Same idea as with *net_buf_pull()*, but a helper for operating on 64-bit little endian data.

Parameters

- **buf** – A valid pointer on a buffer.

Returns

64-bit value converted from little endian to host endian.

static inline uint64_t **net_buf_pull_be64**(struct *net_buf* *buf)

Remove and convert 64 bits from the beginning of the buffer.

Same idea as with *net_buf_pull()*, but a helper for operating on 64-bit big endian data.

Parameters

- **buf** – A valid pointer on a buffer

Returns

64-bit value converted from big endian to host endian.

static inline size_t **net_buf_tailroom**(const struct *net_buf* *buf)

Check buffer tailroom.

Check how much free space there is at the end of the buffer.

Parameters

- **buf** – A valid pointer on a buffer

Returns

Number of bytes available at the end of the buffer.

static inline size_t **net_buf_headroom**(const struct *net_buf* *buf)

Check buffer headroom.

Check how much free space there is in the beginning of the buffer.

buf A valid pointer on a buffer

Returns

Number of bytes available in the beginning of the buffer.

static inline uint16_t **net_buf_max_len**(const struct *net_buf* *buf)

Check maximum *net_buf::len* value.

This value is depending on the number of bytes being reserved as headroom.

Parameters

- **buf** – A valid pointer on a buffer

Returns
Number of bytes usable behind the *net_buf::data* pointer.

static inline uint8_t ***net_buf_tail**(const struct *net_buf* *buf)

Get the tail pointer for a buffer.

Get a pointer to the end of the data in a buffer.

Parameters

- **buf** – Buffer.

Returns
Tail pointer for the buffer.

struct *net_buf* ***net_buf_frag_last**(struct *net_buf* *frags)

Find the last fragment in the fragment list.

Returns
Pointer to last fragment in the list.

void **net_buf_frag_insert**(struct *net_buf* *parent, struct *net_buf* *frag)

Insert a new fragment to a chain of bufs.

Insert a new fragment into the buffer fragments list after the parent.

Note: This function takes ownership of the fragment reference so the caller is not required to unref.

Parameters

- **parent** – Parent buffer/fragment.
- **frag** – Fragment to insert.

struct *net_buf* ***net_buf_frag_add**(struct *net_buf* *head, struct *net_buf* *frag)

Add a new fragment to the end of a chain of bufs.

Append a new fragment into the buffer fragments list.

Note: This function takes ownership of the fragment reference so the caller is not required to unref.

Parameters

- **head** – Head of the fragment chain.
- **frag** – Fragment to add.

Returns
New head of the fragment chain. Either head (if head was non-NULL) or frag (if head was NULL).

struct *net_buf* ***net_buf_frag_del**(struct *net_buf* *parent, struct *net_buf* *frag)

Delete existing fragment from a chain of bufs.

Parameters

- **parent** – Parent buffer/fragment, or NULL if there is no parent.
- **frag** – Fragment to delete.

Returns
Pointer to the buffer following the fragment, or NULL if it had no further fragments.

size_t **net_buf_linearize**(void *dst, size_t dst_len, const struct *net_buf* *src, size_t offset, size_t len)

Copy bytes from *net_buf* chain starting at offset to linear buffer.

Copy (extract) *len* bytes from *src* *net_buf* chain, starting from *offset* in it, to a linear buffer *dst*. Return number of bytes actually copied, which may be less than requested, if *net_buf* chain doesn't have enough data, or destination buffer is too small.

Parameters

- **dst** – Destination buffer
- **dst_len** – Destination buffer length
- **src** – Source *net_buf* chain
- **offset** – Starting offset to copy from
- **len** – Number of bytes to copy

Returns
number of bytes actually copied

size_t **net_buf_append_bytes**(struct *net_buf* *buf, size_t len, const void *value, *k_timeout_t* timeout, *net_buf_allocator_cb* allocate_cb, void *user_data)

Append data to a list of *net_buf*.

Append data to a *net_buf*. If there is not enough space in the *net_buf* then more *net_buf* will be added, unless there are no free *net_buf* and timeout occurs. If not allocator is provided it attempts to allocate from the same pool as the original buffer.

Parameters

- **buf** – Network buffer.
- **len** – Total length of input data
- **value** – Data to be added
- **timeout** – Timeout is passed to the *net_buf* allocator callback.
- **allocate_cb** – When a new *net_buf* is required, use this callback.
- **user_data** – A user data pointer to be supplied to the allocate_cb. This pointer is can be anything from a mem_pool or a *net_pkt*, the logic is left up to the allocate_cb function.

Returns
Length of data actually added. This may be less than input length if other timeout than K_FOREVER was used, and there were no free fragments in a pool to accommodate all data.

size_t **net_buf_data_match**(const struct *net_buf* *buf, size_t offset, const void *data, size_t len)

Match data with a *net_buf*'s content.

Compare data with a content of a *net_buf*. Provide information about the number of bytes matching between both. If needed, traverse through multiple buffer fragments.

Parameters

- **buf** – Network buffer
- **offset** – Starting offset to compare from
- **data** – Data buffer for comparison

- `len` – Number of bytes to compare

Returns
The number of bytes compared before the first difference.

static inline struct *net_buf* *`net_buf_skip`(struct *net_buf* *buf, size_t len)

Skip N number of bytes in a *net_buf*.

Skip N number of bytes starting from fragment's offset. If the total length of data is placed in multiple fragments, this function will skip from all fragments until it reaches N number of bytes. Any fully skipped buffers are removed from the *net_buf* list.

Parameters

- `buf` – Network buffer.
- `len` – Total length of data to be skipped.

Returns
Pointer to the fragment or NULL and pos is 0 after successful skip, NULL and pos is 0xffff otherwise.

static inline size_t `net_buf_frags_len`(const struct *net_buf* *buf)

Calculate amount of bytes stored in fragments.

Calculates the total amount of data stored in the given buffer and the fragments linked to it.

Parameters

- `buf` – Buffer to start off with.

Returns
Number of bytes in the buffer and its fragments.

struct `net_buf_simple`

#include <buf.h> Simple network buffer representation.

This is a simpler variant of the *net_buf* object (in fact *net_buf* uses *net_buf_simple* internally). It doesn't provide any kind of reference counting, user data, dynamic allocation, or in general the ability to pass through kernel objects such as FIFOs.

The main use of this is for scenarios where the meta-data of the normal *net_buf* isn't needed and causes too much overhead. This could be e.g. when the buffer only needs to be allocated on the stack or when the access to and lifetime of the buffer is well controlled and constrained.

Public Members

uint8_t *`data`

Pointer to the start of data in the buffer.

uint16_t `len`

Length of the data behind the data pointer.

To determine the max length, use *net_buf_simple_max_len()*, not *size*!

uint16_t `size`

Amount of data that net_buf_simple::__buf can store.

struct **net_buf_simple_state**

#include <*buf.h*> Parsing state of a buffer.

This is used for temporarily storing the parsing state of a buffer while giving control of the parsing to a routine which we don't control.

Public Members

uint16_t **offset**

Offset of the data pointer from the beginning of the storage.

uint16_t **len**

Length of data.

struct **net_buf**

#include <*buf.h*> Network buffer representation.

This struct is used to represent network buffers. Such buffers are normally defined through the NET_BUF_POOL_*_DEFINE() APIs and allocated using the *net_buf_alloc()* API.

Public Members

sys_snode_t **node**

Allow placing the buffer into sys_slist_t.

struct *net_buf* ***frags**

Fragments associated with this buffer.

uint8_t **ref**

Reference count.

uint8_t **flags**

Bit-field of buffer flags.

uint8_t **pool_id**

Where the buffer should go when freed up.

uint8_t **user_data_size**

Size of user data on this buffer.

uint8_t ***data**

Pointer to the start of data in the buffer.

uint16_t **len**

Length of the data behind the data pointer.

uint16_t **size**

Amount of data that this buffer can store.

union net_buf

Union for convenience access to the *net_buf_simple* members, also preserving the old API.

uint8_t user_data[]

System metadata for this buffer.

struct net_buf_pool

#include <buf.h> Network buffer pool representation.

This struct is used to represent a pool of network buffers.

Public Members

struct k_lifo free

LIFO to place the buffer into when free.

struct *k_spinlock* lock

To prevent concurrent access/modifications.

const uint16_t buf_count

Number of buffers in pool.

uint16_t uninit_count

Number of uninitialized buffers.

uint8_t user_data_size

Size of user data allocated to this pool.

void (*const destroy)(struct *net_buf* *buf)

Optional destroy callback when buffer is freed.

const struct net_buf_data_alloc *alloc

Data allocation handlers.

Packet Management

- *Overview*
  - *Architectural notes*
- *Memory management*
  - *Allocation*
  - *Buffer allocation*
  - *Deallocation*
- *Operations*
  - *Read and Write access*

  - *Data access*
- *API Reference*

Overview Network packets are the main data the networking stack manipulates. Such data is represented through the net_pkt structure which provides a means to hold the packet, write and read it, as well as necessary metadata for the core to hold important information. Such an object is called net_pkt in this document.

The data structure and the whole API around it are defined in include/zephyr/net/net_pkt.h.

Architectural notes There are two network packets flows within the stack, **TX** for the transmission path, and **RX** for the reception one. In both paths, each net_pkt is written and read from the beginning to the end, or more specifically from the headers to the payload.

Memory management

Allocation All net_pkt objects come from a pre-defined pool of struct net_pkt. Such pool is defined via

```
NET_PKT_SLAB_DEFINE(name, count)
```

Note, however, one will rarely have to use it, as the core provides already two pools, one for the TX path and one for the RX path.

Allocating a raw net_pkt can be done through:

```
pkt = net_pkt_alloc(timeout);
```

However, by its nature, a raw net_pkt is useless without a buffer and needs various metadata information to become relevant as well. It requires at least to get the network interface it is meant to be sent through or through which it was received. As this is a very common operation, a helper exist:

```
pkt = net_pkt_alloc_on_iface(iface, timeout);
```

A more complete allocator exists, where both the net_pkt and its buffer can be allocated at once:

```
pkt = net_pkt_alloc_with_buffer(iface, size, family, proto, timeout);
```

See below how the buffer is allocated.

Buffer allocation The net_pkt object does not define its own buffer, but instead uses an existing object for this: *net_buf*. (See *Network Buffer* for more information). However, it mostly hides the usage of such a buffer because net_pkt brings network awareness to buffer allocation and, as we will see later, its operation too.

To allocate a buffer, a net_pkt needs to have at least its network interface set. This works if the family of the packet is unknown at the time of buffer allocation. Then one could do:

```
net_pkt_alloc_buffer(pkt, size, proto, timeout);
```

Where proto could be 0 if unknown (there is no IPPROTO_UNSPEC).

As seen previously, the net_pkt and its buffer can be allocated at once via *net_pkt_alloc_with_buffer()*. It is actually the most widely used allocator.

The network interface, the family, and the protocol of the packet are used by the buffer allocation to determine if the requested size can be allocated. Indeed, the allocator will use the network interface to know the MTU and then the family and protocol for the headers space (if only these 2 are specified). If the whole fits within the MTU, the allocated space will be of the requested size plus, eventually, the headers space. If there is insufficient MTU space, the requested size will be shrunk so the possible headers space and new size will fit within the MTU.

For instance, on an Ethernet network interface, with an MTU of 1500 bytes:

```
pkt = net_pkt_alloc_with_buffer(iface, 800, AF_INET4, IPPROTO_UDP, K_FOREVER);
```

will successfully allocate 800 + 20 + 8 bytes of buffer for the new net_pkt where:

```
pkt = net_pkt_alloc_with_buffer(iface, 1600, AF_INET4, IPPROTO_UDP, K_FOREVER);
```

will successfully allocate 1500 bytes, and where 20 + 8 bytes (IPv4 + UDP headers) will not be used for the payload.

On the receiving side, when the family and protocol are not known:

```
pkt = net_pkt_rx_alloc_with_buffer(iface, 800, AF_UNSPEC, 0, K_FOREVER);
```

will allocate 800 bytes and no extra header space. But a:

```
pkt = net_pkt_rx_alloc_with_buffer(iface, 1600, AF_UNSPEC, 0, K_FOREVER);
```

will allocate 1514 bytes, the MTU + Ethernet header space.

One can increase the amount of buffer space allocated by calling *net_pkt_alloc_buffer()*, as it will take into account the existing buffer. It will also account for the header space if net_pkt's family is a valid one, as well as the proto parameter. In that case, the newly allocated buffer space will be appended to the existing one, and not inserted in the front. Note however such a use case is rather limited. Usually, one should know from the start how much size should be requested.

Deallocation Each net_pkt is reference counted. At allocation, the reference is set to 1. The reference count can be incremented with *net_pkt_ref()* or decremented with *net_pkt_unref()*. When the count drops to zero the buffer is also un-referenced and net_pkt is automatically placed back into the free net_pkt_slabs

If net_pkt's buffer is needed even after net_pkt deallocation, one will need to reference once more all the chain of net_buf before calling last net_pkt_unref. See *Network Buffer* for more information.

Operations There are two ways to access the net_pkt buffer, explained in the following sections: basic read/write access and data access, the latter being the preferred way.

Read and Write access As said earlier, though net_pkt uses net_buf for its buffer, it provides its own API to access it. Indeed, a network packet might be scattered over a chain of net_buf objects, the functions provided by net_buf are then limited for such case. Instead, net_pkt provides functions which hide all the complexity of potential non-contiguous access.

Data movement into the buffer is made through a cursor maintained within each net_pkt. All read/write operations affect this cursor. Note as well that read or write functions are strict on their length parameters: if it cannot r/w the given length it will fail. Length is not interpreted as an upper limit, it is instead the exact amount of data that must be read or written.

As there are two paths, TX and RX, there are two access modes: write and overwrite. This might sound a bit unusual, but is in fact simple and provides flexibility.

In write mode, whatever is written in the buffer affects the length of actual data present in the buffer. Buffer length should not be confused with the buffer size which is a limit any mode cannot pass. In overwrite mode then, whatever is written must happen on valid data, and will not affect the buffer length. By default, a newly allocated net_pkt is on write mode, and its cursor points to the beginning of its buffer.

Let's see now, step by step, the functions and how they behave depending on the mode.

When freshly allocated with a buffer of 500 bytes, a net_pkt has 0 length, which means no valid data is in its buffer. One could verify this by:

```
len = net_pkt_get_len(pkt);
```

Now, let's write 8 bytes:

```
net_pkt_write(pkt, data, 8);
```

The buffer length is now 8 bytes. There are various helpers to write a byte, or big endian uint16_t, uint32_t.

```
net_pkt_write_u8(pkt, &foo);
net_pkt_write_be16(pkt, &ba);
net_pkt_write_be32(pkt, &bar);
```

Logically, net_pkt's length is now 15. But if we try to read at this point, it will fail because there is nothing to read at the cursor where we are at in the net_pkt. It is possible, while in write mode, to read what has been already written by resetting the cursor of the net_pkt. For instance:

```
net_pkt_cursor_init(pkt);
net_pkt_read(pkt, data, 15);
```

This will reset the cursor of the pkt to the beginning of the buffer and then let you read the actual 15 bytes present. The cursor is then again pointing at the end of the buffer.

To set a large area with the same byte, a memset function is provided:

```
net_pkt_memset(pkt, 0, 5);
```

Our net_pkt has now a length of 20 bytes.

Switching between modes can be achieved via `net_pkt_set_overwrite()` function. It is possible to switch mode back and forth at any time. The net_pkt will be set to overwrite and its cursor reset:

```
net_pkt_set_overwrite(pkt, true);
net_pkt_cursor_init(pkt);
```

Now the same operators can be used, but it will be limited to the existing data in the buffer, i.e. 20 bytes.

If it is necessary to know how much space is available in the net_pkt call:

```
net_pkt_available_buffer(pkt);
```

Or, if headers space needs to be accounted for, call:

```
net_pkt_available_payload_buffer(pkt, proto);
```

If you want to place the cursor at a known position use the function *net_pkt_skip()*. For example, to go after the IP header, use:

```
net_pkt_cursor_init(pkt);
net_pkt_skip(pkt, net_pkt_ip_header_len(pkt));
```

Data access Though the API shown previously is rather simple, it involves always copying things to and from the net_pkt buffer. In many occasions, it is more relevant to access the information stored in the buffer contiguously, especially with network packets which embed headers.

These headers are, most of the time, a known fixed set of bytes. It is then more natural to have a structure representing a certain type of header. In addition to this, if it is known the header size appears in a contiguous area of the buffer, it will be way more efficient to cast the actual position in the buffer to the type of header. Either for reading or writing the fields of such header, accessing it directly will save memory.

Net pkt comes with a dedicated API for this, built on top of the previously described API. It is able to handle both contiguous and non-contiguous access transparently.

There are two macros used to define a data access descriptor: `NET_PKT_DATA_ACCESS_DEFINE` when it is not possible to tell if the data will be in a contiguous area, and `NET_PKT_DATA_ACCESS_CONTIGUOUS_DEFINE` when it is guaranteed the data is in a contiguous area.

Let's take the example of IP and UDP. Both IPv4 and IPv6 headers are always found at the beginning of the packet and are small enough to fit in a net_buf of 128 bytes (for instance, though 64 bytes could be chosen).

```
NET_PKT_DATA_ACCESS_CONTIGUOUS_DEFINE(ipv4_access, struct net_ipv4_hdr);
struct net_ipv4_hdr *ipv4_hdr;

ipv4_hdr = (struct net_ipv4_hdr *)net_pkt_get_data(pkt, &ipv4_access);
```

It would be the same for struct net_ipv4_hdr. For a UDP header it is likely not to be in a contiguous area in IPv6 for instance so:

```
NET_PKT_DATA_ACCESS_DEFINE(udp_access, struct net_udp_hdr);
struct net_udp_hdr *udp_hdr;

udp_hdr = (struct net_udp_hdr *)net_pkt_get_data(pkt, &udp_access);
```

At this point, the cursor of the net_pkt points at the beginning of the requested data. On the RX path, these headers will be read but not modified so to proceed further the cursor needs to advance past the data. There is a function dedicated for this:

```
net_pkt_acknowledge_data(pkt, &ipv4_access);
```

On the TX path, however, the header fields have been modified. In such a case:

```
net_pkt_set_data(pkt, &ipv4_access);
```

If the data are in a contiguous area, it will advance the cursor relevantly. If not, it will write the data and the cursor will be updated. Note that *net_pkt_set_data()* could be used in the RX path as well, but it is slightly faster to use *net_pkt_acknowledge_data()* as this one does not care about contiguity at all, it just advances the cursor via *net_pkt_skip()* directly.

Related code samples

Dumb HTTP server
Implement a simple, portable, HTTP server using BSD sockets.

Dumb HTTP server (multi-threaded)
Implement a simple HTTP server supporting simultaneous connections using BSD sockets.

API Reference

group net_pkt

Network packet management library.

Defines

NET_PKT_SLAB_DEFINE(name, count)

Create a *net_pkt* slab.

A *net_pkt* slab is used to store meta-information about network packets. It must be coupled with a data fragment pool (*NET_PKT_DATA_POOL_DEFINE*) used to store the actual packet data. The macro can be used by an application to define additional custom per-context TX packet slabs (see net_context_setup_pools()).

Parameters

- **name** – Name of the slab.
- **count** – Number of *net_pkt* in this slab.

NET_PKT_DATA_POOL_DEFINE(name, count)

Create a data fragment *net_buf* pool.

A *net_buf* pool is used to store actual data for network packets. It must be coupled with a *net_pkt* slab (*NET_PKT_SLAB_DEFINE*) used to store the packet meta-information. The macro can be used by an application to define additional custom per-context TX packet pools (see net_context_setup_pools()).

Parameters

- **name** – Name of the pool.
- **count** – Number of *net_buf* in this pool.

net_pkt_print_frags(pkt)

Print fragment list and the fragment sizes.

Only available if debugging is activated.

Parameters

- **pkt** – Network pkt.

Functions

struct *net_buf* *net_pkt_get_reserve_data(struct *net_buf_pool* *pool, size_t min_len, *k_timeout_t* timeout)

Get a data buffer from a given pool.

Normally this version is not useful for applications but is mainly used by network fragmentation code.

Parameters

- **pool** – The *net_buf* pool to use.
- **min_len** – Minimum length of the requested fragment.
- **timeout** – Affects the action taken should the net buf pool be empty. If K_NO_WAIT, then return immediately. If K_FOREVER, then wait as long as necessary. Otherwise, wait up to the specified time.

Returns
Network buffer if successful, NULL otherwise.

struct *net_buf* ***net_pkt_get_reserve_rx_data**(size_t min_len, *k_timeout_t* timeout)

Get RX DATA buffer from pool.

Normally you should use *net_pkt_get_frag()* instead.

Normally this version is not useful for applications but is mainly used by network fragmentation code.

Parameters

- **min_len** – Minimum length of the requested fragment.
- **timeout** – Affects the action taken should the net buf pool be empty. If K_NO_WAIT, then return immediately. If K_FOREVER, then wait as long as necessary. Otherwise, wait up to the specified time.

Returns
Network buffer if successful, NULL otherwise.

struct *net_buf* ***net_pkt_get_reserve_tx_data**(size_t min_len, *k_timeout_t* timeout)

Get TX DATA buffer from pool.

Normally you should use *net_pkt_get_frag()* instead.

Normally this version is not useful for applications but is mainly used by network fragmentation code.

Parameters

- **min_len** – Minimum length of the requested fragment.
- **timeout** – Affects the action taken should the net buf pool be empty. If K_NO_WAIT, then return immediately. If K_FOREVER, then wait as long as necessary. Otherwise, wait up to the specified time.

Returns
Network buffer if successful, NULL otherwise.

struct *net_buf* ***net_pkt_get_frag**(struct *net_pkt* *pkt, size_t min_len, *k_timeout_t* timeout)

Get a data fragment that might be from user specific buffer pool or from global DATA pool.

Parameters

- **pkt** – Network packet.
- **min_len** – Minimum length of the requested fragment.
- **timeout** – Affects the action taken should the net buf pool be empty. If K_NO_WAIT, then return immediately. If K_FOREVER, then wait as long as necessary. Otherwise, wait up to the specified time.

Returns
Network buffer if successful, NULL otherwise.

void **net_pkt_unref**(struct *net_pkt* *pkt)

Place packet back into the available packets slab.

Releases the packet to other use. This needs to be called by application after it has finished with the packet.

Parameters

- **pkt** – Network packet to release.

struct *net_pkt* *net_pkt_ref(struct *net_pkt* *pkt)

Increase the packet ref count.

Mark the packet to be used still.

Parameters

- pkt – Network packet to ref.

Returns

Network packet if successful, NULL otherwise.

struct *net_buf* *net_pkt_frag_ref(struct *net_buf* *frag)

Increase the packet fragment ref count.

Mark the fragment to be used still.

Parameters

- frag – Network fragment to ref.

Returns

a pointer on the referenced Network fragment.

void net_pkt_frag_unref(struct *net_buf* *frag)

Decrease the packet fragment ref count.

Parameters

- frag – Network fragment to unref.

struct *net_buf* *net_pkt_frag_del(struct *net_pkt* *pkt, struct *net_buf* *parent, struct *net_buf* *frag)

Delete existing fragment from a packet.

Parameters

- pkt – Network packet from which frag belongs to.
- parent – parent fragment of frag, or NULL if none.
- frag – Fragment to delete.

Returns

Pointer to the following fragment, or NULL if it had no further fragments.

void net_pkt_frag_add(struct *net_pkt* *pkt, struct *net_buf* *frag)

Add a fragment to a packet at the end of its fragment list.

Parameters

- pkt – pkt Network packet where to add the fragment
- frag – Fragment to add

void net_pkt_frag_insert(struct *net_pkt* *pkt, struct *net_buf* *frag)

Insert a fragment to a packet at the beginning of its fragment list.

Parameters

- pkt – pkt Network packet where to insert the fragment
- frag – Fragment to insert

void net_pkt_compact(struct *net_pkt* *pkt)

Compact the fragment list of a packet.

After this there is no more any free space in individual fragments.

Parameters

- **pkt** – Network packet.

void **net_pkt_get_info**(struct k_mem_slab **rx, struct k_mem_slab **tx, struct *net_buf_pool* **rx_data, struct *net_buf_pool* **tx_data)

Get information about predefined RX, TX and DATA pools.

Parameters

- **rx** – Pointer to RX pool is returned.
- **tx** – Pointer to TX pool is returned.
- **rx_data** – Pointer to RX DATA pool is returned.
- **tx_data** – Pointer to TX DATA pool is returned.

struct *net_pkt* ***net_pkt_alloc**(*k_timeout_t* timeout)

Allocate an initialized *net_pkt*.

for the time being, 2 pools are used. One for TX and one for RX. This allocator has to be used for TX.

Parameters

- **timeout** – Maximum time to wait for an allocation.

Returns
a pointer to a newly allocated *net_pkt* on success, NULL otherwise.

struct *net_pkt* ***net_pkt_alloc_from_slab**(struct k_mem_slab *slab, *k_timeout_t* timeout)

Allocate an initialized *net_pkt* from a specific slab.

unlike *net_pkt_alloc()* which uses core slabs, this one will use an external slab (see *NET_PKT_SLAB_DEFINE()*). Do *not* use it unless you know what you are doing. Basically, only net_context should be using this, in order to allocate packet and then buffer on its local slab/pool (if any).

Parameters

- **slab** – The slab to use for allocating the packet
- **timeout** – Maximum time to wait for an allocation.

Returns
a pointer to a newly allocated *net_pkt* on success, NULL otherwise.

struct *net_pkt* ***net_pkt_rx_alloc**(*k_timeout_t* timeout)

Allocate an initialized *net_pkt* for RX.

for the time being, 2 pools are used. One for TX and one for RX. This allocator has to be used for RX.

Parameters

- **timeout** – Maximum time to wait for an allocation.

Returns
a pointer to a newly allocated *net_pkt* on success, NULL otherwise.

struct *net_pkt* ***net_pkt_alloc_on_iface**(struct *net_if* *iface, *k_timeout_t* timeout)

Allocate a network packet for a specific network interface.

Parameters

- **iface** – The network interface the packet is supposed to go through.
- **timeout** – Maximum time to wait for an allocation.

Returns
a pointer to a newly allocated *net_pkt* on success, NULL otherwise.

int **net_pkt_alloc_buffer**(struct *net_pkt* *pkt, size_t size, enum *net_ip_protocol* proto, *k_timeout_t* timeout)

Allocate buffer for a *net_pkt*.

: such allocator will take into account space necessary for headers, MTU, and existing buffer (if any). Beware that, due to all these criteria, the allocated size might be smaller/bigger than requested one.

Parameters

- **pkt** – The network packet requiring buffer to be allocated.
- **size** – The size of buffer being requested.
- **proto** – The IP protocol type (can be 0 for none).
- **timeout** – Maximum time to wait for an allocation.

Returns
0 on success, negative errno code otherwise.

int **net_pkt_alloc_buffer_raw**(struct *net_pkt* *pkt, size_t size, *k_timeout_t* timeout)

Allocate buffer for a *net_pkt*, of specified size, w/o any additional preconditions.

: The actual buffer size may be larger than requested one if fixed size buffers are in use.

Parameters

- **pkt** – The network packet requiring buffer to be allocated.
- **size** – The size of buffer being requested.
- **timeout** – Maximum time to wait for an allocation.

Returns
0 on success, negative errno code otherwise.

struct *net_pkt* ***net_pkt_alloc_with_buffer**(struct *net_if* *iface, size_t size, *sa_family_t* family, enum *net_ip_protocol* proto, *k_timeout_t* timeout)

Allocate a network packet and buffer at once.

Parameters

- **iface** – The network interface the packet is supposed to go through.
- **size** – The size of buffer.
- **family** – The family to which the packet belongs.
- **proto** – The IP protocol type (can be 0 for none).
- **timeout** – Maximum time to wait for an allocation.

Returns
a pointer to a newly allocated *net_pkt* on success, NULL otherwise.

void **net_pkt_append_buffer**(struct *net_pkt* *pkt, struct *net_buf* *buffer)

Append a buffer in packet.

Parameters

- **pkt** – Network packet where to append the buffer
- **buffer** – Buffer to append

size_t **net_pkt_available_buffer**(struct *net_pkt* *pkt)

Get available buffer space from a pkt.

> **Note**
>
> Reserved bytes (headroom) in any of the fragments are not considered to be available.

Parameters

- **pkt** – The *net_pkt* which buffer availability should be evaluated

Returns

the amount of buffer available

size_t **net_pkt_available_payload_buffer**(struct *net_pkt* *pkt, enum *net_ip_protocol* proto)

Get available buffer space for payload from a pkt.

Unlike *net_pkt_available_buffer()*, this will take into account the headers space.

> **Note**
>
> Reserved bytes (headroom) in any of the fragments are not considered to be available.

Parameters

- **pkt** – The *net_pkt* which payload buffer availability should be evaluated
- **proto** – The IP protocol type (can be 0 for none).

Returns

the amount of buffer available for payload

void **net_pkt_trim_buffer**(struct *net_pkt* *pkt)

Trim *net_pkt* buffer.

This will basically check for unused buffers and deallocate them relevantly

Parameters

- **pkt** – The *net_pkt* which buffer will be trimmed

int **net_pkt_remove_tail**(struct *net_pkt* *pkt, size_t length)

Remove *length* bytes from tail of packet.

This function does not take packet cursor into account. It is a helper to remove unneeded bytes from tail of packet (like appended CRC). It takes care of buffer deallocation if removed bytes span whole buffer(s).

Parameters

- **pkt** – Network packet
- **length** – Number of bytes to be removed

Return values

- **0** – On success.
- **-EINVAL** – If packet length is shorter than *length*.

void **net_pkt_cursor_init**(struct *net_pkt* *pkt)

Initialize *net_pkt* cursor.

This will initialize the *net_pkt* cursor from its buffer.

Parameters

- **pkt** – The *net_pkt* whose cursor is going to be initialized

static inline void **net_pkt_cursor_backup**(struct *net_pkt* *pkt, struct net_pkt_cursor *backup)

Backup *net_pkt* cursor.

Parameters

- **pkt** – The *net_pkt* whose cursor is going to be backed up
- **backup** – The cursor where to backup *net_pkt* cursor

static inline void **net_pkt_cursor_restore**(struct *net_pkt* *pkt, struct net_pkt_cursor *backup)

Restore *net_pkt* cursor from a backup.

Parameters

- **pkt** – The *net_pkt* whose cursor is going to be restored
- **backup** – The cursor from where to restore *net_pkt* cursor

static inline void ***net_pkt_cursor_get_pos**(struct *net_pkt* *pkt)

Returns current position of the cursor.

Parameters

- **pkt** – The *net_pkt* whose cursor position is going to be returned

Returns

cursor's position

int **net_pkt_skip**(struct *net_pkt* *pkt, size_t length)

Skip some data from a *net_pkt*.

net_pkt's cursor should be properly initialized Cursor position will be updated after the operation. Depending on the value of pkt->overwrite bit, this function will affect the buffer length or not. If it's true, it will advance the cursor to the requested length. If it's false, it will do the same but if the cursor was already also at the end of existing data, it will increment the buffer length. So in this case, its behavior is just like net_pkt_write or net_pkt_memset, difference being that it will not affect the buffer content itself (which may be just garbage then).

Parameters

- **pkt** – The *net_pkt* whose cursor will be updated to skip given amount of data from the buffer.
- **length** – Amount of data to skip in the buffer

Returns

0 in success, negative errno code otherwise.

int **net_pkt_memset**(struct *net_pkt* *pkt, int byte, size_t length)

Memset some data in a *net_pkt*.

net_pkt's cursor should be properly initialized and, if needed, positioned using net_pkt_skip. Cursor position will be updated after the operation.

Parameters

- **pkt** – The *net_pkt* whose buffer to fill starting at the current cursor position.
- **byte** – The byte to write in memory
- **length** – Amount of data to memset with given byte

Returns
0 in success, negative errno code otherwise.

int **net_pkt_copy**(struct *net_pkt* *pkt_dst, struct *net_pkt* *pkt_src, size_t length)

Copy data from a packet into another one.

Both *net_pkt* cursors should be properly initialized and, if needed, positioned using net_pkt_skip. The cursors will be updated after the operation.

Parameters

- **pkt_dst** – Destination network packet.
- **pkt_src** – Source network packet.
- **length** – Length of data to be copied.

Returns
0 on success, negative errno code otherwise.

struct *net_pkt* ***net_pkt_clone**(struct *net_pkt* *pkt, *k_timeout_t* timeout)

Clone pkt and its buffer.

The cloned packet will be allocated on the same pool as the original one.

Parameters

- **pkt** – Original pkt to be cloned
- **timeout** – Timeout to wait for free buffer

Returns
NULL if error, cloned packet otherwise.

struct *net_pkt* ***net_pkt_rx_clone**(struct *net_pkt* *pkt, *k_timeout_t* timeout)

Clone pkt and its buffer.

The cloned packet will be allocated on the RX packet poll.

Parameters

- **pkt** – Original pkt to be cloned
- **timeout** – Timeout to wait for free buffer

Returns
NULL if error, cloned packet otherwise.

struct *net_pkt* ***net_pkt_shallow_clone**(struct *net_pkt* *pkt, *k_timeout_t* timeout)

Clone pkt and increase the refcount of its buffer.

Parameters

- **pkt** – Original pkt to be shallow cloned
- **timeout** – Timeout to wait for free packet

Returns
NULL if error, cloned packet otherwise.

int **net_pkt_read**(struct *net_pkt* *pkt, void *data, size_t length)

Read some data from a *net_pkt*.

net_pkt's cursor should be properly initialized and, if needed, positioned using net_pkt_skip. Cursor position will be updated after the operation.

Parameters

- `pkt` – The network packet from where to read some data
- `data` – The destination buffer where to copy the data
- `length` – The amount of data to copy

Returns
0 on success, negative errno code otherwise.

static inline int `net_pkt_read_u8`(struct *net_pkt* *pkt, uint8_t *data)

Read a byte (uint8_t) from a *net_pkt*.

net_pkt's cursor should be properly initialized and, if needed, positioned using net_pkt_skip. Cursor position will be updated after the operation.

Parameters

- `pkt` – The network packet from where to read
- `data` – The destination uint8_t where to copy the data

Returns
0 on success, negative errno code otherwise.

int `net_pkt_read_be16`(struct *net_pkt* *pkt, uint16_t *data)

Read uint16_t big endian data from a *net_pkt*.

net_pkt's cursor should be properly initialized and, if needed, positioned using net_pkt_skip. Cursor position will be updated after the operation.

Parameters

- `pkt` – The network packet from where to read
- `data` – The destination uint16_t where to copy the data

Returns
0 on success, negative errno code otherwise.

int `net_pkt_read_le16`(struct *net_pkt* *pkt, uint16_t *data)

Read uint16_t little endian data from a *net_pkt*.

net_pkt's cursor should be properly initialized and, if needed, positioned using net_pkt_skip. Cursor position will be updated after the operation.

Parameters

- `pkt` – The network packet from where to read
- `data` – The destination uint16_t where to copy the data

Returns
0 on success, negative errno code otherwise.

int `net_pkt_read_be32`(struct *net_pkt* *pkt, uint32_t *data)

Read uint32_t big endian data from a *net_pkt*.

net_pkt's cursor should be properly initialized and, if needed, positioned using net_pkt_skip. Cursor position will be updated after the operation.

Parameters

- `pkt` – The network packet from where to read
- `data` – The destination uint32_t where to copy the data

Returns
0 on success, negative errno code otherwise.

int **net_pkt_write**(struct *net_pkt* *pkt, const void *data, size_t length)

Write data into a *net_pkt*.

net_pkt's cursor should be properly initialized and, if needed, positioned using net_pkt_skip. Cursor position will be updated after the operation.

Parameters

- **pkt** – The network packet where to write
- **data** – Data to be written
- **length** – Length of the data to be written

Returns

0 on success, negative errno code otherwise.

static inline int **net_pkt_write_u8**(struct *net_pkt* *pkt, uint8_t data)

Write a byte (uint8_t) data to a *net_pkt*.

net_pkt's cursor should be properly initialized and, if needed, positioned using net_pkt_skip. Cursor position will be updated after the operation.

Parameters

- **pkt** – The network packet from where to read
- **data** – The uint8_t value to write

Returns

0 on success, negative errno code otherwise.

static inline int **net_pkt_write_be16**(struct *net_pkt* *pkt, uint16_t data)

Write a uint16_t big endian data to a *net_pkt*.

net_pkt's cursor should be properly initialized and, if needed, positioned using net_pkt_skip. Cursor position will be updated after the operation.

Parameters

- **pkt** – The network packet from where to read
- **data** – The uint16_t value in host byte order to write

Returns

0 on success, negative errno code otherwise.

static inline int **net_pkt_write_be32**(struct *net_pkt* *pkt, uint32_t data)

Write a uint32_t big endian data to a *net_pkt*.

net_pkt's cursor should be properly initialized and, if needed, positioned using net_pkt_skip. Cursor position will be updated after the operation.

Parameters

- **pkt** – The network packet from where to read
- **data** – The uint32_t value in host byte order to write

Returns

0 on success, negative errno code otherwise.

static inline int **net_pkt_write_le32**(struct *net_pkt* *pkt, uint32_t data)

Write a uint32_t little endian data to a *net_pkt*.

net_pkt's cursor should be properly initialized and, if needed, positioned using net_pkt_skip. Cursor position will be updated after the operation.

Parameters

- **pkt** – The network packet from where to read

- **data** – The uint32_t value in host byte order to write

Returns
0 on success, negative errno code otherwise.

static inline int **net_pkt_write_le16**(struct *net_pkt* *pkt, uint16_t data)

Write a uint16_t little endian data to a *net_pkt*.

net_pkt's cursor should be properly initialized and, if needed, positioned using net_pkt_skip. Cursor position will be updated after the operation.

Parameters

- **pkt** – The network packet from where to read
- **data** – The uint16_t value in host byte order to write

Returns
0 on success, negative errno code otherwise.

size_t **net_pkt_remaining_data**(struct *net_pkt* *pkt)

Get the amount of data which can be read from current cursor position.

Parameters

- **pkt** – Network packet

Returns
Amount of data which can be read from current pkt cursor

int **net_pkt_update_length**(struct *net_pkt* *pkt, size_t length)

Update the overall length of a packet.

Unlike *net_pkt_pull()* below, this does not take packet cursor into account. It's mainly a helper dedicated for ipv4 and ipv6 input functions. It shrinks the overall length by given parameter.

Parameters

- **pkt** – Network packet
- **length** – The new length of the packet

Returns
0 on success, negative errno code otherwise.

int **net_pkt_pull**(struct *net_pkt* *pkt, size_t length)

Remove data from the packet at current location.

net_pkt's cursor should be properly initialized and, eventually, properly positioned using net_pkt_skip/read/write. Note that *net_pkt*'s cursor is reset by this function.

Parameters

- **pkt** – Network packet
- **length** – Number of bytes to be removed

Returns
0 on success, negative errno code otherwise.

uint16_t **net_pkt_get_current_offset**(struct *net_pkt* *pkt)

Get the actual offset in the packet from its cursor.

Parameters

- **pkt** – Network packet.

Returns
a valid offset on success, 0 otherwise as there is nothing that can be done to evaluate the offset.

bool **net_pkt_is_contiguous**(struct *net_pkt* *pkt, size_t size)

Check if a data size could fit contiguously.

net_pkt's cursor should be properly initialized and, if needed, positioned using net_pkt_skip.

Parameters

- **pkt** – Network packet.
- **size** – The size to check for contiguity

Returns

true if that is the case, false otherwise.

size_t **net_pkt_get_contiguous_len**(struct *net_pkt* *pkt)

Get the contiguous buffer space.

Parameters

- **pkt** – Network packet

Returns

The available contiguous buffer space in bytes starting from the current cursor position. 0 in case of an error.

void ***net_pkt_get_data**(struct *net_pkt* *pkt, struct net_pkt_data_access *access)

Get data from a network packet in a contiguous way.

net_pkt's cursor should be properly initialized and, if needed, positioned using net_pkt_skip. Unlike other functions, cursor position will not be updated after the operation.

Parameters

- **pkt** – The network packet from where to get the data.
- **access** – A pointer to a valid net_pkt_data_access describing the data to get in a contiguous way.

Returns

a pointer to the requested contiguous data, NULL otherwise.

int **net_pkt_set_data**(struct *net_pkt* *pkt, struct net_pkt_data_access *access)

Set contiguous data into a network packet.

net_pkt's cursor should be properly initialized and, if needed, positioned using net_pkt_skip. Cursor position will be updated after the operation.

Parameters

- **pkt** – The network packet to where the data should be set.
- **access** – A pointer to a valid net_pkt_data_access describing the data to set.

Returns

0 on success, a negative errno otherwise.

static inline int **net_pkt_acknowledge_data**(struct *net_pkt* *pkt, struct net_pkt_data_access *access)

Acknowledge previously contiguous data taken from a network packet Packet needs to be set to overwrite mode.

struct **net_pkt**

#include <net_pkt.h> Network packet.

Note that if you add new fields into *net_pkt*, remember to update *net_pkt_clone()* function.

Public Members

intptr_t **fifo**

The fifo is used by RX/TX threads and by socket layer.

The *net_pkt* is queued via fifo to the processing thread.

struct k_mem_slab ***slab**

Slab pointer from where it belongs to.

struct *net_buf* ***frags**

buffer fragment

struct *net_buf* ***buffer**

alias to a buffer fragment

union net_pkt

buffer holding the packet

struct net_pkt_cursor **cursor**

Internal buffer iterator used for reading/writing.

struct net_context ***context**

Network connection context.

struct *net_if* ***iface**

Network interface.

Networking Technologies

Ethernet

- *Overview*
- *API Reference*

Virtual LAN (VLAN) Support

- *Overview*
- *API Reference*

Overview Virtual LAN (VLAN) is a partitioned and isolated computer network at the data link layer (OSI layer 2). For ethernet network this refers to IEEE 802.1Q

In Zephyr, each individual VLAN is modeled as a virtual network interface. This means that there is an ethernet network interface that corresponds to a real physical ethernet port in the system. A virtual network interface is created for each VLAN, and this virtual network interface connects to the real network interface. This is similar to how Linux implements VLANs. The *eth0* is the real network interface and *vlan0* is a virtual network interface that is run on top of *eth0*.

VLAN support must be enabled at compile time by setting option `CONFIG_NET_VLAN` and `CONFIG_NET_VLAN_COUNT` to reflect how many network interfaces there will be in the system. For example, if there is one network interface without VLAN support, and two with VLAN support, the `CONFIG_NET_VLAN_COUNT` option should be set to 3.

Even if VLAN is enabled in a `prj.conf` file, the VLAN needs to be activated at runtime by the application. The VLAN API provides a *net_eth_vlan_enable()* function to do that. The application needs to give the network interface and desired VLAN tag as a parameter to that function. The VLAN tagging for a given network interface can be disabled by a *net_eth_vlan_disable()* function. The application needs to configure the VLAN network interface itself, such as setting the IP address, etc.

See also the VLAN sample application for API usage example. The source code for that sample application can be found at samples/net/vlan.

The net-shell module contains *net vlan add* and *net vlan del* commands that can be used to enable or disable VLAN tags for a given network interface.

See the IEEE 802.1Q spec for more information about ethernet VLANs.

Related code samples

Virtual LAN
Setup two virtual LAN networks and use net-shell to view the networks' settings.

API Reference

group `vlan_api`

VLAN definitions and helpers.

Defines

`NET_VLAN_TAG_UNSPEC`

Unspecified VLAN tag value.

Functions

static inline uint16_t **net_eth_vlan_get_vid**(uint16_t tci)

Get VLAN identifier from TCI.

Parameters

- **tci** – VLAN tag control information.

Returns

VLAN identifier.

static inline uint8_t **net_eth_vlan_get_dei**(uint16_t tci)

Get Drop Eligible Indicator from TCI.

Parameters

- **tci** – VLAN tag control information.

Returns

Drop eligible indicator.

static inline uint8_t **net_eth_vlan_get_pcp**(uint16_t tci)

Get Priority Code Point from TCI.

Parameters

- **tci** – VLAN tag control information.

Returns

Priority code point.

static inline uint16_t **net_eth_vlan_set_vid**(uint16_t tci, uint16_t vid)

Set VLAN identifier to TCI.

Parameters

- **tci** – VLAN tag control information.
- **vid** – VLAN identifier.

Returns

New TCI value.

static inline uint16_t **net_eth_vlan_set_dei**(uint16_t tci, bool dei)

Set Drop Eligible Indicator to TCI.

Parameters

- **tci** – VLAN tag control information.
- **dei** – Drop eligible indicator.

Returns

New TCI value.

static inline uint16_t **net_eth_vlan_set_pcp**(uint16_t tci, uint8_t pcp)

Set Priority Code Point to TCI.

Parameters

- **tci** – VLAN tag control information.
- **pcp** – Priority code point.

Returns

New TCI value.

Link Layer Discovery Protocol

- *Overview*
- *API Reference*

Overview The Link Layer Discovery Protocol (LLDP) is a vendor-neutral link layer protocol used by network devices for advertising their identity, capabilities, and neighbors on a wired Ethernet network.

For more information, see this LLDP Wikipedia article.

Related code samples

Link Layer Discovery Protocol (LLDP)
Enable LLDP support and setup VLANs.

API Reference

group `lldp`

LLDP definitions and helpers.

Defines

`net_lldp_set_lldpdu`(iface)

Set LLDP protocol data unit (LLDPDU) for the network interface.

Parameters

- `iface` – Network interface

Returns

<0 if error, index in lldp array if iface is found there

`net_lldp_unset_lldpdu`(iface)

Unset LLDP protocol data unit (LLDPDU) for the network interface.

Parameters

- `iface` – Network interface

Typedefs

typedef enum *net_verdict* (*`net_lldp_recv_cb_t`)(struct *net_if* *iface, struct *net_pkt* *pkt)

LLDP Receive packet callback.

Callback gets called upon receiving packet. It is responsible for freeing packet or indicating to the stack that it needs to free packet by returning correct net_verdict.

Returns:

- NET_DROP, if packet was invalid, rejected or we want the stack to free it. In this case the core stack will free the packet.
- NET_OK, if the packet was accepted, in this case the ownership of the *net_pkt* goes to callback and core network stack will forget it.

Enums

enum **net_lldp_tlv_type**

TLV Types.

Please refer to table 8-1 from IEEE 802.1AB standard.

Values:

enumerator **LLDP_TLV_END_LLDPDU** = 0

End Of LLDPDU (optional)

enumerator **LLDP_TLV_CHASSIS_ID** = 1

Chassis ID (mandatory)

enumerator **LLDP_TLV_PORT_ID** = 2

Port ID (mandatory)

enumerator **LLDP_TLV_TTL** = 3

Time To Live (mandatory)

enumerator **LLDP_TLV_PORT_DESC** = 4

Port Description (optional)

enumerator **LLDP_TLV_SYSTEM_NAME** = 5

System Name (optional)

enumerator **LLDP_TLV_SYSTEM_DESC** = 6

System Description (optional)

enumerator **LLDP_TLV_SYSTEM_CAPABILITIES** = 7

System Capability (optional)

enumerator **LLDP_TLV_MANAGEMENT_ADDR** = 8

Management Address (optional)

enumerator **LLDP_TLV_ORG_SPECIFIC** = 127

Org specific TLVs (optional)

Functions

int **net_lldp_config**(struct *net_if* *iface, const struct *net_lldpdu* *lldpdu)

Set the LLDP data unit for a network interface.

Parameters

- **iface** – Network interface
- **lldpdu** – LLDP data unit struct

Returns

0 if ok, <0 if error

int **net_lldp_config_optional**(struct *net_if* *iface, const uint8_t *tlv, size_t len)

Set the Optional LLDP TLVs for a network interface.

Parameters

- **iface** – Network interface
- **tlv** – LLDP optional TLVs following mandatory part
- **len** – Length of the optional TLVs

Returns

0 if ok, <0 if error

void **net_lldp_init**(void)

Initialize LLDP engine.

int **net_lldp_register_callback**(struct *net_if* *iface, *net_lldp_recv_cb_t* cb)

Register LLDP Rx callback function.

Parameters

- **iface** – Network interface
- **cb** – Callback function

Returns

0 if ok, < 0 if error

enum *net_verdict* **net_lldp_recv**(struct *net_if* *iface, struct *net_pkt* *pkt)

Parse LLDP packet.

Parameters

- **iface** – Network interface
- **pkt** – Network packet

Returns

Return the policy for network buffer

struct **net_lldp_chassis_tlv**

#include <*lldp.h*> Chassis ID TLV, see chapter 8.5.2 in IEEE 802.1AB.

Public Members

uint16_t **type_length**

7 bits for type, 9 bits for length

uint8_t **subtype**

ID subtype.

uint8_t **value**[NET_LLDP_CHASSIS_ID_VALUE_LEN]

Chassis ID value.

struct **net_lldp_port_tlv**

#include <*lldp.h*> Port ID TLV, see chapter 8.5.3 in IEEE 802.1AB.

Public Members

uint16_t **type_length**
7 bits for type, 9 bits for length

uint8_t **subtype**
ID subtype.

uint8_t **value**[NET_LLDP_PORT_ID_VALUE_LEN]
Port ID value.

struct **net_lldp_time_to_live_tlv**
#include <*lldp.h*> Time To Live TLV, see chapter 8.5.4 in IEEE 802.1AB.

Public Members

uint16_t **type_length**
7 bits for type, 9 bits for length

uint16_t **ttl**
Time To Live (TTL) value.

struct **net_lldpdu**
#include <*lldp.h*> LLDP Data Unit (LLDPDU) shall contain the following ordered TLVs as stated in "8.2 LLDPDU format" from the IEEE 802.1AB.

Public Members

struct *net_lldp_chassis_tlv* **chassis_id**
Mandatory Chassis TLV.

struct *net_lldp_port_tlv* **port_id**
Mandatory Port TLV.

struct *net_lldp_time_to_live_tlv* **ttl**
Mandatory TTL TLV.

IEEE 802.1Qav

Overview Credit-based shaping is an alternative scheduling algorithm used in network schedulers to achieve fairness when sharing a limited network resource. Zephyr has support for configuring a credit-based shaper described in the IEEE 802.1Qav-2009 standard. Zephyr does not implement the actual shaper; it only provides a way to configure the shaper implemented by the Ethernet device driver.

Enabling 802.1Qav To enable 802.1Qav shaper, the Ethernet device driver must declare that it supports credit-based shaping. The Ethernet driver's capability function must return ETHERNET_QAV value for this purpose. Typically also priority queues ETHERNET_PRIORITY_QUEUES need to be supported.

```
static enum ethernet_hw_caps eth_get_capabilities(const struct device *dev)
{
        ARG_UNUSED(dev);

        return ETHERNET_QAV | ETHERNET_PRIORITY_QUEUES |
               ETHERNET_HW_VLAN | ETHERNET_LINK_10BASE_T |
               ETHERNET_LINK_100BASE_T;
}
```

See sam-e70-xplained board Ethernet driver drivers/ethernet/eth_sam_gmac.c for an example.

Configuring 802.1Qav The application can configure the credit-based shaper like this:

```
#include <zephyr/net/net_if.h>
#include <zephyr/net/ethernet.h>
#include <zephyr/net/ethernet_mgmt.h>

static void qav_set_status(struct net_if *iface,
                           int queue_id, bool enable)
{
        struct ethernet_req_params params;
        int ret;

        memset(&params, 0, sizeof(params));

        params.qav_param.queue_id = queue_id;
        params.qav_param.enabled = enable;
        params.qav_param.type = ETHERNET_QAV_PARAM_TYPE_STATUS;

        /* Disable or enable Qav for a queue */
        ret = net_mgmt(NET_REQUEST_ETHERNET_SET_QAV_PARAM,
                       iface, &params,
                       sizeof(struct ethernet_req_params));
        if (ret) {
                LOG_ERR("Cannot %s Qav for queue %d for interface %p",
                        enable ? "enable" : "disable",
                        queue_id, iface);
        }
}

static void qav_set_bandwidth_and_slope(struct net_if *iface,
                                        int queue_id,
                                        unsigned int bandwidth,
                                        unsigned int idle_slope)
{
        struct ethernet_req_params params;
        int ret;

        memset(&params, 0, sizeof(params));

        params.qav_param.queue_id = queue_id;
        params.qav_param.delta_bandwidth = bandwidth;
        params.qav_param.type = ETHERNET_QAV_PARAM_TYPE_DELTA_BANDWIDTH;

        ret = net_mgmt(NET_REQUEST_ETHERNET_SET_QAV_PARAM,
                       iface, &params,
```

(continues on next page)

(continued from previous page)

```
                        sizeof(struct ethernet_req_params));
        if (ret) {
                LOG_ERR("Cannot set Qav delta bandwidth %u for "
                        "queue %d for interface %p",
                        bandwidth, queue_id, iface);
        }

        params.qav_param.idle_slope = idle_slope;
        params.qav_param.type = ETHERNET_QAV_PARAM_TYPE_IDLE_SLOPE;

        ret = net_mgmt(NET_REQUEST_ETHERNET_SET_QAV_PARAM,
                       iface, &params,
                       sizeof(struct ethernet_req_params));
        if (ret) {
                LOG_ERR("Cannot set Qav idle slope %u for "
                        "queue %d for interface %p",
                        idle_slope, queue_id, iface);
        }
}
```

Overview Ethernet is a networking technology commonly used in local area networks (LAN). For more information, see this Ethernet Wikipedia article.

Zephyr supports following Ethernet features:

- 10, 100 and 1000 Mbit/sec links
- Auto negotiation
- Half/full duplex
- Promiscuous mode
- TX and RX checksum offloading
- MAC address filtering
- *Virtual LANs*
- *Priority queues*
- *IEEE 802.1AS (gPTP)*
- *IEEE 802.1Qav (credit based shaping)*
- *LLDP (Link Layer Discovery Protocol)*

Not all Ethernet device drivers support all of these features. You can see what is supported by `net iface` net-shell command. It will print currently supported Ethernet features.

Related code samples

Inter-VM Shared Memory (ivshmem) Ethernet
Communicate with another ”cell” in the Jailhouse hypervisor using IVSHMEM Ethernet.

Packet socket
Use raw packet sockets over Ethernet.

UDP sender using SO_TXTIME
Control the transmission time of a packet using SO_TXTIME socket option.

API Reference

group `ethernet`

Ethernet support functions.

Defines

`NET_ETH_ADDR_LEN`

Ethernet MAC address length.

`NET_ETH_MINIMAL_FRAME_SIZE`

Minimum Ethernet frame size.

`NET_ETH_MTU`

Ethernet MTU size.

`ETH_NET_DEVICE_INIT`(dev_id, name, init_fn, pm, data, config, prio, api, mtu)

Create an Ethernet network interface and bind it to network device.

Parameters

- `dev_id` – Network device id.
- `name` – The name this instance of the driver exposes to the system.
- `init_fn` – Address to the init function of the driver.
- `pm` – Reference to struct *pm_device* associated with the device. (optional).
- `data` – Pointer to the device's private data.
- `config` – The address to the structure containing the configuration information for this instance of the driver.
- `prio` – The initialization level at which configuration occurs.
- `api` – Provides an initial pointer to the API function struct used by the driver. Can be NULL.
- `mtu` – Maximum transfer unit in bytes for this network interface.

`ETH_NET_DEVICE_INIT_INSTANCE`(dev_id, name, instance, init_fn, pm, data, config, prio, api, mtu)

Create multiple Ethernet network interfaces and bind them to network devices.

If your network device needs more than one instance of a network interface, use this macro below and provide a different instance suffix each time (0, 1, 2, ... or a, b, c ... whatever works for you)

Parameters

- `dev_id` – Network device id.
- `name` – The name this instance of the driver exposes to the system.
- `instance` – Instance identifier.
- `init_fn` – Address to the init function of the driver.
- `pm` – Reference to struct *pm_device* associated with the device. (optional).
- `data` – Pointer to the device's private data.
- `config` – The address to the structure containing the configuration information for this instance of the driver.

- **prio** – The initialization level at which configuration occurs.
- **api** – Provides an initial pointer to the API function struct used by the driver. Can be NULL.
- **mtu** – Maximum transfer unit in bytes for this network interface.

ETH_NET_DEVICE_DT_DEFINE(node_id, init_fn, pm, data, config, prio, api, mtu)

Like ETH_NET_DEVICE_INIT but taking metadata from a devicetree.

Create an Ethernet network interface and bind it to network device.

Parameters

- **node_id** – The devicetree node identifier.
- **init_fn** – Address to the init function of the driver.
- **pm** – Reference to struct *pm_device* associated with the device. (optional).
- **data** – Pointer to the device's private data.
- **config** – The address to the structure containing the configuration information for this instance of the driver.
- **prio** – The initialization level at which configuration occurs.
- **api** – Provides an initial pointer to the API function struct used by the driver. Can be NULL.
- **mtu** – Maximum transfer unit in bytes for this network interface.

ETH_NET_DEVICE_DT_INST_DEFINE(inst, ...)

Like ETH_NET_DEVICE_DT_DEFINE for an instance of a DT_DRV_COMPAT compatible.

Parameters

- **inst** – instance number. This is replaced by DT_DRV_COMPAT(inst) in the call to ETH_NET_DEVICE_DT_DEFINE.
- **...** – other parameters as expected by ETH_NET_DEVICE_DT_DEFINE.

Enums

enum **ethernet_hw_caps**

Ethernet hardware capabilities.

Values:

enumerator **ETHERNET_HW_TX_CHKSUM_OFFLOAD** = *BIT*(0)

TX Checksum offloading supported for all of IPv4, UDP, TCP.

enumerator **ETHERNET_HW_RX_CHKSUM_OFFLOAD** = *BIT*(1)

RX Checksum offloading supported for all of IPv4, UDP, TCP.

enumerator **ETHERNET_HW_VLAN** = *BIT*(2)

VLAN supported.

enumerator **ETHERNET_AUTO_NEGOTIATION_SET** = *BIT*(3)

Enabling/disabling auto negotiation supported.

enumerator ETHERNET_LINK_10BASE_T = *BIT*(4)
10 Mbits link supported

enumerator ETHERNET_LINK_100BASE_T = *BIT*(5)
100 Mbits link supported

enumerator ETHERNET_LINK_1000BASE_T = *BIT*(6)
1 Gbits link supported

enumerator ETHERNET_DUPLEX_SET = *BIT*(7)
Changing duplex (half/full) supported.

enumerator ETHERNET_PTP = *BIT*(8)
IEEE 802.1AS (gPTP) clock supported.

enumerator ETHERNET_QAV = *BIT*(9)
IEEE 802.1Qav (credit-based shaping) supported.

enumerator ETHERNET_PROMISC_MODE = *BIT*(10)
Promiscuous mode supported.

enumerator ETHERNET_PRIORITY_QUEUES = *BIT*(11)
Priority queues available.

enumerator ETHERNET_HW_FILTERING = *BIT*(12)
MAC address filtering supported.

enumerator ETHERNET_LLDP = *BIT*(13)
Link Layer Discovery Protocol supported.

enumerator ETHERNET_HW_VLAN_TAG_STRIP = *BIT*(14)
VLAN Tag stripping.

enumerator ETHERNET_DSA_SLAVE_PORT = *BIT*(15)
DSA switch slave port.

enumerator ETHERNET_DSA_MASTER_PORT = *BIT*(16)
DSA switch master port.

enumerator ETHERNET_QBV = *BIT*(17)
IEEE 802.1Qbv (scheduled traffic) supported.

enumerator ETHERNET_QBU = *BIT*(18)
IEEE 802.1Qbu (frame preemption) supported.

enumerator ETHERNET_TXTIME = *BIT*(19)
TXTIME supported.

enumerator **ETHERNET_TXINJECTION_MODE** = *BIT*(20)

TX-Injection supported.

enum **ethernet_if_types**

Types of Ethernet L2.

Values:

enumerator **L2_ETH_IF_TYPE_ETHERNET**

IEEE 802.3 Ethernet (default)

enumerator **L2_ETH_IF_TYPE_WIFI**

IEEE 802.11 Wi-Fi.

enum **ethernet_checksum_support**

Protocols that are supported by checksum offloading.

Values:

enumerator **ETHERNET_CHECKSUM_SUPPORT_NONE** = NET_IF_CHECKSUM_NONE_BIT

Device does not support any L3/L4 checksum offloading.

enumerator **ETHERNET_CHECKSUM_SUPPORT_IPV4_HEADER** = NET_IF_CHECKSUM_IPV4_HEADER_BIT

Device supports checksum offloading for the IPv4 header.

enumerator **ETHERNET_CHECKSUM_SUPPORT_IPV4_ICMP** = NET_IF_CHECKSUM_IPV4_ICMP_BIT

Device supports checksum offloading for ICMPv4 payload (implies IPv4 header)

enumerator **ETHERNET_CHECKSUM_SUPPORT_IPV6_HEADER** = NET_IF_CHECKSUM_IPV6_HEADER_BIT

Device supports checksum offloading for the IPv6 header.

enumerator **ETHERNET_CHECKSUM_SUPPORT_IPV6_ICMP** = NET_IF_CHECKSUM_IPV6_ICMP_BIT

Device supports checksum offloading for ICMPv6 payload (implies IPv6 header)

enumerator **ETHERNET_CHECKSUM_SUPPORT_TCP** = NET_IF_CHECKSUM_TCP_BIT

Device supports TCP checksum offloading for all supported IP protocols.

enumerator **ETHERNET_CHECKSUM_SUPPORT_UDP** = NET_IF_CHECKSUM_UDP_BIT

Device supports UDP checksum offloading for all supported IP protocols.

Functions

static inline bool **net_eth_is_addr_broadcast**(struct *net_eth_addr* *addr)

Check if the Ethernet MAC address is a broadcast address.

Parameters

- **addr** – A valid pointer to a Ethernet MAC address.

Returns
true if address is a broadcast address, false if not

static inline bool **net_eth_is_addr_all_zeroes**(struct *net_eth_addr* *addr)
Check if the Ethernet MAC address is a all zeroes address.

Parameters
- **addr** – A valid pointer to an Ethernet MAC address.

Returns
true if address is an all zeroes address, false if not

static inline bool **net_eth_is_addr_unspecified**(struct *net_eth_addr* *addr)
Check if the Ethernet MAC address is unspecified.

Parameters
- **addr** – A valid pointer to a Ethernet MAC address.

Returns
true if address is unspecified, false if not

static inline bool **net_eth_is_addr_multicast**(struct *net_eth_addr* *addr)
Check if the Ethernet MAC address is a multicast address.

Parameters
- **addr** – A valid pointer to a Ethernet MAC address.

Returns
true if address is a multicast address, false if not

static inline bool **net_eth_is_addr_group**(struct *net_eth_addr* *addr)
Check if the Ethernet MAC address is a group address.

Parameters
- **addr** – A valid pointer to a Ethernet MAC address.

Returns
true if address is a group address, false if not

static inline bool **net_eth_is_addr_valid**(struct *net_eth_addr* *addr)
Check if the Ethernet MAC address is valid.

Parameters
- **addr** – A valid pointer to a Ethernet MAC address.

Returns
true if address is valid, false if not

static inline bool **net_eth_is_addr_lldp_multicast**(struct *net_eth_addr* *addr)
Check if the Ethernet MAC address is a LLDP multicast address.

Parameters
- **addr** – A valid pointer to a Ethernet MAC address.

Returns
true if address is a LLDP multicast address, false if not

static inline bool **net_eth_is_addr_ptp_multicast**(struct *net_eth_addr* *addr)
Check if the Ethernet MAC address is a PTP multicast address.

Parameters
- **addr** – A valid pointer to a Ethernet MAC address.

Returns
true if address is a PTP multicast address, false if not

const struct *net_eth_addr* ***net_eth_broadcast_addr**(void)

Return Ethernet broadcast address.

Returns
Ethernet broadcast address.

void **net_eth_ipv4_mcast_to_mac_addr**(const struct *in_addr* *ipv4_addr, struct *net_eth_addr* *mac_addr)

Convert IPv4 multicast address to Ethernet address.

Parameters

- **ipv4_addr** – IPv4 multicast address
- **mac_addr** – Output buffer for Ethernet address

void **net_eth_ipv6_mcast_to_mac_addr**(const struct *in6_addr* *ipv6_addr, struct *net_eth_addr* *mac_addr)

Convert IPv6 multicast address to Ethernet address.

Parameters

- **ipv6_addr** – IPv6 multicast address
- **mac_addr** – Output buffer for Ethernet address

static inline enum *ethernet_hw_caps* **net_eth_get_hw_capabilities**(struct *net_if* *iface)

Return ethernet device hardware capability information.

Parameters

- **iface** – Network interface

Returns
Hardware capabilities

static inline int **net_eth_get_hw_config**(struct *net_if* *iface, enum ethernet_config_type type, struct ethernet_config *config)

Return ethernet device hardware configuration information.

Parameters

- **iface** – Network interface
- **type** – configuration type
- **config** – Ethernet configuration

Returns
0 if ok, <0 if error

static inline int **net_eth_vlan_enable**(struct *net_if* *iface, uint16_t tag)

Add VLAN tag to the interface.

Parameters

- **iface** – Interface to use.
- **tag** – VLAN tag to add

Returns
0 if ok, <0 if error

static inline int **net_eth_vlan_disable**(struct *net_if* *iface, uint16_t tag)

Remove VLAN tag from the interface.

Parameters

- **iface** – Interface to use.
- **tag** – VLAN tag to remove

Returns

0 if ok, <0 if error

static inline uint16_t **net_eth_get_vlan_tag**(struct *net_if* *iface)

Return VLAN tag specified to network interface.

Note that the interface parameter must be the VLAN interface, and not the Ethernet one.

Parameters

- **iface** – VLAN network interface.

Returns

VLAN tag for this interface or NET_VLAN_TAG_UNSPEC if VLAN is not configured for that interface.

static inline struct *net_if* ***net_eth_get_vlan_iface**(struct *net_if* *iface, uint16_t tag)

Return network interface related to this VLAN tag.

Parameters

- **iface** – Main network interface (not the VLAN one).
- **tag** – VLAN tag

Returns

Network interface related to this tag or NULL if no such interface exists.

static inline struct *net_if* ***net_eth_get_vlan_main**(struct *net_if* *iface)

Return main network interface that is attached to this VLAN tag.

Parameters

- **iface** – VLAN network interface. This is used to get the pointer to ethernet L2 context

Returns

Network interface related to this tag or NULL if no such interface exists.

static inline bool **net_eth_is_vlan_enabled**(struct ethernet_context *ctx, struct *net_if* *iface)

Check if there are any VLAN interfaces enabled to this specific Ethernet network interface.

Note that the iface must be the actual Ethernet interface and not the virtual VLAN interface.

Parameters

- **ctx** – Ethernet context
- **iface** – Ethernet network interface

Returns

True if there are enabled VLANs for this network interface, false if not.

static inline bool **net_eth_get_vlan_status**(struct *net_if* *iface)

Get VLAN status for a given network interface (enabled or not).

Parameters

- **iface** – Network interface

Returns

True if VLAN is enabled for this network interface, false if not.

static inline bool **net_eth_is_vlan_interface**(struct *net_if* *iface)

Check if the given interface is a VLAN interface.

Parameters

- **iface** – Network interface

Returns

True if this network interface is VLAN one, false if not.

void **net_eth_carrier_on**(struct *net_if* *iface)

Inform ethernet L2 driver that ethernet carrier is detected.

This happens when cable is connected.

Parameters

- **iface** – Network interface

void **net_eth_carrier_off**(struct *net_if* *iface)

Inform ethernet L2 driver that ethernet carrier was lost.

This happens when cable is disconnected.

Parameters

- **iface** – Network interface

int **net_eth_promisc_mode**(struct *net_if* *iface, bool enable)

Set promiscuous mode either ON or OFF.

Parameters

- **iface** – Network interface
- **enable** – on (true) or off (false)

Returns

0 if mode set or unset was successful, <0 otherwise.

int **net_eth_txinjection_mode**(struct *net_if* *iface, bool enable)

Set TX-Injection mode either ON or OFF.

Parameters

- **iface** – Network interface
- **enable** – on (true) or off (false)

Returns

0 if mode set or unset was successful, <0 otherwise.

int **net_eth_mac_filter**(struct *net_if* *iface, struct *net_eth_addr* *mac, enum ethernet_filter_type type, bool enable)

Set or unset HW filtering for MAC address `mac`.

Parameters

- **iface** – Network interface
- **mac** – Pointer to an ethernet MAC address

- **type** – Filter type, either source or destination
- **enable** – Set (true) or unset (false)

Returns
0 if filter set or unset was successful, <0 otherwise.

static inline const struct *device* ***net_eth_get_ptp_clock**(struct *net_if* *iface)

Return PTP clock that is tied to this ethernet network interface.

Parameters

- **iface** – Network interface

Returns
Pointer to PTP clock if found, NULL if not found or if this ethernet interface does not support PTP.

const struct *device* ***net_eth_get_ptp_clock_by_index**(int index)

Return PTP clock that is tied to this ethernet network interface index.

Parameters

- **index** – Network interface index

Returns
Pointer to PTP clock if found, NULL if not found or if this ethernet interface index does not support PTP.

static inline int **net_eth_get_ptp_port**(struct *net_if* *iface)

Return PTP port number attached to this interface.

Parameters

- **iface** – Network interface

Returns
Port number, no such port if < 0

static inline void **net_eth_set_ptp_port**(struct *net_if* *iface, int port)

Set PTP port number attached to this interface.

Parameters

- **iface** – Network interface
- **port** – Port number to set

static inline bool **net_eth_type_is_wifi**(struct *net_if* *iface)

Check if the Ethernet L2 network interface can perform Wi-Fi.

Parameters

- **iface** – Pointer to network interface

Returns
True if interface supports Wi-Fi, False otherwise.

struct **net_eth_addr**

#include <ethernet.h> Ethernet address.

Public Members

uint8_t **addr**[6U]

Buffer storing the address.

struct **ethernet_t1s_param**

#include <ethernet.h> Ethernet T1S specific parameters.

Public Members

enum ethernet_t1s_param_type **type**

Type of T1S parameter.

bool **enable**

T1S PLCA enabled.

uint8_t **node_id**

T1S PLCA node id range: 0 to 254.

uint8_t **node_count**

T1S PLCA node count range: 1 to 255.

uint8_t **burst_count**

T1S PLCA burst count range: 0x0 to 0xFF.

uint8_t **burst_timer**

T1S PLCA burst timer.

uint8_t **to_timer**

T1S PLCA TO value.

struct *ethernet_t1s_param* **plca**

PLCA is the Physical Layer (PHY) Collision Avoidance technique employed with multidrop 10Base-T1S standard.

The PLCA parameters are described in standard [1] as registers in memory map 4 (MMS = 4) (point 9.6).

IDVER (PLCA ID Version) CTRL0 (PLCA Control 0) CTRL1 (PLCA Control 1) STATUS (PLCA Status) TOTMR (PLCA TO Control) BURST (PLCA Burst Control)

Those registers are implemented by each OA TC6 compliant vendor (like for e.g. LAN865x - e.g. [2]).

Documents: [1] - “OPEN Alliance 10BASE-T1x MAC-PHY Serial

Interface” (ver. 1.1) [2] - “DS60001734C” - LAN865x data sheet

struct **ethernet_qav_param**

#include <ethernet.h> Ethernet Qav specific parameters.

Public Members

int **queue_id**
ID of the priority queue to use.

enum ethernet_qav_param_type **type**
Type of Qav parameter.

bool **enabled**
True if Qav is enabled for queue.

unsigned int **delta_bandwidth**
Delta Bandwidth (percentage of bandwidth)

unsigned int **idle_slope**
Idle Slope (bits per second)

unsigned int **oper_idle_slope**
Oper Idle Slope (bits per second)

unsigned int **traffic_class**
Traffic class the queue is bound to.

struct **ethernet_qbv_param**
#include <ethernet.h> Ethernet Qbv specific parameters.

Public Members

int **port_id**
Port id.

enum ethernet_qbv_param_type **type**
Type of Qbv parameter.

enum ethernet_qbv_state_type **state**
What state (Admin/Oper) parameters are these.

bool **enabled**
True if Qbv is enabled or not.

bool **gate_status**[NET_TC_TX_COUNT]
True = open, False = closed.

enum ethernet_gate_state_operation **operation**
GateState operation.

uint32_t **time_interval**
Time interval ticks (nanoseconds)

uint16_t **row**
Gate control list row.

struct *ethernet_qbv_param* **gate_control**
Gate control information.

uint32_t **gate_control_list_len**
Number of entries in gate control list.

struct *net_ptp_extended_time* **base_time**
Base time.

struct *net_ptp_time* **cycle_time**
Cycle time.

uint32_t **extension_time**
Extension time (nanoseconds)

struct **ethernet_qbu_param**
#include <ethernet.h> Ethernet Qbu specific parameters.

Public Members

int **port_id**
Port id.

enum ethernet_qbu_param_type **type**
Type of Qbu parameter.

uint32_t **hold_advance**
Hold advance (nanoseconds)

uint32_t **release_advance**
Release advance (nanoseconds)

enum ethernet_qbu_preempt_status **frame_preempt_statuses**[NET_TC_TX_COUNT]
sequence of framePreemptionAdminStatus values

bool **enabled**
True if Qbu is enabled or not.

bool **link_partner_status**
Link partner status (from Qbr)

uint8_t **additional_fragment_size**
Additional fragment size (from Qbr).
The minimum non-final fragment size is (additional_fragment_size + 1) * 64 octets

struct **ethernet_filter**

#include <*ethernet.h*> Ethernet filter description.

Public Members

enum ethernet_filter_type **type**

Type of filter.

struct *net_eth_addr* **mac_address**

MAC address to filter.

bool **set**

Set (true) or unset (false) the filter.

struct **ethernet_txtime_param**

#include <*ethernet.h*> Ethernet TXTIME specific parameters.

Public Members

enum ethernet_txtime_param_type **type**

Type of TXTIME parameter.

int **queue_id**

Queue number for configuring TXTIME.

bool **enable_txtime**

Enable or disable TXTIME per queue.

struct **ethernet_api**

#include <*ethernet.h*> Ethernet L2 API operations.

Public Members

struct net_if_api **iface_api**

The net_if_api must be placed in first position in this struct so that we are compatible with network interface API.

int (***start**)(const struct *device* *dev)

Collect optional ethernet specific statistics.

This pointer should be set by driver if statistics needs to be collected for that driver. Start the device

int (***stop**)(const struct *device* *dev)

Stop the device.

enum *ethernet_hw_caps* (***get_capabilities***)(const struct *device* *dev)

Get the device capabilities.

int (***set_config***)(const struct *device* *dev, enum ethernet_config_type type, const struct ethernet_config *config)

Set specific hardware configuration.

int (***get_config***)(const struct *device* *dev, enum ethernet_config_type type, struct ethernet_config *config)

Get hardware specific configuration.

int (***send***)(const struct *device* *dev, struct *net_pkt* *pkt)

The IP stack will call this function when a VLAN tag is enabled or disabled.

If enable is set to true, then the VLAN tag was added, if it is false then the tag was removed. The driver can utilize this information if needed. Return ptp_clock device that is tied to this ethernet device Send a network packet

struct **ethernet_lldp**

#include <ethernet.h> Ethernet LLDP specific parameters.

Public Members

sys_snode_t **node**

Used for track timers.

const struct *net_lldpdu* ***lldpdu**

LLDP Data Unit mandatory TLVs for the interface.

const uint8_t ***optional_du**

LLDP Data Unit optional TLVs for the interface.

size_t **optional_len**

Length of the optional Data Unit TLVs.

struct *net_if* ***iface**

Network interface that has LLDP supported.

int64_t **tx_timer_start**

LLDP TX timer start time.

uint32_t **tx_timer_timeout**

LLDP TX timeout.

net_lldp_recv_cb_t **cb**

LLDP RX callback function.

group **ethernet_mii**

Ethernet MII (media independent interface) functions.

Defines

`MII_BMCR`
Basic Mode Control Register.

`MII_BMSR`
Basic Mode Status Register.

`MII_PHYID1R`
PHY ID 1 Register.

`MII_PHYID2R`
PHY ID 2 Register.

`MII_ANAR`
Auto-Negotiation Advertisement Register.

`MII_ANLPAR`
Auto-Negotiation Link Partner Ability Reg.

`MII_ANER`
Auto-Negotiation Expansion Register.

`MII_ANNPTR`
Auto-Negotiation Next Page Transmit Register.

`MII_ANLPRNPR`
Auto-Negotiation Link Partner Received Next Page Reg.

`MII_1KTCR`
1000BASE-T Control Register

`MII_1KSTSR`
1000BASE-T Status Register

`MII_MMD_ACR`
MMD Access Control Register.

`MII_MMD_AADR`
MMD Access Address Data Register.

`MII_ESTAT`
Extended Status Register.

`MII_BMCR_RESET`
PHY reset.

`MII_BMCR_LOOPBACK`
enable loopback mode

`MII_BMCR_SPEED_LSB`
: 10=1000Mbps 01=100Mbps; 00=10Mbps

`MII_BMCR_AUTONEG_ENABLE`
: Auto-Negotiation enable.

`MII_BMCR_POWER_DOWN`
: power down mode

`MII_BMCR_ISOLATE`
: isolate electrically PHY from MII

`MII_BMCR_AUTONEG_RESTART`
: restart auto-negotiation

`MII_BMCR_DUPLEX_MODE`
: full duplex mode

`MII_BMCR_SPEED_MSB`
: 10=1000Mbps 01=100Mbps; 00=10Mbps

`MII_BMCR_SPEED_MASK`
: Link Speed Field.

`MII_BMCR_SPEED_10`
: select speed 10 Mb/s

`MII_BMCR_SPEED_100`
: select speed 100 Mb/s

`MII_BMCR_SPEED_1000`
: select speed 1000 Mb/s

`MII_BMSR_100BASE_T4`
: 100BASE-T4 capable

`MII_BMSR_100BASE_X_FULL`
: 100BASE-X full duplex capable

`MII_BMSR_100BASE_X_HALF`
: 100BASE-X half duplex capable

`MII_BMSR_10_FULL`
: 10 Mb/s full duplex capable

`MII_BMSR_10_HALF`
: 10 Mb/s half duplex capable

`MII_BMSR_100BASE_T2_FULL`
: 100BASE-T2 full duplex capable

`MII_BMSR_100BASE_T2_HALF`
: 100BASE-T2 half duplex capable

`MII_BMSR_EXTEND_STATUS`
: extend status information in reg 15

`MII_BMSR_MF_PREAMB_SUPPR`
: PHY accepts management frames with preamble suppressed.

`MII_BMSR_AUTONEG_COMPLETE`
: Auto-negotiation process completed.

`MII_BMSR_REMOTE_FAULT`
: remote fault detected

`MII_BMSR_AUTONEG_ABILITY`
: PHY is able to perform Auto-Negotiation.

`MII_BMSR_LINK_STATUS`
: link is up

`MII_BMSR_JABBER_DETECT`
: jabber condition detected

`MII_BMSR_EXTEND_CAPAB`
: extended register capabilities

`MII_ADVERTISE_NEXT_PAGE`
: next page

`MII_ADVERTISE_LPACK`
: link partner acknowledge response

`MII_ADVERTISE_REMOTE_FAULT`
: remote fault

`MII_ADVERTISE_ASYM_PAUSE`
: try for asymmetric pause

`MII_ADVERTISE_PAUSE`
: try for pause

`MII_ADVERTISE_100BASE_T4`
: try for 100BASE-T4 support

`MII_ADVERTISE_100_FULL`
: try for 100BASE-X full duplex support

`MII_ADVERTISE_100_HALF`
: try for 100BASE-X support

`MII_ADVERTISE_10_FULL`
: try for 10 Mb/s full duplex support

`MII_ADVERTISE_10_HALF`
: try for 10 Mb/s half duplex support

`MII_ADVERTISE_SEL_MASK`
: Selector Field Mask.

`MII_ADVERTISE_SEL_IEEE_802_3`
: Selector Field.

`MII_ADVERTISE_1000_FULL`
: try for 1000BASE-T full duplex support

`MII_ADVERTISE_1000_HALF`
: try for 1000BASE-T half duplex support

`MII_ADVERTISE_ALL`
: Advertise all speeds.

`MII_ESTAT_1000BASE_X_FULL`
: 1000BASE-X full-duplex capable

`MII_ESTAT_1000BASE_X_HALF`
: 1000BASE-X half-duplex capable

`MII_ESTAT_1000BASE_T_FULL`
: 1000BASE-T full-duplex capable

`MII_ESTAT_1000BASE_T_HALF`
: 1000BASE-T half-duplex capable

IEEE 802.15.4

- *Introduction*
- *API Reference*
  - *IEEE 802.15.4 API Overview*
  - *IEEE 802.15.4 Management API*
  - *IEEE 802.15.4 Driver API*

- *IEEE 802.15.4 L2 / Native Stack API*
- *OpenThread L2 Adaptation Layer API*

Introduction IEEE 802.15.4 is a technical standard which defines the operation of low-rate wireless personal area networks (LR-WPANs). For a more detailed overview of this standard, see the IEEE 802.15.4 Wikipedia article.

The most recent version of the standard is accessible through the IEEE GET Program. You need to create a free IEEE account and can then downloading it.

We're currently following the IEEE 802.15.4-2020 specification. This version is backwards compatible with IEEE 802.15.4-2015, parts of which are contained in the Thread protocol stack. The 2020 version also includes prior extensions that were accepted into the standard, namely IEEE 802.15.4g (SUN FSK) and IEEE 802.15.4e (TSCH) which are of relevance to industrial IoT and automation. For recent developments in UWB ranging technology, see IEEE 802.15.4z which is not yet integrated into the standard's mainline.

Whenever sections from the standard are cited in the documentation, they refer to IEEE 802.15.4-2020 section, table and figure numbering - unless otherwise specified.

Zephyr supports both, native IEEE 802.15.4 and Thread, with 6LoWPAN. Zephyr's *Thread protocol* implementation is based on OpenThread. The IPv6 header compression in 6LoWPAN is used for native IEEE 802.15.4.

API Reference

IEEE 802.15.4 API Overview Gives an introduction and overview over the whole IEEE 802.15.4 subsystem and all of its APIs, configuration and user interfaces for all audiences.

Related code samples

802.15.4 "serial-radio"
Implement a slip-radio device for Contiki-based border routers.

802.15.4 USB
Implement a device that exposes an IEEE 802.15.4 radio over USB.

group `ieee802154`

IEEE 802.15.4 native and OpenThread L2, configuration, management and driver APIs.

Since
1.0

Version
0.8.0

The IEEE 802.15.4 and Thread subsystems comprise the OpenThread L2 subsystem, the native IEEE 802.15.4 L2 subsystem ("Soft" MAC), a mostly vendor and protocol agnostic driver API shared between the OpenThread and native L2 stacks ("Hard" MAC and PHY) as well as several APIs to configure the subsystem (shell, net management, Kconfig, devicetree, etc.).

The **OpenThread subsystem API** integrates the external OpenThread stack into Zephyr. It builds upon Zephyr's native IEEE 802.15.4 driver API.

The **native IEEE 802.15.4 subsystem APIs** are exposed at different levels and address several audiences:

- shell (end users, application developers):
  - a set of IEEE 802.15.4 shell commands (see `shell> ieee802154 help`)
- application API (application developers):
  - IPv6, DGRAM and RAW sockets for actual peer-to-peer, multicast and broadcast data exchange between nodes including connection specific configuration (sample coming soon, see https://github.com/linux-wpan/wpan-tools/tree/master/examples for now which inspired our API and therefore has a similar socket API),
  - Kconfig and devicetree configuration options (net config library extension, subsystem-wide MAC and PHY Kconfig/DT options, driver/vendor specific Kconfig/DT options, watch out for options prefixed with IEEE802154/ieee802154),
  - Network Management: runtime configuration of the IEEE 802.15.4 protocols stack at the MAC (L2) and PHY (L1) levels (see *IEEE 802.15.4 Net Management*),
- L2 integration (subsystem contributors):
  - see *IEEE 802.15.4 L2*
  - implementation of Zephyr's internal L2-level socket and network context abstractions (context/socket operations, see *Network L2 Abstraction Layer*),
  - protocol-specific extension to the interface structure (see *Network Interface abstraction layer*)
  - protocol-specific extensions to the network packet structure (see *Network Packet Library*),
- OpenThread and native IEEE 802.15.4 share a common **driver API** (driver maintainers/contributors):
  - see *IEEE 802.15.4 Drivers*
  - a basic, mostly PHY-level driver API to be implemented by all drivers,
  - several "hard MAC" (hardware/firmware offloading) extension points for performance critical or timing sensitive aspects of the protocol

IEEE 802.15.4 Management API This is the main subsystem-specific API of interest to IEEE 802.15.4 **application developers** as it allows to configure the IEEE 802.15.4 subsystem at runtime. Other relevant interfaces for application developers are the typical shell, socket, Kconfig and devicetree APIs that can be accessed through Zephyr's generic subsystem-independent documentation. Look out for IEEE802154/ieee802154 prefixes there.

group `ieee802154_mgmt`

IEEE 802.15.4 net management library.

Since
1.0

Version
0.8.0

The IEEE 802.15.4 net management library provides runtime configuration features that applications can interface with directly.

Most of these commands are also accessible via shell commands. See the shell's help feature (`shell> ieee802154 help`).

Note

All section, table and figure references are to the IEEE 802.15.4-2020 standard.

Command Macros

IEEE 802.15.4 net management commands.

These IEEE 802.15.4 subsystem net management commands can be called by applications via *Network Management* macro.

All attributes and parameters are given in CPU byte order (scalars) or big endian (byte arrays) unless otherwise specified.

The following IEEE 802.15.4 MAC management service primitives are referenced in this enumeration:

- MLME-ASSOCIATE.request, see section 8.2.3
- MLME-DISASSOCIATE.request, see section 8.2.4
- MLME-SET/GET.request, see section 8.2.6
- MLME-SCAN.request, see section 8.2.11

The following IEEE 802.15.4 MAC data service primitives are referenced in this enumeration:

- MLME-DATA.request, see section 8.3.2

MAC PIB attributes (mac.../sec...): see sections 8.4.3 and 9.5. PHY PIB attributes (phy...): see section 11.3. Both are accessed through MLME-SET/GET primitives.

`NET_REQUEST_IEEE802154_SET_ACK`
: Sets AckTx for all subsequent MLME-DATA (aka TX) requests.

`NET_REQUEST_IEEE802154_UNSET_ACK`
: Unsets AckTx for all subsequent MLME-DATA requests.

`NET_REQUEST_IEEE802154_PASSIVE_SCAN`
: MLME-SCAN(PASSIVE, ...) request.

  See *ieee802154_req_params* for associated command parameters.

`NET_REQUEST_IEEE802154_ACTIVE_SCAN`
: MLME-SCAN(ACTIVE, ...) request.

  See *ieee802154_req_params* for associated command parameters.

`NET_REQUEST_IEEE802154_CANCEL_SCAN`
: Cancels an ongoing MLME-SCAN(...) command (non-standard).

`NET_REQUEST_IEEE802154_ASSOCIATE`
: MLME-ASSOCIATE(...) request.

`NET_REQUEST_IEEE802154_DISASSOCIATE`
: MLME-DISASSOCIATE(...) request.

`NET_REQUEST_IEEE802154_SET_CHANNEL`
: MLME-SET(phyCurrentChannel) request.

`NET_REQUEST_IEEE802154_GET_CHANNEL`
: MLME-GET(phyCurrentChannel) request.

`NET_REQUEST_IEEE802154_SET_PAN_ID`
: MLME-SET(macPanId) request.

`NET_REQUEST_IEEE802154_GET_PAN_ID`
: MLME-GET(macPanId) request.

`NET_REQUEST_IEEE802154_SET_EXT_ADDR`
: Sets the extended interface address (non-standard), see sections 7.1 and 8.4.3.1, in big endian byte order.

`NET_REQUEST_IEEE802154_GET_EXT_ADDR`
: like MLME-GET(macExtendedAddress) but in big endian byte order

`NET_REQUEST_IEEE802154_SET_SHORT_ADDR`
: MLME-SET(macShortAddress) request, only allowed for co-ordinators.

`NET_REQUEST_IEEE802154_GET_SHORT_ADDR`
: MLME-GET(macShortAddress) request.

`NET_REQUEST_IEEE802154_GET_TX_POWER`
: MLME-SET(phyUnicastTxPower/phyBroadcastTxPower) request (currently not distinguished)

`NET_REQUEST_IEEE802154_SET_TX_POWER`
: MLME-GET(phyUnicastTxPower/phyBroadcastTxPower) request.

`NET_REQUEST_IEEE802154_SET_SECURITY_SETTINGS`
: Configures basic sec* MAC PIB attributes, implies macSecurityEnabled=true.

  See *ieee802154_security_params* for associated command parameters.

`NET_REQUEST_IEEE802154_GET_SECURITY_SETTINGS`
: Gets the configured sec* attributes.

  See *ieee802154_security_params* for associated command parameters.

Event Macros

IEEE 802.15.4 net management events.

These IEEE 802.15.4 subsystem net management events can be subscribed to by applications via *net_mgmt_init_event_callback*, *net_mgmt_add_event_callback* and *net_mgmt_del_event_callback*.

NET_EVENT_IEEE802154_SCAN_RESULT

Signals the result of the *NET_REQUEST_IEEE802154_ACTIVE_SCAN* or *NET_REQUEST_IEEE802154_PASSIVE_SCAN* net management commands.

See *ieee802154_req_params* for associated event parameters.

struct **ieee802154_req_params**

#include <*ieee802154_mgmt.h*> Scanning parameters.

Used to request a scan and get results as well, see section 8.2.11.2

Public Members

uint32_t **channel_set**

The set of channels to scan, use above macros to manage it.

uint32_t **duration**

Duration of scan, per-channel, in milliseconds.

uint16_t **channel**

Current channel in use as a result.

uint16_t **pan_id**

Current pan_id in use as a result.

uint16_t **short_addr**

in CPU byte order

uint8_t **addr**[*IEEE802154_MAX_ADDR_LENGTH*]

in big endian

union ieee802154_req_params

Result address.

uint8_t **len**

length of address

uint8_t **lqi**

Link quality information, between 0 and 255.

struct **ieee802154_security_params**

#include <*ieee802154_mgmt.h*> Security parameters.

Used to setup the link-layer security settings, see tables 9-9 and 9-10 in section 9.5.

Public Members

uint8_t **key**[16]

secKeyDescriptor.secKey

uint8_t **key_len**

Key length of 16 bytes is mandatory for standards conformance.

uint8_t **key_mode**

secKeyIdMode

uint8_t **level**

Used instead of a frame-specific SecurityLevel parameter when constructing the auxiliary security header.

IEEE 802.15.4 Driver API This is the main API of interest to IEEE 802.15.4 **driver developers**.

group **ieee802154_driver**

IEEE 802.15.4 driver API.

Since

1.0

Version

0.8.0

This API provides a common representation of vendor-specific hardware and firmware to the native IEEE 802.15.4 L2 and OpenThread stacks. **Application developers should never interface directly with this API.** It is of interest to driver maintainers only.

The IEEE 802.15.4 driver API consists of two separate parts:

- a basic, mostly PHY-level driver API to be implemented by all drivers,
- several optional MAC-level extension points to offload performance critical or timing sensitive aspects at MAC level to the driver hardware or firmware ("hard" MAC).

Implementing the basic driver API will ensure integration with the native L2 stack as well as basic support for OpenThread. Depending on the hardware, offloading to vendor-specific hardware or firmware features may be required to achieve full compliance with the Thread protocol or IEEE 802.15.4 subprotocols (e.g. fast enough ACK packages, precise timing of timed TX/RX in the TSCH or CSL subprotocols).

Whether or not MAC-level offloading extension points need to be implemented is to be decided by individual driver maintainers. Upper layers SHOULD provide a "soft" MAC fallback whenever possible.

Note

All section, table and figure references are to the IEEE 802.15.4-2020 standard.

IEEE 802.15.4, section 7.4.2: MAC header information elements

enum **ieee802154_ie_type**

Information Element Types.

See sections 7.4.2.1 and 7.4.3.1.

Values:

enumerator **IEEE802154_IE_TYPE_HEADER** = 0x0

Header type.

enumerator **IEEE802154_IE_TYPE_PAYLOAD**

Payload type.

enum **ieee802154_header_ie_element_id**

Header Information Element IDs.

See section 7.4.2.1, table 7-7, partial list, only IEs actually used are implemented.

Values:

enumerator **IEEE802154_HEADER_IE_ELEMENT_ID_VENDOR_SPECIFIC_IE** = 0x00

Vendor specific IE.

enumerator **IEEE802154_HEADER_IE_ELEMENT_ID_CSL_IE** = 0x1a

CSL IE.

enumerator **IEEE802154_HEADER_IE_ELEMENT_ID_RIT_IE** = 0x1b

RIT IE.

enumerator **IEEE802154_HEADER_IE_ELEMENT_ID_RENDEZVOUS_TIME_IE** = 0x1d

Rendezvous time IE.

enumerator **IEEE802154_HEADER_IE_ELEMENT_ID_TIME_CORRECTION_IE** = 0x1e

Time correction IE.

enumerator **IEEE802154_HEADER_IE_ELEMENT_ID_HEADER_TERMINATION_1** = 0x7e

Header termination 1.

enumerator **IEEE802154_HEADER_IE_ELEMENT_ID_HEADER_TERMINATION_2** = 0x7f

Header termination 2.

static inline int16_t **ieee802154_header_ie_get_time_correction_us**(struct *ieee802154_header_ie_time_correcti* *ie)

Retrieve the time correction value in microseconds from a Time Correction IE, see section 7.4.2.7.

Parameters

- **ie** – **[in]** pointer to the Time Correction IE structure

Returns

The time correction value in microseconds.

static inline void **ieee802154_header_ie_set_element_id**(struct ieee802154_header_ie *ie, uint8_t element_id)

Set the element ID of a header IE.

Parameters

- **ie** – **[in]** pointer to a header IE
- **element_id** – **[in]** IE element id in CPU byte order

static inline uint8_t **ieee802154_header_ie_get_element_id**(struct ieee802154_header_ie *ie)

Get the element ID of a header IE.

Parameters

- **ie** – **[in]** pointer to a header IE

Returns
header IE element id in CPU byte order

IEEE802154_HEADER_IE_HEADER_LENGTH

The header IE's header length (2 bytes).

IEEE802154_DEFINE_HEADER_IE_VENDOR_SPECIFIC(_vendor_oui, _vendor_specific_info, _vendor_specific_info_len)

Define a vendor specific header IE, see section 7.4.2.3.

Example usage (all parameters in little endian):

```
uint8_t vendor_specific_info[] = {...some vendor specific IE content...};
struct ieee802154_header_ie header_ie = IEEE802154_DEFINE_HEADER_IE_VENDOR_
↪SPECIFIC(
    {0x9b, 0xb8, 0xea}, vendor_specific_info, sizeof(vendor_specific_info));
```

Parameters

- **_vendor_oui** – an initializer for a 3 byte vendor oui array in little endian
- **_vendor_specific_info** – pointer to a variable length uint8_t array with the vendor specific IE content
- **_vendor_specific_info_len** – the length of the vendor specific IE content (in bytes)

IEEE802154_DEFINE_HEADER_IE_CSL_REDUCED(_csl_phase, _csl_period)

Define a reduced CSL IE, see section 7.4.2.3.

Example usage (all parameters in CPU byte order):

```
uint16_t csl_phase = ...;
uint16_t csl_period = ...;
struct ieee802154_header_ie header_ie =
    IEEE802154_DEFINE_HEADER_IE_CSL_REDUCED(csl_phase, csl_period);
```

Parameters

- **_csl_phase** – CSL phase in CPU byte order
- **_csl_period** – CSL period in CPU byte order

IEEE802154_DEFINE_HEADER_IE_CSL_FULL(_csl_phase, _csl_period, _csl_rendezvous_time)

Define a full CSL IE, see section 7.4.2.3.

Example usage (all parameters in CPU byte order):

```
uint16_t csl_phase = ...;
uint16_t csl_period = ...;
uint16_t csl_rendezvous_time = ...;
struct ieee802154_header_ie header_ie =
    IEEE802154_DEFINE_HEADER_IE_CSL_REDUCED(csl_phase, csl_period, csl_rendezvous_
↪time);
```

Parameters

- **_csl_phase** – CSL phase in CPU byte order
- **_csl_period** – CSL period in CPU byte order
- **_csl_rendezvous_time** – CSL rendezvous time in CPU byte order

IEEE802154_DEFINE_HEADER_IE_TIME_CORRECTION(_ack, _time_correction_us)

Define a Time Correction IE, see section 7.4.2.7.

Example usage (parameter in CPU byte order):

```
uint16_t time_sync_info = ...;
struct ieee802154_header_ie header_ie =
    IEEE802154_DEFINE_HEADER_IE_TIME_CORRECTION(true, time_sync_info);
```

Parameters

- **_ack** – whether or not the enhanced ACK frame that receives this IE is an ACK (true) or NACK (false)
- **_time_correction_us** – the positive or negative deviation from expected RX time in microseconds

IEEE802154_TIME_CORRECTION_HEADER_IE_LEN

The length in bytes of a “Time Correction” header IE.

IEEE802154_HEADER_TERMINATION_1_HEADER_IE_LEN

The length in bytes of a “Header Termination 1” header IE.

IEEE 802.15.4-2020, Section 10: General PHY requirements

enum ieee802154_phy_channel_page

PHY channel pages, see section 10.1.3.

A device driver must support the mandatory channel pages, frequency bands and channels of at least one IEEE 802.15.4 PHY.

Channel page and number assignments have developed over several versions of the standard and are not particularly well documented. Therefore some notes about peculiarities of channel pages and channel numbering:

- The 2006 version of the standard had a read-only phyChannelsSupported PHY PIB attribute that represented channel page/number combinations as a bitmap. This attribute was removed in later versions of the standard as the number of channels increased beyond what could be represented by a bit map. That’s the reason why it was decided to represent supported channels as a combination of channel pages and ranges instead.
- In the 2020 version of the standard, 13 channel pages are explicitly defined, but up to 32 pages could in principle be supported. This was a hard requirement in the 2006 standard. In later standards it is implicit from field specifications, e.g. the MAC PIB attribute macChannelPage (section 8.4.3.4, table 8-100) or channel page fields used in the SRM protocol (see section 8.2.26.5).
- ASK PHY (channel page one) was deprecated in the 2015 version of the standard. The 2020 version of the standard is a bit ambivalent whether channel page one disappeared as well or should be interpreted as O-QPSK now (see section 10.1.3.3). In Zephyr this ambivalence is resolved by deprecating channel page one.

- For some PHYs the standard doesn't clearly specify a channel page, namely the GFSK, RS-GFSK, CMB and TASK PHYs. These are all rather new and left out in our list as long as no driver wants to implement them.

Warning

The bit numbers are not arbitrary but represent the channel page numbers as defined by the standard. Therefore do not change the bit numbering.

Values:

enumerator `IEEE802154_ATTR_PHY_CHANNEL_PAGE_ZERO_OQPSK_2450_BPSK_868_915` = *BIT*(0)

Channel page zero supports the 2.4G channels of the O-QPSK PHY and all channels from the BPSK PHYs initially defined in the 2003 editions of the standard.

For channel page zero, 16 channels are available in the 2450 MHz band (channels 11-26, O-QPSK), 10 in the 915 MHz band (channels 1-10, BPSK), and 1 in the 868 MHz band (channel 0, BPSK).

You can retrieve the channels supported by a specific driver on this page via *IEEE802154_ATTR_PHY_SUPPORTED_CHANNEL_RANGES* attribute.

see section 10.1.3.3

enumerator `IEEE802154_ATTR_PHY_CHANNEL_PAGE_ONE_DEPRECATED` = *BIT*(1)

Formerly ASK PHY - deprecated in IEEE 802.15.4-2015.

enumerator `IEEE802154_ATTR_PHY_CHANNEL_PAGE_TWO_OQPSK_868_915` = *BIT*(2)

O-QPSK PHY - 868 MHz and 915 MHz bands, see section 10.1.3.3.

enumerator `IEEE802154_ATTR_PHY_CHANNEL_PAGE_THREE_CSS` = *BIT*(3)

CSS PHY - 2450 MHz band, see section 10.1.3.4.

enumerator `IEEE802154_ATTR_PHY_CHANNEL_PAGE_FOUR_HRP_UWB` = *BIT*(4)

UWB PHY - SubG, low and high bands, see section 10.1.3.5.

enumerator `IEEE802154_ATTR_PHY_CHANNEL_PAGE_FIVE_OQPSK_780` = *BIT*(5)

O-QPSK PHY - 780 MHz band, see section 10.1.3.2.

enumerator `IEEE802154_ATTR_PHY_CHANNEL_PAGE_SIX_RESERVED` = *BIT*(6)

reserved - not currently assigned

enumerator `IEEE802154_ATTR_PHY_CHANNEL_PAGE_SEVEN_MSK` = *BIT*(7)

MSK PHY - 780 MHz and 2450 MHz bands, see sections 10.1.3.6, 10.1.3.7.

enumerator `IEEE802154_ATTR_PHY_CHANNEL_PAGE_EIGHT_LRP_UWB` = *BIT*(8)

LRP UWB PHY, see sections 10.1.3.8.

enumerator `IEEE802154_ATTR_PHY_CHANNEL_PAGE_NINE_SUN_PREDEFINED` = *BIT*(9)

SUN FSK/OFDM/O-QPSK PHYs - predefined bands, operating modes and channels, see sections 10.1.3.9.

enumerator IEEE802154_ATTR_PHY_CHANNEL_PAGE_TEN_SUN_FSK_GENERIC = *BIT*(10)

SUN FSK/OFDM/O-QPSK PHYs - generic modulation and channel description, see sections 10.1.3.9, 7.4.4.11.

enumerator IEEE802154_ATTR_PHY_CHANNEL_PAGE_ELEVEN_OQPSK_2380 = *BIT*(11)

O-QPSK PHY - 2380 MHz band, see section 10.1.3.10.

enumerator IEEE802154_ATTR_PHY_CHANNEL_PAGE_TWELVE_LECIM = *BIT*(12)

LECIM DSSS/FSK PHYs, see section 10.1.3.11.

enumerator IEEE802154_ATTR_PHY_CHANNEL_PAGE_THIRTEEN_RCC = *BIT*(13)

RCC PHY, see section 10.1.3.12.

IEEE802154_DEFINE_PHY_SUPPORTED_CHANNELS(drv_attr, from, to)

Allocate memory for the supported channels driver attribute with a single channel range constant across all driver instances.

This is what most IEEE 802.15.4 drivers need.

Example usage:

```
IEEE802154_DEFINE_PHY_SUPPORTED_CHANNELS(drv_attr, 11, 26);
```

The attribute may then be referenced like this:

```
... &drv_attr.phy_supported_channels ...
```

See *ieee802154_attr_get_channel_page_and_range()* for a further shortcut that can be combined with this macro.

Parameters

- **drv_attr** – name of the local static variable to be declared for the local attributes structure
- **from** – the first channel to be supported
- **to** – the last channel to be supported

IEEE 802.15.4-2020, Section 15: HRP UWB PHY

For HRP UWB the symbol period is derived from the preamble symbol period (T_psym), see section 11.3, table 11-1 and section 15.2.5, table 15-4 (confirmed in IEEE 802.15.4z, section 15.1). Choosing among those periods cannot be done based on channel page and channel alone. The mean pulse repetition frequency must also be known, see the 'UwbPrf' parameter of the MCPS-DATA.request primitive (section 8.3.2, table 8-88) and the preamble parameters for HRP-ERDEV length 91 codes (IEEE 802.15.4z, section 15.2.6.2, table 15-7b).

enum ieee802154_phy_hrp_uwb_nominal_prf

represents the nominal pulse rate frequency of an HRP UWB PHY

Values:

enumerator IEEE802154_PHY_HRP_UWB_PRF_OFF = 0

standard modes, see section 8.3.2, table 8-88.

enumerator IEEE802154_PHY_HRP_UWB_NOMINAL_4_M = *BIT*(0)

enumerator IEEE802154_PHY_HRP_UWB_NOMINAL_16_M = *BIT*(1)

enumerator IEEE802154_PHY_HRP_UWB_NOMINAL_64_M = *BIT*(2)

enumerator IEEE802154_PHY_HRP_UWB_NOMINAL_64_M_BPRF = *BIT*(3)

enhanced ranging device (ERDEV) modes not specified in table 8-88, see IEEE 802.15.4z, section 15.1, section 15.2.6.2, table 15-7b, section 15.3.4.2 and section 15.3.4.3.

enumerator IEEE802154_PHY_HRP_UWB_NOMINAL_128_M_HPRF = *BIT*(4)

enumerator IEEE802154_PHY_HRP_UWB_NOMINAL_256_M_HPRF = *BIT*(5)

IEEE802154_PHY_HRP_UWB_PRF4_TPSYM_SYMBOL_PERIOD_NS

Nominal PRF 4MHz symbol period.

IEEE802154_PHY_HRP_UWB_PRF16_TPSYM_SYMBOL_PERIOD_NS

Nominal PRF 16MHz symbol period.

IEEE802154_PHY_HRP_UWB_PRF64_TPSYM_SYMBOL_PERIOD_NS

Nominal PRF 64MHz symbol period.

IEEE802154_PHY_HRP_UWB_ERDEV_TPSYM_SYMBOL_PERIOD_NS

ERDEV symbol period.

IEEE802154_PHY_HRP_UWB_RDEV

RDEV device mask.

IEEE802154_PHY_HRP_UWB_ERDEV

ERDEV device mask.

IEEE 802.15.4 Driver API

enum ieee802154_hw_caps

IEEE 802.15.4 driver capabilities.

Any driver properties that can be represented in binary form should be modeled as capabilities. These are called “hardware” capabilities for historical reasons but may also represent driver firmware capabilities (e.g. MAC offloading features).

Values:

enumerator IEEE802154_HW_ENERGY_SCAN = *BIT*(0)

Energy detection (ED) supported (optional)

enumerator IEEE802154_HW_FCS = *BIT*(1)

Frame checksum verification supported.

enumerator **IEEE802154_HW_FILTER** = *BIT*(2)

Filtering of PAN ID, extended and short address supported.

enumerator **IEEE802154_HW_PROMISC** = *BIT*(3)

Promiscuous mode supported.

enumerator **IEEE802154_HW_CSMA** = *BIT*(4)

CSMA-CA procedure supported on TX.

enumerator **IEEE802154_HW_TX_RX_ACK** = *BIT*(5)

Waits for ACK on TX if AR bit is set in TX pkt.

enumerator **IEEE802154_HW_RETRANSMISSION** = *BIT*(6)

Supports retransmission on TX ACK timeout.

enumerator **IEEE802154_HW_RX_TX_ACK** = *BIT*(7)

Sends ACK on RX if AR bit is set in RX pkt.

enumerator **IEEE802154_HW_TXTIME** = *BIT*(8)

TX at specified time supported.

enumerator **IEEE802154_HW_SLEEP_TO_TX** = *BIT*(9)

TX directly from sleep supported.

```
@note This HW capability does not conform to the requirements
specified in #61227 as it closely couples the driver to OpenThread's
capability and device model which is different from Zephyr's:
 - "Sleeping" is a well defined term in Zephyr related to internal
   power and thread management and different from "RX off" as
   defined in OT.
 - Currently all OT-capable drivers have the "sleep to TX"
   capability anyway plus we expect future drivers to implement it
   ootb as well, so no information is actually conveyed by this
   capability.
 - The `start()`/`stop()` API of a net device controls the
   interface's operational state. Drivers MUST respond with
   -ENETDOWN when calling `tx()` while their operational state is
   "DOWN", only devices in the "UP" state MAY transmit packets (RFC
   2863).
 - A migration path has been defined in #63670 for actual removal of
   this capability in favor of a standard compliant
   `configure(rx_on/rx_off)` call, see there for details.

@deprecated Drivers and L2 SHALL not introduce additional references
to this capability and remove existing ones as outlined in #63670.
```

enumerator **IEEE802154_HW_RXTIME** = *BIT*(10)

Timed RX window scheduling supported.

enumerator **IEEE802154_HW_TX_SEC** = *BIT*(11)

TX security supported (key management, encryption and authentication)

enumerator **IEEE802154_RX_ON_WHEN_IDLE** = *BIT*(12)

RxOnWhenIdle handling supported.

enum `ieee802154_filter_type`

Filter type, see *ieee802154_radio_api::filter*.

Values:

enumerator `IEEE802154_FILTER_TYPE_IEEE_ADDR`

Address type filter.

enumerator `IEEE802154_FILTER_TYPE_SHORT_ADDR`

Short address type filter.

enumerator `IEEE802154_FILTER_TYPE_PAN_ID`

PAN id type filter.

enumerator `IEEE802154_FILTER_TYPE_SRC_IEEE_ADDR`

Source address type filter.

enumerator `IEEE802154_FILTER_TYPE_SRC_SHORT_ADDR`

Source short address type filter.

enum `ieee802154_event`

Driver events, see *IEEE802154_CONFIG_EVENT_HANDLER*.

Values:

enumerator `IEEE802154_EVENT_TX_STARTED`

Data transmission started.

enumerator `IEEE802154_EVENT_RX_FAILED`

Data reception failed.

enumerator `IEEE802154_EVENT_RX_OFF`

An RX slot ended, requires *IEEE802154_HW_RXTIME*.

Note

This event SHALL not be triggered by drivers when RX is synchronously switched of due to a call to `stop()` or an RX slot being configured.

enum `ieee802154_rx_fail_reason`

RX failed event reasons, see *IEEE802154_EVENT_RX_FAILED*.

Values:

enumerator `IEEE802154_RX_FAIL_NOT_RECEIVED`

Nothing received.

enumerator `IEEE802154_RX_FAIL_INVALID_FCS`

Frame had invalid checksum.

enumerator `IEEE802154_RX_FAIL_ADDR_FILTERED`
Address did not match.

enumerator `IEEE802154_RX_FAIL_OTHER`
General reason.

enum `ieee802154_tx_mode`
IEEE 802.15.4 Transmission mode.

Values:

enumerator `IEEE802154_TX_MODE_DIRECT`
Transmit packet immediately, no CCA.

enumerator `IEEE802154_TX_MODE_CCA`
Perform CCA before packet transmission.

enumerator `IEEE802154_TX_MODE_CSMA_CA`
Perform full CSMA/CA procedure before packet transmission.

> **Note**
> requires IEEE802154_HW_CSMA capability.

enumerator `IEEE802154_TX_MODE_TXTIME`
Transmit packet in the future, at the specified time, no CCA.

> **Note**
> requires IEEE802154_HW_TXTIME capability.

enumerator `IEEE802154_TX_MODE_TXTIME_CCA`
Transmit packet in the future, perform CCA before transmission.

> **Note**
> requires IEEE802154_HW_TXTIME capability.

> **Note**
> Required for Thread 1.2 Coordinated Sampled Listening feature (see Thread specification 1.2.0, ch. 3.2.6.3).

enumerator `IEEE802154_TX_MODE_COMMON_COUNT`
Number of modes defined in ieee802154_tx_mode.

enumerator IEEE802154_TX_MODE_PRIV_START = *IEEE802154_TX_MODE_COMMON_COUNT*

This and higher values are specific to the protocol- or driver-specific extensions.

enum ieee802154_fpb_mode

IEEE 802.15.4 Frame Pending Bit table address matching mode.

Values:

enumerator IEEE802154_FPB_ADDR_MATCH_THREAD

The pending bit shall be set only for addresses found in the list.

enumerator IEEE802154_FPB_ADDR_MATCH_ZIGBEE

The pending bit shall be cleared for short addresses found in the list.

enum ieee802154_config_type

IEEE 802.15.4 driver configuration types.

Values:

enumerator IEEE802154_CONFIG_AUTO_ACK_FPB

Indicates how the driver should set the Frame Pending bit in ACK responses for Data Requests.

If enabled, the driver should determine whether to set the bit or not based on the information provided with *IEEE802154_CONFIG_ACK_FPB* config and FPB address matching mode specified. Otherwise, Frame Pending bit should be set to 1 (see section 6.7.3).

Note

requires *IEEE802154_HW_TX_RX_ACK* capability and is available in any interface operational state.

enumerator IEEE802154_CONFIG_ACK_FPB

Indicates whether to set ACK Frame Pending bit for specific address or not.

Disabling the Frame Pending bit with no address provided (NULL pointer) should disable it for all enabled addresses.

Note

requires *IEEE802154_HW_TX_RX_ACK* capability and is available in any interface operational state.

enumerator IEEE802154_CONFIG_PAN_COORDINATOR

Indicates whether the device is a PAN coordinator.

This influences packet filtering.

Note

Available in any interface operational state.

enumerator `IEEE802154_CONFIG_PROMISCUOUS`

Enable/disable promiscuous mode.

Note

Available in any interface operational state.

enumerator `IEEE802154_CONFIG_EVENT_HANDLER`

Specifies new IEEE 802.15.4 driver event handler.

Specifying NULL as a handler will disable events notification.

Note

Available in any interface operational state.

enumerator `IEEE802154_CONFIG_MAC_KEYS`

Updates MAC keys, key index and the per-key frame counter for drivers supporting transmit security offloading, see section 9.5, tables 9-9 and 9-10.

The key configuration SHALL NOT be accepted if the frame counter (in case frame counter per key is true) is not strictly larger than the current frame counter associated with the same key, see sections 8.2.2, 9.2.4 g/h) and 9.4.3.

Note

Requires *IEEE802154_HW_TX_SEC* capability and is available in any interface operational state.

enumerator `IEEE802154_CONFIG_FRAME_COUNTER`

Sets the current MAC frame counter value associated with the interface for drivers supporting transmit security offloading, see section 9.5, table 9-8, secFrameCounter.

Note

Requires *IEEE802154_HW_TX_SEC* capability and is available in any interface operational state.

Warning

The frame counter MUST NOT be accepted if it is not strictly greater than the current frame counter associated with the interface, see sections 8.2.2, 9.2.4 g/h) and 9.4.3. Otherwise the replay protection provided by the frame counter may

be compromised. Drivers SHALL return -EINVAL in case the configured frame counter does not conform to this requirement.

enumerator `IEEE802154_CONFIG_FRAME_COUNTER_IF_LARGER`

Sets the current MAC frame counter value if the provided value is greater than the current one.

Note

Requires *IEEE802154_HW_TX_SEC* capability and is available in any interface operational state.

Warning

This configuration option does not conform to the requirements specified in #61227 as it is redundant with *IEEE802154_CONFIG_FRAME_COUNTER*, and will therefore be deprecated in the future.

enumerator `IEEE802154_CONFIG_RX_SLOT`

Set or unset a radio reception window (RX slot).

This can be used for any scheduled reception, e.g.: Zigbee GP device, CSL, TSCH, etc.

The start and duration parameters of the RX slot are relative to the network subsystem's local clock. If the start parameter of the RX slot is -1 then any previously configured RX slot SHALL be canceled immediately. If the start parameter is any value in the past (including 0) or the duration parameter is zero then the receiver SHALL remain off forever until the RX slot has either been removed or re-configured to point to a future start time. If an RX slot is configured while the previous RX slot is still scheduled, then the previous slot SHALL be cancelled and the new slot scheduled instead.

RX slots MAY be programmed while the driver is "DOWN". If any past or future RX slot is configured when calling `start()` then the interface SHALL be placed in "UP" state but the receiver SHALL not be started.

The driver SHALL take care to start/stop the receiver autonomously, asynchronously and automatically around the RX slot. The driver SHALL resume power just before the RX slot and suspend it again after the slot unless another programmed event forces the driver not to suspend. The driver SHALL switch to the programmed channel before the RX slot and back to the channel set with set_channel() after the RX slot. If the driver interface is "DOWN" when the start time of an RX slot arrives, then the RX slot SHALL not be observed and the receiver SHALL remain off.

If the driver is "UP" while configuring an RX slot, the driver SHALL turn off the receiver immediately and (possibly asynchronously) put the driver into the lowest possible power saving mode until the start of the RX slot. If the driver is "UP" while the RX slot is deleted, then the driver SHALL enable the receiver immediately. The receiver MUST be ready to receive packets before returning from the `configure()` operation in this case.

This behavior means that setting an RX slot implicitly sets the MAC PIB attribute macRxOnWhenIdle (see section 8.4.3.1, table 8-94) to "false" while deleting the RX

slot implicitly sets macRxOnWhenIdle to “true”.

Note

requires *IEEE802154_HW_RXTIME* capability and is available in any interface operational state.

Note

Required for Thread 1.2 Coordinated Sampled Listening feature (see Thread specification 1.2.0, ch. 3.2.6.3).

enumerator `IEEE802154_CONFIG_CSL_PERIOD`

Enables or disables a device as a CSL receiver and configures its CSL period.

Configures the CSL period in units of 10 symbol periods. Values greater than zero enable CSL if the driver supports it and the device starts to operate as a CSL receiver. Setting this to zero disables CSL on the device. If the driver does not support CSL, the configuration call SHALL return -ENOTSUP.

See section 7.4.2.3 and section 8.4.3.6, table 8-104, macCslPeriod.

To offload CSL receiver timing to the driver the upper layer SHALL combine several configuration options in the following way:

i. Use *IEEE802154_CONFIG_ENH_ACK_HEADER_IE* once with an appropriate prefilled CSL IE and the CSL phase set to an arbitrary value or left uninitialized. The CSL phase SHALL be injected on-the-fly by the driver at runtime as outlined in 2. below. Adding a short and extended address will inform the driver of the specific CSL receiver to which it SHALL inject CSL IEs. If no addresses are given then the CSL IE will be injected into all enhanced ACK frames as soon as CSL is enabled. This configuration SHALL be done before enabling CSL by setting a CSL period greater than zero.

ii. Configure *IEEE802154_CONFIG_EXPECTED_RX_TIME* immediately followed by *IEEE802154_CONFIG_CSL_PERIOD*. To prevent race conditions, the upper layer SHALL ensure that the receiver is not enabled during or between the two calls (e.g. by a previously configured RX slot) nor SHALL a frame be transmitted concurrently.

    The expected RX time SHALL point to the end of SFD of an ideally timed RX frame in an arbitrary past or future CSL channel sample, i.e. whose “end of SFD” arrives exactly at the locally predicted time inside the CSL channel sample.

    The driver SHALL derive CSL anchor points and the CSL phase from the given expected RX time as follows:

```
cslAnchorPointNs = last expected RX time
                   + PHY-specific PHR duration in ns

startOfMhrNs = start of MHR of the frame containing the
               CSL IE relative to the local network clock

cslPhase = (startOfMhrNs - cslAnchorPointNs)
```

(continues on next page)

(continued from previous page)

```
           / (10 * PHY specific symbol period in ns)
           % cslPeriod
```

The driver SHALL set the CSL phase in the IE configured in 1. and inject that IE on-the-fly into outgoing enhanced ACK frames if the destination address conforms to the IE's address filter.

iii. Use *IEEE802154_CONFIG_RX_SLOT* periodically to schedule each CSL channel sample early enough before its start time. The size of the CSL channel sample SHALL take relative clock drift and scheduling uncertainties with respect to CSL transmitters into account as specified by the standard such that at least the full SHR of a legitimate RX frame is guaranteed to land inside the channel sample.

To this avail, the last configured expected RX time plus an integer number of CSL periods SHALL point to a fixed offset of the RX slot (not necessarily its center):

```
expectedRxTimeNs_N = last expected RX time
    + N * (cslPeriod * 10 * PHY-specific symbol period in ns)

expectedRxTimeNs_N - rxSlot_N.start == const for all N
```

While the configured CSL period is greater than zero, drivers SHOULD validate the offset of the expected RX time inside each RX slot accordingly. If the driver finds that the offset varies from slot to slot, drivers SHOULD log the difference but SHALL nevertheless accept and schedule the RX slot with a zero success value to work around minor implementation or rounding errors in upper layers.

Configure and start a CSL receiver:

```
ENH_ACK_HEADER_IE
   |
   | EXPECTED_RX_TIME (end of SFD of a perfectly timed RX frame
   |    |              in any past or future channel sample)
   |    |
   |    | CSL_PERIOD (>0)            RX_SLOT
   |    |    |                          |
   v    v    v                          v
-----------------------------------------------[-CSL channel sample ]----+
                                    ^                                     |
                                    |                                     |
                                    +---------------------- loop ---------+
```

Disable CSL on the receiver:

```
CSL_PERIOD (=0)
   |
   v
---------------------
```

Update the CSL period to a new value:

```
EXPECTED_RX_TIME (based on updated period)
   |
   |  CSL_PERIOD (>0, updated)       RX_SLOT
   |     |                             |
   v     v                             v
-----------------------------------------------[-CSL channel sample ]----+
                                    ^                                     |
                                    |                                     |
                                    +---------------------- loop ---------+
```

Note

Confusingly the standard calls the CSL receiver “CSL

coordinator” (i.e. “coordinating the CSL protocol timing”, see section 6.12.2.2), although, typically, a CSL coordinator is NOT also an IEEE 802.15.4 FFD coordinator or PAN coordintor but a simple RFD end device (compare the device roles outlined in sections 5.1, 5.3, 5.5 and 6.1). To avoid confusion we therefore prefer calling CSL coordinators (typically an RFD end device) “CSL receivers” and CSL peer devices (typically FFD coordinators or PAN coordinators) “CSL

transmitters”. Also note that at this time, we do NOT support unsynchronized transmission with CSL wake up frames as specified in section 6.12.2.4.4.

Note

Available in any interface operational state.

Note

Required for Thread 1.2 Coordinated Sampled Listening feature (see Thread specification 1.2.0, ch. 3.2.6.3).

enumerator `IEEE802154_CONFIG_EXPECTED_RX_TIME`

Configure a timepoint at which an RX frame is expected to arrive.

Configure the nanosecond resolution timepoint relative to the network subsystem’s local clock at which an RX frame’s end of SFD (i.e. equivalently its end of SHR, start of PHR, or in the case of PHYs with RDEV or ERDEV capability the RMARKER) is expected to arrive at the local antenna assuming perfectly synchronized local and remote network clocks and zero distance between antennas.

This parameter MAY be used to offload parts of timing sensitive TDMA (e.g. TSCH, beacon-enabled PAN including DSME), low-energy (e.g. CSL, RIT) or ranging (TDoA) protocols to the driver. In these protocols, medium access is tightly controlled such that the expected arrival time of a frame can be predicted within a well-defined time window. This feature will typically be combined with *IEEE802154_CONFIG_RX_SLOT* although this is not a hard requirement.

The “expected RX time” MAY be interpreted slightly differently depending on the protocol context:

- CSL phase (i.e. time to the next expected CSL transmission) or anchor time (i.e. any arbitrary timepoint with “zero CSL phase”) SHALL be derived by adding the PHY header duration to the expected RX time to calculate the “start of MHR” (“first symbol of MAC”, see section 6.12.2.1) required by the CSL protocol, compare *IEEE802154_CONFIG_CSL_PERIOD*.
- In TSCH the expected RX time MAY be set to macTsRxOffset + macTsRxWait / 2. Then the time correction SHALL be calculated as the expected RX time minus actual arrival timestamp, see section 6.5.4.3.
- In ranging applications, time difference of arrival (TDOA) MAY be calculated inside the driver comparing actual RMARKER timestamps against the assumed synchronized time at which the ranging frame was sent, see IEEE 802.15.4z.

In case of periodic protocols (e.g. CSL channel samples, periodic beacons of a single PAN, periodic ranging “blinks”), a single timestamp at any time in the past or in

the future may be given from which other expected timestamps can be derived by adding or subtracting multiples of the RX period. See e.g. the CSL documentation in this API.

Additionally this parameter MAY be used by drivers to discipline their local representation of a distributed network clock by deriving synchronization instants related to a remote representation of the same clock (as in PTP).

Note

Available in any interface operational state.

Note

Required for Thread 1.2 Coordinated Sampled Listening feature (see Thread specification 1.2.0, ch. 3.2.6.3).

enumerator `IEEE802154_CONFIG_ENH_ACK_HEADER_IE`

Adds a header information element (IE) to be injected into enhanced ACK frames generated by the driver if the given destination address filter matches.

Drivers implementing the *IEEE802154_HW_RX_TX_ACK* capability generate ACK frames autonomously. Setting this configuration will ask the driver to inject the given preconfigured header IE when generating enhanced ACK frames where appropriate by the standard. IEs for all other frame types SHALL be provided by L2.

The driver shall return -ENOTSUP in the following cases:
- It does not support the *IEEE802154_HW_RX_TX_ACK*,
- It does not support header IE injection,
- It cannot inject the runtime fields on-the-fly required for the given IE element ID (see list below).

Enhanced ACK header IEs (element IDs in parentheses) that either need to be rejected or explicitly supported and parsed by the driver because they require on-the-fly timing information injection are:
- CSL IE (0x1a)
- Rendezvous Time IE (0x1d)
- Time Correction IE (0x1e)

Drivers accepting this configuration option SHALL check the list of configured IEs for each outgoing enhanced ACK frame, select the ones appropriate for the received frame based on their element ID, inject any required runtime information on-the-fly and include the selected IEs into the enhanced ACK frame's MAC header.

Drivers supporting enhanced ACK header IE injection SHALL autonomously inject header termination IEs as required by the standard.

A destination short address and extended address MAY be given by L2 to filter the devices to which the given IE is included. Setting the short address to the broadcast address and the extended address to NULL will inject the given IE into all ACK frames unless a more specific filter is also present for any given destination device (fallback configuration). L2 SHALL take care to either set both address fields to valid device addresses or none.

This configuration type may be called several times with distinct element IDs and/or addresses. The driver SHALL either store all configured IE/address combinations or return -ENOMEM if no additional configuration can be stored.

Configuring a header IE with a previously configured element ID and address filter SHALL override the previous configuration. This implies that repetition of the same header IE/address combination is NOT supported.

Configuring an existing element ID/address filter combination with the header IE's length field set to zero SHALL remove that configuration. SHALL remove the fallback configuration if no address is given.

Configuring a header IE for an address filter with the header IE pointer set to NULL SHALL remove all header IE's for that address filter. SHALL remove ALL header IE configuration (including but not limited to fallbacks) if no address is given.

If any of the deleted configurations didn't previously exist, then the call SHALL be ignored. Whenever the length field is set to zero, the content fields MUST NOT be accessed by the driver.

L2 SHALL minimize the space required to keep IE configuration inside the driver by consolidating address filters and by removing configuration that is no longer required.

Note

requires *IEEE802154_HW_RX_TX_ACK* capability and is available in any interface operational state. Currently we only support header IEs but that may change in the future.

Note

Required for Thread 1.2 Coordinated Sampled Listening feature (see Thread specification 1.2.0, ch. 3.2.6.3).

Note

Required for Thread 1.2 Link Metrics feature (see Thread specification 1.2.0, ch. 4.11.3.3).

enumerator `IEEE802154_CONFIG_RX_ON_WHEN_IDLE`

Enable/disable RxOnWhenIdle MAC PIB attribute (Table 8-94).

Since there is no clear guidance in IEEE 802.15.4 specification about the definition of an "idle period", this implementation expects that drivers use the RxOnWhenIdle attribute to determine next radio state (false –> off, true –> receive) in the following scenarios:

- Finalization of a regular frame reception task, provided that:
  - The frame is received without errors and passes the filtering and it's not an spurious ACK.
  - ACK is not requested or transmission of ACK is not possible due to internal conditions.
- Finalization of a frame transmission or transmission of an ACK frame, when ACK is not requested in the transmitted frame.
- Finalization of the reception operation of a requested ACK due to:
  - ACK timeout expiration.
  - Reception of an invalid ACK or not an ACK frame.
  - Reception of the proper ACK, unless the transmitted frame was a Data Request Command and the frame pending bit on the received ACK is set to true.

In this case the radio platform implementation SHOULD keep the receiver on until a determined timeout which triggers an idle period start.
- Finalization of a stand alone CCA task.
- Finalization of a CCA operation with busy result during CSMA/CA procedure.
- Finalization of an Energy Detection task.
- Finalization of a scheduled radio reception window (see *IEEE802154_CONFIG_RX_SLOT*).

enumerator **IEEE802154_CONFIG_COMMON_COUNT**

Number of types defined in ieee802154_config_type.

enumerator **IEEE802154_CONFIG_PRIV_START** = *IEEE802154_CONFIG_COMMON_COUNT*

This and higher values are specific to the protocol- or driver-specific extensions.

enum **ieee802154_attr**

IEEE 802.15.4 driver attributes.

See *ieee802154_attr_value* and *ieee802154_radio_api* for usage details.

Values:

enumerator **IEEE802154_ATTR_PHY_SUPPORTED_CHANNEL_PAGES**

Retrieves a bit field with supported channel pages.

This attribute SHALL be implemented by all drivers.

enumerator **IEEE802154_ATTR_PHY_SUPPORTED_CHANNEL_RANGES**

Retrieves a pointer to the array of supported channel ranges within the currently configured channel page.

This attribute SHALL be implemented by all drivers.

enumerator **IEEE802154_ATTR_PHY_HRP_UWB_SUPPORTED_PRFS**

Retrieves a bit field with supported HRP UWB nominal pulse repetition frequencies.

This attribute SHALL be implemented by all devices that support channel page four (HRP UWB).

enumerator **IEEE802154_ATTR_COMMON_COUNT**

Number of attributes defined in ieee802154_attr.

enumerator **IEEE802154_ATTR_PRIV_START** = *IEEE802154_ATTR_COMMON_COUNT*

This and higher values are specific to the protocol- or driver-specific extensions.

typedef void (***energy_scan_done_cb_t**)(const struct *device* *dev, int16_t max_ed)

Energy scan callback.

typedef void (***ieee802154_event_cb_t**)(const struct *device* *dev, enum *ieee802154_event* evt, void *event_params)

Driver event callback.

static inline int **ieee802154_attr_get_channel_page_and_range**(enum *ieee802154_attr* attr, const enum *ieee802154_phy_channel_page* phy_supported_channel_page, const struct *ieee802154_phy_supported_channels* *phy_supported_channels, struct *ieee802154_attr_value* *value)

Helper function to handle channel page and range to be called from drivers' attr_get() implementation.

This only applies to drivers with a single channel page.

Parameters

- **attr** – The attribute to be retrieved.
- **phy_supported_channel_page** – The driver's unique channel page.
- **phy_supported_channels** – Pointer to the structure that contains the driver's channel range or ranges.
- **value** – The pointer to the value struct provided by the user.

Return values

- **0** – if the attribute could be resolved
- **-ENOENT** – if the attribute could not be resolved

IEEE802154_HW_CAPS_BITS_COMMON_COUNT

Number of bits used by ieee802154_hw_caps type.

IEEE802154_HW_CAPS_BITS_PRIV_START

This and higher values are specific to the protocol- or driver-specific extensions.

IEEE802154_CONFIG_RX_SLOT_NONE

Configuring an RX slot with the start parameter set to this value will cancel and delete any previously configured RX slot.

IEEE802154_CONFIG_RX_SLOT_OFF

Configuring an RX slot with this start parameter while the driver is "down", will keep RX off when the driver is being started.

Configuring an RX slot with this start value while the driver is "up" will immediately switch RX off until either the slot is deleted, see *IEEE802154_CONFIG_RX_SLOT_NONE* or a slot with a future start parameter is configured and that start time arrives.

IEEE 802.15.4 driver utils

static inline bool **ieee802154_is_ar_flag_set**(struct *net_buf* *frag)

Check if the AR flag is set on the frame inside the given *Network Packet Library*.

Parameters

- **frag** – A valid pointer on a *net_buf* structure, must not be NULL, and its length should be at least 1 byte (ImmAck frames are the shortest supported frames with 3 bytes excluding FCS).

Returns
true if AR flag is set, false otherwise

IEEE 802.15.4 driver callbacks

enum *net_verdict* **ieee802154_handle_ack**(struct *net_if* *iface, struct *net_pkt* *pkt)

IEEE 802.15.4 driver ACK handling callback into L2 that drivers must call when receiving an ACK package.

The IEEE 802.15.4 standard prescribes generic procedures for ACK handling on L2 (MAC) level. L2 stacks therefore have to provides a fast and re-usable generic implementation of this callback for drivers to call when receiving an ACK packet.

Note: This function is part of Zephyr's 802.15.4 stack driver -> L2 "inversion-of-control" adaptation API and must be implemented by all IEEE 802.15.4 L2 stacks.

Warning

Deviating from other functions in the net stack returning net_verdict, this function will not unref the package even if it returns NET_OK.

Parameters

- **iface** – A valid pointer on a network interface that received the packet
- **pkt** – A valid pointer on a packet to check

Returns
NET_OK if L2 handles the ACK package, NET_CONTINUE or NET_DROP otherwise.

void **ieee802154_init**(struct *net_if* *iface)

IEEE 802.15.4 driver initialization callback into L2 called by drivers to initialize the active L2 stack for a given interface.

Drivers must call this function as part of their own initialization routine.

Note: This function is part of Zephyr's 802.15.4 stack driver -> L2 "inversion-of-control" adaptation API and must be implemented by all IEEE 802.15.4 L2 stacks.

Parameters

- **iface** – A valid pointer on a network interface

IEEE 802.15.4-2020, Section 6: MAC functional description

IEEE802154_PHY_SYMBOLS_PER_SECOND(symbol_period_ns)

The symbol period (and therefore symbol rate) is defined in section 6.1: "Some of the timing parameters in definition of the MAC are in units of PHY symbols.

For PHYs that have multiple symbol periods, the duration to be used for the MAC parameters is defined in that PHY clause."

This is not necessarily the true physical symbol period, so take care to use this macro only when either the symbol period used for MAC timing is the same as the physical symbol period or if you actually mean the MAC timing symbol period.

PHY specific symbol periods are defined in PHY specific sections below.

IEEE 802.15.4-2020, Section 8: MAC services

IEEE802154_MAC_A_BASE_SLOT_DURATION

The number of PHY symbols forming a superframe slot when the superframe order is equal to zero, see sections 8.4.2, table 8-93, aBaseSlotDuration and section 6.2.1.

IEEE802154_MAC_A_NUM_SUPERFRAME_SLOTS

The number of slots contained in any superframe, see section 8.4.2, table 8-93, aNumSuperframeSlots.

IEEE802154_MAC_A_BASE_SUPERFRAME_DURATION

The number of PHY symbols forming a superframe when the superframe order is equal to zero, see section 8.4.2, table 8-93, aBaseSuperframeDuration.

IEEE802154_MAC_A_UNIT_BACKOFF_PERIOD(turnaround_time)

MAC PIB attribute aUnitBackoffPeriod, see section 8.4.2, table 8-93, in symbol periods, valid for all PHYs except SUN PHY in the 920 MHz band.

IEEE802154_MAC_RESPONSE_WAIT_TIME_DEFAULT

Default macResponseWaitTime in multiples of aBaseSuperframeDuration as defined in section 8.4.3.1, table 8-94.

IEEE 802.15.4-2020, Section 11: PHY services

IEEE802154_PHY_A_TURNAROUND_TIME_DEFAULT

Default PHY PIB attribute aTurnaroundTime, in PHY symbols, see section 11.3, table 11-1.

IEEE802154_PHY_A_TURNAROUND_TIME_1MS(symbol_period_ns)

PHY PIB attribute aTurnaroundTime for SUN, RS-GFSK, TVWS, and LECIM FSK PHY, in PHY symbols, see section 11.3, table 11-1.

IEEE802154_PHY_A_CCA_TIME

PHY PIB attribute aCcaTime, in PHY symbols, all PHYs except for SUN O-QPSK, see section 11.3, table 11-1.

IEEE 802.15.4-2020, Section 12: O-QPSK PHY

IEEE802154_PHY_OQPSK_868MHZ_SYMBOL_PERIOD_NS

O-QPSK 868Mhz band symbol period, see section 12.3.3.

IEEE802154_PHY_OQPSK_780_TO_2450MHZ_SYMBOL_PERIOD_NS

O-QPSK 780MHz, 915MHz, 2380MHz and 2450MHz bands symbol period, see section 12.3.3.

IEEE 802.15.4-2020, Section 13: BPSK PHY

IEEE802154_PHY_BPSK_868MHZ_SYMBOL_PERIOD_NS
BPSK 868MHz band symbol period, see section 13.3.3.

IEEE802154_PHY_BPSK_915MHZ_SYMBOL_PERIOD_NS
BPSK 915MHz band symbol period, see section 13.3.3.

IEEE 802.15.4-2020, Section 19: SUN FSK PHY

IEEE802154_PHY_SUN_FSK_863MHZ_915MHZ_SYMBOL_PERIOD_NS
SUN FSK 863Mhz and 915MHz band symbol periods, see section 19.1, table 19-1.

IEEE802154_PHY_SUN_FSK_PHR_LEN
SUN FSK PHY header length, in bytes, see section 19.2.4.

struct ieee802154_header_ie_vendor_specific
#include <ieee802154_ie.h> Vendor Specific Header IE, see section 7.4.2.3.

Public Members

uint8_t vendor_oui[IEEE802154_VENDOR_SPECIFIC_IE_OUI_LEN]
Vendor OUI.

uint8_t *vendor_specific_info
Vendor specific information.

struct ieee802154_header_ie_csl_full
#include <ieee802154_ie.h> Full CSL IE, see section 7.4.2.3.

Public Members

uint16_t csl_phase
CSL phase.

uint16_t csl_period
CSL period.

uint16_t csl_rendezvous_time
Rendezvous time.

struct ieee802154_header_ie_csl_reduced
#include <ieee802154_ie.h> Reduced CSL IE, see section 7.4.2.3.

Public Members

uint16_t **csl_phase**
CSL phase.

uint16_t **csl_period**
CSL period.

struct **ieee802154_header_ie_csl**
#include <ieee802154_ie.h> Generic CSL IE, see section 7.4.2.3.

Public Members

struct *ieee802154_header_ie_csl_full* **full**
CSL full information.

struct *ieee802154_header_ie_csl_reduced* **reduced**
CSL reduced information.

struct **ieee802154_header_ie_rit**
#include <ieee802154_ie.h> RIT IE, see section 7.4.2.4.

Public Members

uint8_t **time_to_first_listen**
Time to First Listen.

uint8_t **number_of_repeat_listen**
Number of Repeat Listen.

uint16_t **repeat_listen_interval**
Repeat listen interval.

struct **ieee802154_header_ie_rendezvous_time_full**
#include <ieee802154_ie.h> Full Rendezvous Time IE, see section 7.4.2.6 (macCslInterval is nonzero).

Public Members

uint16_t **rendezvous_time**
Rendezvous time.

uint16_t **wakeup_interval**
Wakeup interval.

struct **ieee802154_header_ie_rendezvous_time_reduced**

#include <ieee802154_ie.h> Reduced Rendezvous Time IE, see section 7.4.2.6 (macCslInterval is zero).

Public Members

uint16_t **rendezvous_time**

Rendezvous time.

struct **ieee802154_header_ie_rendezvous_time**

#include <ieee802154_ie.h> Rendezvous Time IE, see section 7.4.2.6.

Public Members

struct *ieee802154_header_ie_rendezvous_time_full* **full**

Rendezvous time full information.

struct *ieee802154_header_ie_rendezvous_time_reduced* **reduced**

Rendezvous time reduced information.

struct **ieee802154_header_ie_time_correction**

#include <ieee802154_ie.h> Time Correction IE, see section 7.4.2.7.

Public Members

uint16_t **time_sync_info**

Time synchronization information.

struct **ieee802154_phy_channel_range**

#include <ieee802154_radio.h> Represents a supported channel range, see *ieee802154_phy_supported_channels*.

Public Members

uint16_t **from_channel**

From channel range.

uint16_t **to_channel**

To channel range.

struct **ieee802154_phy_supported_channels**

#include <ieee802154_radio.h> Represents a list channels supported by a driver for a given interface, see *IEEE802154_ATTR_PHY_SUPPORTED_CHANNEL_RANGES*.

Public Members

const struct *ieee802154_phy_channel_range* *const **ranges**

Pointer to an array of channel range structures.

> **Warning**
>
> The pointer must be valid and constant throughout the life of the interface.

const uint8_t **num_ranges**

The number of currently available channel ranges.

struct **ieee802154_filter**

#include <*ieee802154_radio.h*> Filter value, see *ieee802154_radio_api::filter*.

Public Members

uint8_t ***ieee_addr**

Extended address, in little endian.

uint16_t **short_addr**

Short address, in CPU byte order.

uint16_t **pan_id**

PAN ID, in CPU byte order.

struct **ieee802154_key**

#include <*ieee802154_radio.h*> Key configuration for transmit security offloading, see *IEEE802154_CONFIG_MAC_KEYS*.

Public Members

uint8_t ***key_value**

Key material.

uint32_t **key_frame_counter**

Initial value of frame counter associated with the key, see section 9.4.3.

bool **frame_counter_per_key**

Indicates if per-key frame counter should be used, see section 9.4.3.

uint8_t **key_id_mode**

Key Identifier Mode, see section 9.4.2.3, Table 9-7.

uint8_t ***key_id**

Key Identifier, see section 9.4.4.

struct **ieee802154_config**

#include <ieee802154_radio.h> IEEE 802.15.4 driver configuration data.

Public Members

bool **enabled**

Is auto ACK FPB enabled.

Is enabled.

enum *ieee802154_fpb_mode* **mode**

Auto ACK FPB mode.

struct *ieee802154_config* **auto_ack_fpb**

see *IEEE802154_CONFIG_AUTO_ACK_FPB*

uint8_t ***addr**

little endian for both short and extended address

bool **extended**

Is extended address.

struct *ieee802154_config* **ack_fpb**

see *IEEE802154_CONFIG_ACK_FPB*

bool **pan_coordinator**

see *IEEE802154_CONFIG_PAN_COORDINATOR*

bool **promiscuous**

see *IEEE802154_CONFIG_PROMISCUOUS*

bool **rx_on_when_idle**

see *IEEE802154_CONFIG_RX_ON_WHEN_IDLE*

ieee802154_event_cb_t **event_handler**

see *IEEE802154_CONFIG_EVENT_HANDLER*

struct *ieee802154_key* ***mac_keys**

see *IEEE802154_CONFIG_MAC_KEYS*

Pointer to an array containing a list of keys used for MAC encryption. Refer to secKeyIdLookupDescriptor and secKeyDescriptor in IEEE 802.15.4

The key_value field points to a buffer containing the 16 byte key. The buffer SHALL be copied by the driver before returning from the call.

The variable length array is terminated by key_value field set to NULL.

uint32_t **frame_counter**

see *IEEE802154_CONFIG_FRAME_COUNTER*

net_time_t **start**

Nanosecond resolution timestamp relative to the network subsystem's local clock defining the start of the RX window during which the receiver is expected to be listening (i.e.

not including any driver startup times).

Configuring an rx_slot with the start attribute set to -1 will cancel and delete any previously active rx slot.

net_time_t **duration**

Nanosecond resolution duration of the RX window relative to the above RX window start time during which the receiver is expected to be listening (i.e.

not including any shutdown times). Only positive values larger than or equal zero are allowed.

Setting the duration to zero will disable the receiver, no matter what the start parameter.

uint8_t **channel**

Used channel.

struct *ieee802154_config* **rx_slot**

see *IEEE802154_CONFIG_RX_SLOT*

uint32_t **csl_period**

see *IEEE802154_CONFIG_CSL_PERIOD*

in CPU byte order

net_time_t **expected_rx_time**

see *IEEE802154_CONFIG_EXPECTED_RX_TIME*

struct ieee802154_header_ie ***header_ie**

Pointer to the header IE, see section 7.4.2.1, figure 7-21.

Certain header IEs may be incomplete if they require timing information to be injected at runtime on-the-fly, see the list in *IEEE802154_CONFIG_ENH_ACK_HEADER_IE*.

const uint8_t ***ext_addr**

Filters the devices that will receive this IE by extended address.

MAY be set to NULL to configure a fallback for all devices (implies that short_addr MUST also be set to *IEEE802154_BROADCAST_ADDRESS*).

in big endian

uint16_t **short_addr**

Filters the devices that will receive this IE by short address.

MAY be set to *IEEE802154_BROADCAST_ADDRESS* to configure a fallback for all devices (implies that ext_addr MUST also set to NULL in this case).

in CPU byte order

bool **purge_ie**

Flag for purging enh ACK header IEs.

When flag is set to true, driver should remove all existing header IEs, and all other entries in config should be ignored. This means that purging current header IEs and configuring a new one in the same call is not allowed.

struct *ieee802154_config* **ack_ie**

see *IEEE802154_CONFIG_ENH_ACK_HEADER_IE*

union ieee802154_config

Configuration data.

struct **ieee802154_attr_value**

#include <ieee802154_radio.h> IEEE 802.15.4 driver attribute values.

This structure is reserved to scalar and structured attributes that originate in the driver implementation and can neither be implemented as boolean *ieee802154_hw_caps* nor be derived directly or indirectly by the MAC (L2) layer. In particular this structure MUST NOT be used to return configuration data that originate from L2.

Note

To keep this union reasonably small, any attribute requiring a large memory area, SHALL be provided pointing to static memory allocated by the driver and valid throughout the lifetime of the driver instance.

Public Members

uint32_t **phy_supported_channel_pages**

A bit field that represents the supported channel pages, see *ieee802154_phy_channel_page*.

Note

To keep the API extensible as required by the standard, supported pages are modeled as a bitmap to support drivers that implement runtime switching between multiple channel pages.

Note

Currently none of the Zephyr drivers implements more than one channel page at runtime, therefore only one bit will be set and the current channel page (see the PHY PIB attribute phyCurrentPage, section 11.3, table 11-2) is considered to be read-only, fixed and “well known” via the supported channel pages attribute.

const struct *ieee802154_phy_supported_channels* ***phy_supported_channels**

Pointer to a structure representing channel ranges currently available on the selected channel page.

The selected channel page corresponds to the phyCurrentPage PHY PIB attribute, see the description of phy_supported_channel_pages above. Currently it can be retrieved via the *IEEE802154_ATTR_PHY_SUPPORTED_CHANNEL_PAGES* attribute.

Most drivers will expose a single channel page with a single, often zero-based, fixed channel range.

Some notable exceptions:

- The legacy channel page (zero) exposes ranges in different bands and even PHYs that are usually not implemented by a single driver.
- SUN and LECIM PHYs specify a large number of bands and operating modes on a single page with overlapping channel ranges each. Some of these ranges are not zero-based or contain "holes". This explains why several ranges may be necessary to represent all available channels.
- UWB PHYs often support partial channel ranges on the same channel page depending on the supported bands.

In these cases, drivers may expose custom configuration attributes (Kconfig, devicetree, runtime, ...) that allow switching between sub-ranges within the same channel page (e.g. switching between SubG and 2.4G bands on channel page zero or switching between multiple operating modes in the SUN or LECIM PHYs.

Warning

The pointer must be valid and constant throughout the life of the interface.

uint32_t **phy_hrp_uwb_supported_nominal_prfs**

A bit field representing supported HRP UWB pulse repetition frequencies (PRF), see enum ieee802154_phy_hrp_uwb_nominal_prf.

Note

Currently none of the Zephyr HRP UWB drivers implements more than one nominal PRF at runtime, therefore only one bit will be set and the current PRF (UwbPrf, MCPS-DATA.request, section 8.3.2, table 8-88) is considered to be read-only, fixed and "well known" via the supported PRF attribute.

struct **ieee802154_radio_api**

#include <ieee802154_radio.h> IEEE 802.15.4 driver interface API.

While L1-level driver features are exclusively implemented by drivers and MAY be mandatory to support certain application requirements, L2 features SHOULD be optional by default and only need to be implemented for performance optimization or precise timing as deemed necessary by driver maintainers. Fallback implementations ("Soft MAC") SHOULD be provided in the driver-independent L2 layer for all L2/MAC features especially if these features are not implemented in vendor hardware/firmware by a majority of existing in-tree drivers. If, however, a driver offers offloading opportunities then L2 implementations SHALL delegate performance critical or resource intensive tasks to the driver.

All drivers SHALL support two externally observable interface operational states: "UP" and "DOWN". Drivers MAY additionally support a "TESTING" interface state (see *continuous_carrier()*).

The following rules apply:

- An interface is considered "UP" when it is able to transmit and receive packets, "DOWN" otherwise (see precise definitions of the corresponding ifOperStatus values in RFC 2863, section 3.1.14, *net_if_oper_state* and the *continuous_carrier()* exception below). A device that has its receiver temporarily disabled during "UP" state due to an active receive window configuration is still considered "UP".
- Upper layers will assume that the interface managed by the driver is "UP" after a call to *start()* returned zero or -EALREADY. Upper layers assume that the interface is "DOWN" after calling *stop()* returned zero or -EALREADY.
- The driver SHALL block *start()*/*stop()* calls until the interface fully transitioned to the new state (e.g. the receiver is operational, ongoing transmissions were finished, etc.). Drivers SHOULD yield the calling thread (i.e. "sleep") if waiting for the new state without CPU interaction is possible.
- Drivers are responsible of guaranteeing atomicity of state changes. Appropriate means of synchronization SHALL be implemented (locking, atomic flags, ...).
- While the interface is "DOWN", the driver SHALL be placed in the lowest possible power state. The driver MAY return from a call to *stop()* before it reaches the lowest possible power state, i.e. manage power asynchronously. While the interface is "UP", the driver SHOULD autonomously and asynchronously transition to lower power states whenever possible. If the driver claims to support timed RX/TX capabilities and the upper layers configure an RX slot, then the driver SHALL immediately transition (asynchronously) to the lowest possible power state until the start of the RX slot or until a scheduled packet needs to be transmitted.
- The driver SHALL NOT change the interface's "UP"/"DOWN" state on its own. Initially, the interface SHALL be in the "DOWN" state.
- Drivers that implement the optional *continuous_carrier()* operation will be considered to be in the RFC 2863 "testing" ifOperStatus state if that operation returns zero. This state is active until either *start()* or *stop()* is called. If *continuous_carrier()* returns a non-zero value then the previous state is assumed by upper layers.
- If calls to *start()*/*stop()* return any other value than zero or -EALREADY, upper layers will consider the interface to be in a "lowerLayerDown" state as defined in RFC 2863.
- The RFC 2863 "dormant", "unknown" and "notPresent" ifOperStatus states are currently not supported. The "lowerLevelUp" state.
- The *ed_scan()*, *cca()* and *tx()* operations SHALL only be supported in the "UP" state and return -ENETDOWN in any other state. See the function-level API documentation below for further details.

Note

This structure is called "radio" API for backwards compatibility. A better name would be "IEEE 802.15.4 driver API" as typical drivers will not only implement L1/radio (PHY) features but also L2 (MAC) features if the vendor-specific driver hardware or firmware offers offloading opportunities.

Note

In case of devices that support timed RX/TX, the "UP" state is not equal to "receiver enabled". If a receive window (i.e. RX slot, see *IEEE802154_CONFIG_RX_SLOT*) is configured before calling *start()* then the receiver will not be enabled when transitioning to the "UP" state. Configuring a receive window while the interface is "UP"

will cause the receiver to be disabled immediately until the configured reception time has arrived.

Public Members

struct net_if_api **iface_api**

network interface API

> **Note**
>
> Network devices must extend the network interface API. It is therefore mandatory to place it at the top of the driver API struct so that it can be cast to a network interface.

enum *ieee802154_hw_caps* (***get_capabilities**)(const struct *device* *dev)

Get the device driver capabilities.

> **Note**
>
> Implementations SHALL be **isr-ok** and MUST NOT **sleep**. MAY be called in any interface state once the driver is fully initialized (“ready”).

Param dev
pointer to IEEE 802.15.4 driver device
Return
Bit field with all supported device driver capabilities.

int (***cca**)(const struct *device* *dev)

Clear Channel Assessment - Check channel’s activity.

> **Note**
>
> Implementations SHALL be **isr-ok** and MAY **sleep**. SHALL return -ENETDOWN unless the interface is “UP”.

Param dev
pointer to IEEE 802.15.4 driver device
Retval 0
the channel is available
Retval -EBUSY
The channel is busy.
Retval -EWOULDBLOCK
The operation is called from ISR context but temporarily cannot be executed without blocking.
Retval -ENETDOWN
The interface is not “UP”.
Retval -ENOTSUP
CCA is not supported by this driver.
Retval -EIO
The CCA procedure could not be executed.

int (*`set_channel`)(const struct *device* *dev, uint16_t channel)

Set current channel.

> **Note**
>
> Implementations SHALL be **isr-ok** and MAY **sleep**. SHALL return -EIO unless the interface is either “UP” or “DOWN”.

Param dev
pointer to IEEE 802.15.4 driver device
Param channel
the number of the channel to be set in CPU byte order
Retval 0
channel was successfully set
Retval -EALREADY
The previous channel is the same as the requested channel.
Retval -EINVAL
The given channel is not within the range of valid channels of the driver’s current channel page, see the IEEE802154_ATTR_PHY_SUPPORTED_CHANNEL_RANGES driver attribute.
Retval -EWOULDBLOCK
The operation is called from ISR context but temporarily cannot be executed without blocking.
Retval -ENOTSUP
The given channel is within the range of valid channels of the driver’s current channel page but unsupported by the current driver.
Retval -EIO
The channel could not be set.

int (*`filter`)(const struct *device* *dev, bool set, enum *ieee802154_filter_type* type, const struct *ieee802154_filter* *filter)

Set/Unset PAN ID, extended or short address filters.

> **Note**
>
> requires IEEE802154_HW_FILTER capability.

> **Note**
>
> Implementations SHALL be **isr-ok** and MAY **sleep**. SHALL return -EIO unless the interface is either “UP” or “DOWN”.

Param dev
pointer to IEEE 802.15.4 driver device
Param set
true to set the filter, false to remove it
Param type
the type of entity to be added/removed from the filter list (a PAN ID or a source/destination address)
Param filter
the entity to be added/removed from the filter list
Retval 0
The filter was successfully added/removed.

Retval -EINVAL
The given filter entity or filter entity type was not valid.
Retval -EWOULDBLOCK
The operation is called from ISR context but temporarily cannot be executed without blocking.
Retval -ENOTSUP
Setting/removing this filter or filter type is not supported by this driver.
Retval -EIO
Error while setting/removing the filter.

int (***set_txpower**)(const struct *device* *dev, int16_t dbm)

Set TX power level in dbm.

> **Note**
>
> Implementations SHALL be **isr-ok** and MAY **sleep**. SHALL return -EIO unless the interface is either “UP” or “DOWN”.

Param dev
pointer to IEEE 802.15.4 driver device
Param dbm
TX power in dbm
Retval 0
The TX power was successfully set.
Retval -EINVAL
The given dbm value is invalid or not supported by the driver.
Retval -EWOULDBLOCK
The operation is called from ISR context but temporarily cannot be executed without blocking.
Retval -EIO
The TX power could not be set.

int (***tx**)(const struct *device* *dev, enum *ieee802154_tx_mode* mode, struct *net_pkt* *pkt, struct *net_buf* *frag)

Transmit a packet fragment as a single frame.

Depending on the level of offloading features supported by the driver, the frame MAY not be fully encrypted/authenticated or it MAY not contain an FCS. It is the responsibility of L2 implementations to prepare the frame according to the offloading capabilities announced by the driver and to decide whether CCA, CSMA/CA, ACK or retransmission procedures need to be executed outside (“soft MAC”) or inside (“hard MAC”) the driver .

All frames originating from L2 SHALL have all required IEs pre-allocated and prefilled such that the driver does not have to parse and manipulate IEs at all. This includes ACK packets if the driver does not have the *IEEE802154_HW_RX_TX_ACK* capability. Also see *IEEE802154_CONFIG_ENH_ACK_HEADER_IE* for drivers that have the *IEEE802154_HW_RX_TX_ACK* capability.

IEs that cannot be prepared by L2 unless the TX time is known (e.g. CSL IE, Rendezvous Time IE, Time Correction IE, ...) SHALL be sent in any of the timed TX modes with appropriate timing information pre-filled in the IE such that drivers do not have to parse and manipulate IEs at all unless the frame is generated by the driver itself.

In case any of the timed TX modes is supported and used (see *ieee802154_hw_caps* and *ieee802154_tx_mode*), the driver SHALL take responsibility of scheduling and sending the packet at the precise programmed time autonomously without further interaction by upper layers. The call to *tx()* will block until the package has

either been sent successfully (possibly including channel acquisition and packet acknowledgment) or a terminal transmission error occurred. The driver SHALL sleep and keep power consumption to the lowest possible level until the scheduled transmission time arrives or during any other idle waiting time.

Note

Implementations MAY **sleep** and will usually NOT be **isr-ok** - especially when timed TX, CSMA/CA, retransmissions, auto-ACK or any other offloading feature is supported that implies considerable idle waiting time. SHALL return `-ENETDOWN` unless the interface is "UP".

Warning

The driver SHALL NOT take ownership of the given network packet and frame (fragment) buffer. Any data required by the driver including the actual frame content must be read synchronously and copied internally if needed at a later time (e.g. the contents of IEs required for protocol configuration, states of frame counters, sequence numbers, etc). Both, the packet and the buffer MAY be reused or released by upper layers immediately after the function returns.

Param dev
pointer to IEEE 802.15.4 driver device
Param mode
the transmission mode, some of which require specific offloading capabilities.
Param pkt
pointer to the network packet to be transmitted.
Param frag
pointer to a network buffer containing a single fragment with the frame data to be transmitted
Retval 0
The frame was successfully sent or scheduled. If the driver supports ACK offloading and the frame requested acknowledgment (AR bit set), this means that the packet was successfully acknowledged by its peer.
Retval -EINVAL
Invalid packet (e.g. an expected IE is missing or the encryption/authentication state is not as expected).
Retval -EBUSY
The frame could not be sent because the medium was busy (CSMA/CA or CCA offloading feature only).
Retval -ENOMSG
The frame was not confirmed by an ACK packet (TX ACK offloading feature only) or the received ACK packet was invalid.
Retval -ENOBUFS
The frame could not be scheduled due to missing internal resources (timed TX offloading feature only).
Retval -ENETDOWN
The interface is not "UP".
Retval -ENOTSUP
The given TX mode is not supported.
Retval -EIO
The frame could not be sent due to some unspecified driver error (e.g. the driver being busy).

int (***start**)(const struct *device* *dev)

Start the device.

Upper layers will assume the interface is "UP" if this operation returns with zero or `-EALREADY`. The interface is placed in receive mode before returning from this operation unless an RX slot has been configured (even if it lies in the past, see *IEEE802154_CONFIG_RX_SLOT*).

Note

Implementations SHALL be **isr-ok** and MAY **sleep**. MAY be called in any interface state once the driver is fully initialized ("ready").

Param dev
pointer to IEEE 802.15.4 driver device
Retval 0
The driver was successfully started.
Retval -EALREADY
The driver was already "UP".
Retval -EWOULDBLOCK
The operation is called from ISR context but temporarily cannot be executed without blocking.
Retval -EIO
The driver could not be started.

int (***stop**)(const struct *device* *dev)

Stop the device.

Upper layers will assume the interface is "DOWN" if this operation returns with zero or `-EALREADY`. The driver switches off the receiver before returning if it was previously on. The driver enters the lowest possible power mode after this operation is called. This MAY happen asynchronously (i.e. after the operation already returned control).

Note

Implementations SHALL be **isr-ok** and MAY **sleep**. MAY be called in any interface state once the driver is fully initialized ("ready").

Param dev
pointer to IEEE 802.15.4 driver device
Retval 0
The driver was successfully stopped.
Retval -EWOULDBLOCK
The operation is called from ISR context but temporarily cannot be executed without blocking.
Retval -EALREADY
The driver was already "DOWN".
Retval -EIO
The driver could not be stopped.

int (***continuous_carrier**)(const struct *device* *dev)

Start continuous carrier wave transmission.

The method blocks until the interface has started to emit a continuous carrier. To leave this mode, *start()* or *stop()* should be called, which will put the driver back into the "UP" or "DOWN" states, respectively.

> **Note**
>
> Implementations MAY **sleep** and will usually NOT be **isr-ok**. MAY be called in any interface state once the driver is fully initialized (“ready”).

Param dev
pointer to IEEE 802.15.4 driver device
Retval 0
continuous carrier wave transmission started
Retval -EALREADY
The driver was already in “TESTING” state and emitting a continuous carrier.
Retval -EIO
not started

int (*`configure`)(const struct *device* *dev, enum *ieee802154_config_type* type, const struct *ieee802154_config* *config)

Set or update driver configuration.

The method blocks until the interface has been reconfigured atomically with respect to ongoing package reception, transmission or any other ongoing driver operation.

> **Note**
>
> Implementations SHALL be **isr-ok** and MAY **sleep**. MAY be called in any interface state once the driver is fully initialized (“ready”). Some configuration options may not be supported in all interface operational states, see the detailed specifications in *ieee802154_config_type*. In this case the operation returns `-EACCES`.

Param dev
pointer to IEEE 802.15.4 driver device
Param type
the configuration type to be set
Param config
the configuration parameters to be set for the given configuration type
Retval 0
configuration successful
Retval -EINVAL
The configuration parameters are invalid for the given configuration type.
Retval -ENOTSUP
The given configuration type is not supported by this driver.
Retval -EACCES
The given configuration type is supported by this driver but cannot be configured in the current interface operational state.
Retval -ENOMEM
The configuration cannot be saved due to missing memory resources.
Retval -ENOENT
The resource referenced in the configuration parameters cannot be found in the configuration.
Retval -EWOULDBLOCK
The operation is called from ISR context but temporarily cannot be executed without blocking.
Retval -EIO
An internal error occurred while trying to configure the given configura-

tion parameter.

int (*ed_scan)(const struct *device* *dev, uint16_t duration, *energy_scan_done_cb_t* done_cb)

Run an energy detection scan.

> **Note**
>
> requires IEEE802154_HW_ENERGY_SCAN capability

> **Note**
>
> The radio channel must be set prior to calling this function.

> **Note**
>
> Implementations SHALL be **isr-ok** and MAY **sleep**. SHALL return `-ENETDOWN` unless the interface is "UP".

Param dev
pointer to IEEE 802.15.4 driver device
Param duration
duration of energy scan in ms
Param done_cb
function called when the energy scan has finished
Retval 0
the energy detection scan was successfully scheduled
Retval -EBUSY
the energy detection scan could not be scheduled at this time
Retval -EALREADY
a previous energy detection scan has not finished yet.
Retval -ENETDOWN
The interface is not "UP".
Retval -ENOTSUP
This driver does not support energy scans.
Retval -EIO
The energy detection procedure could not be executed.

net_time_t (*get_time)(const struct *device* *dev)

Get the current time in nanoseconds relative to the network subsystem's local uptime clock as represented by this network interface.

See *net_time_t* for semantic details.

> **Note**
>
> requires IEEE802154_HW_TXTIME and/or IEEE802154_HW_RXTIME capabilities. Implementations SHALL be **isr-ok** and MUST NOT **sleep**. MAY be called in any interface state once the driver is fully initialized ("ready").

Param dev
pointer to IEEE 802.15.4 driver device

Return
nanoseconds relative to the network subsystem's local clock, -1 if an error occurred or the operation is not supported

uint8_t (*`get_sch_acc`)(const struct *device* *dev)

Get the current estimated worst case accuracy (maximum ± deviation from the nominal frequency) of the network subsystem's local clock used to calculate tolerances and guard times when scheduling delayed receive or transmit radio operations.

The deviation is given in units of PPM (parts per million).

Note

requires IEEE802154_HW_TXTIME and/or IEEE802154_HW_RXTIME capabilities.

Note

Implementations may estimate this value based on current operating conditions (e.g. temperature). Implementations SHALL be **isr-ok** and MUST NOT **sleep**. MAY be called in any interface state once the driver is fully initialized ("ready").

Param dev
pointer to IEEE 802.15.4 driver device
Return
current estimated clock accuracy in PPM

int (*`attr_get`)(const struct *device* *dev, enum *ieee802154_attr* attr, struct *ieee802154_attr_value* *value)

Get the value of a driver specific attribute.

Note

This function SHALL NOT return any values configurable by the MAC (L2) layer. It is reserved to non-boolean (i.e. scalar or structured) attributes that originate from the driver implementation and cannot be directly or indirectly derived by L2. Boolean attributes SHALL be implemented as *ieee802154_hw_caps*.

Note

Implementations SHALL be **isr-ok** and MUST NOT **sleep**. MAY be called in any interface state once the driver is fully initialized ("ready").

Retval 0
The requested attribute is supported by the driver and the value can be retrieved from the corresponding *ieee802154_attr_value* member.
Retval -ENOENT
The driver does not provide the requested attribute. The value structure has not been updated with attribute data. The content of the value attribute is undefined.

IEEE 802.15.4 L2 / Native Stack API This documents the IEEE 802.15.4 L2 native stack, which neither applications nor drivers will ever access directly. It is called internally by Zephyr's upper network layers (L3+), its socket and network context abstractions. This API is therefore of interest to IEEE 802.15.4 **subsystem contributors** only.

group `ieee802154_l2`

IEEE 802.15.4 L2 APIs.

Since
1.0

Version
0.8.0

This API provides integration with Zephyr's sockets and network contexts. **Application and driver developers should never interface directly with this API.** It is of interest to subsystem maintainers only.

The API implements and extends the following structures:

- implements Zephyr's internal L2-level socket and network context abstractions (context/socket operations, see *Network L2 Abstraction Layer*),
- protocol-specific extension to the interface structure (see *Network Interface abstraction layer*)
- protocol-specific extensions to the network packet structure (see *Network Packet Library*),

> **Note**
>
> All section, table and figure references are to the IEEE 802.15.4-2020 standard.

Defines

`IEEE802154_MAX_PHY_PACKET_SIZE`

Represents the PHY constant aMaxPhyPacketSize, see section 11.3.

> **Note**
>
> Currently only 127 byte sized packets are supported although some PHYs (e.g. SUN, MSK, LECIM, ...) support larger packet sizes. Needs to be changed once those PHYs should be fully supported.

`IEEE802154_FCS_LENGTH`

Represents the frame check sequence length, see section 7.2.1.1.

> **Note**
>
> Currently only a 2 byte FCS is supported although some PHYs (e.g. SUN, TVWS, ...) optionally support a 4 byte FCS. Needs to be changed once those PHYs should be fully supported.

`IEEE802154_MTU`

IEEE 802.15.4 “hardware” MTU (not to be confused with L3/IP MTU), i.e.

the actual payload available to the next higher layer.

This is equivalent to the IEEE 802.15.4 MAC frame length minus checksum bytes which is again equivalent to the PHY payload aka PSDU length minus checksum bytes. This definition exists for compatibility with the same concept in Linux and Zephyr’s L3. It is not a concept from the IEEE 802.15.4 standard.

> **Note**
>
> Currently only the original frame size from the 2006 standard version and earlier is supported. The 2015+ standard introduced PHYs with larger PHY payload. These are not (yet) supported in Zephyr.

`IEEE802154_SHORT_ADDR_LENGTH`

IEEE 802.15.4 short address length.

`IEEE802154_EXT_ADDR_LENGTH`

IEEE 802.15.4 extended address length.

`IEEE802154_MAX_ADDR_LENGTH`

IEEE 802.15.4 maximum address length.

`IEEE802154_NO_CHANNEL`

A special channel value that symbolizes “all” channels or “any” channel - depending on context.

`IEEE802154_BROADCAST_ADDRESS`

Represents the IEEE 802.15.4 broadcast short address, see sections 6.1 and 8.4.3, table 8-94, macShortAddress.

`IEEE802154_NO_SHORT_ADDRESS_ASSIGNED`

Represents a special IEEE 802.15.4 short address that indicates that a device has been associated with a coordinator but did not receive a short address, see sections 6.4.1 and 8.4.3, table 8-94, macShortAddress.

`IEEE802154_BROADCAST_PAN_ID`

Represents the IEEE 802.15.4 broadcast PAN ID, see section 6.1.

`IEEE802154_SHORT_ADDRESS_NOT_ASSOCIATED`

Represents a special value of the macShortAddress MAC PIB attribute, while the device is not associated, see section 8.4.3, table 8-94.

`IEEE802154_PAN_ID_NOT_ASSOCIATED`

Represents a special value of the macPanId MAC PIB attribute, while the device is not associated, see section 8.4.3, table 8-94.

Enums

enum `ieee802154_device_role`

IEEE 802.15.4 device role.

Values:

enumerator `IEEE802154_DEVICE_ROLE_ENDDEVICE`

End device.

enumerator `IEEE802154_DEVICE_ROLE_COORDINATOR`

Coordinator.

enumerator `IEEE802154_DEVICE_ROLE_PAN_COORDINATOR`

PAN coordinator.

struct `ieee802154_security_ctx`

#include <ieee802154.h> Interface-level security attributes, see section 9.5.

Public Members

uint32_t `frame_counter`

Interface-level outgoing frame counter, section 9.5, table 9-8, secFrameCounter.

Only used when the driver does not implement key-specific frame counters.

uint8_t `key`[16]

Interface-level frame encryption security key material.

Currently native L2 only supports a single secKeySource, see section 9.5, table 9-9, in combination with secKeyMode zero (implicit key mode), see section 9.4.2.3, table 9-7.

Warning

This is no longer in accordance with the 2015+ versions of the standard and needs to be extended in the future for full security procedure compliance.

uint8_t `key_len`

Length in bytes of the interface-level security key material.

uint8_t `level`

Frame security level, possible values are defined in section 9.4.2.2, table 9-6.

Warning

Currently native L2 allows to configure one common security level for all frame types, commands and information elements. This is no longer in accordance with the 2015+ versions of the standard and needs to be extended in the future for full security procedure compliance.

uint8_t **key_mode**

Frame security key mode.

Currently only implicit key mode is partially supported, see section 9.4.2.3, table 9-7, secKeyMode.

Warning

This is no longer in accordance with the 2015+ versions of the standard and needs to be extended in the future for full security procedure compliance.

struct **ieee802154_context**

#include <*ieee802154.h*> IEEE 802.15.4 L2 context.

Public Members

uint16_t **pan_id**

PAN ID.

The identifier of the PAN on which the device is operating. If this value is 0xffff, the device is not associated. See section 8.4.3.1, table 8-94, macPanId.

in CPU byte order

uint16_t **channel**

Channel Number.

The RF channel to use for all transmissions and receptions, see section 11.3, table 11-2, phyCurrentChannel. The allowable range of values is PHY dependent as defined in section 10.1.3.

in CPU byte order

uint16_t **short_addr**

Short Address (in CPU byte order)

Range:
- 0x0000–0xfffd: associated, short address was assigned
- 0xfffe: associated but no short address assigned
- 0xffff: not associated (default),

See section 6.4.1, table 6-4 (Usage of the shart address) and section 8.4.3.1, table 8-94, macShortAddress.

uint8_t **ext_addr**[8]

Extended Address (in little endian)

The extended address is device specific, usually permanently stored on the device and immutable.

See section 8.4.3.1, table 8-94, macExtendedAddress.

struct *net_linkaddr_storage* **linkaddr**

Link layer address (in big endian)

struct *ieee802154_security_ctx* **sec_ctx**

Security context.

struct *ieee802154_req_params* ***scan_ctx**

Pointer to scanning parameters and results, guarded by scan_ctx_lock.

struct k_sem **scan_ctx_lock**

Used to maintain integrity of data for all fields in this struct unless otherwise documented on field level.

uint8_t **coord_ext_addr**[8]

Coordinator extended address.

see section 8.4.3.1, table 8-94, macCoordExtendedAddress, the address of the coordinator through which the device is associated.

A value of zero indicates that a coordinator extended address is unknown (default).

in little endian

uint16_t **coord_short_addr**

Coordinator short address.

see section 8.4.3.1, table 8-94, macCoordShortAddress, the short address assigned to the coordinator through which the device is associated.

A value of 0xfffe indicates that the coordinator is only using its extended address. A value of 0xffff indicates that this value is unknown.

in CPU byte order

int16_t **tx_power**

Transmission power in dBm.

enum *net_l2_flags* **flags**

L2 flags.

uint8_t **sequence**

Data sequence number.

The sequence number added to the transmitted Data frame or MAC command, see section 8.4.3.1, table 8-94, macDsn.

uint8_t **device_role**

Device Role.

See section 6.1: A device may be operating as end device (0), coordinator (1), or PAN coordinator (2). If no device role is explicitly configured then the device will be treated as an end device.

A value of 3 is undefined.

Can be read/set via *ieee802154_device_role*.

uint8_t **ack_requested**

ACK requested flag, guarded by ack_lock.

uint8_t **ack_seq**
: ACK expected sequence number, guarded by ack_lock.

struct k_sem **ack_lock**
: ACK lock, guards ack_* fields.

struct k_sem **ctx_lock**
: Context lock.

  This lock guards all mutable context attributes unless otherwise mentioned on attribute level.

OpenThread L2 Adaptation Layer API Zephyr's OpenThread L2 platform adaptation layer glues the external OpenThread stack together with Zephyr's IEEE 802.15.4 protocol agnostic driver API. This API is of interest to OpenThread L2 **subsystem contributors** only.

The OpenThread API is part of the *Thread protocol* subsystem and documented there.

Thread protocol

- *Overview*
- *Internet connectivity*
- *Sample usage*
- *Thread related APIs*
  - *OpenThread Driver API*
  - *OpenThread L2 Adaptation Layer API*

Overview Thread is a low-power mesh networking technology, designed specifically for home automation applications. It is an IPv6-based standard, which uses 6LoWPAN technology over IEEE 802.15.4 protocol. IP connectivity lets you easily connect a Thread mesh network to the internet with a Thread Border Router.

The Thread specification provides a high level of network security. Mesh networks built with Thread are secure - only authenticated devices can join the network and all communications within the mesh are encrypted. More information about Thread protocol can be found at Thread Group website.

Zephyr integrates an open source Thread protocol implementation called OpenThread, documented on the OpenThread website.

Internet connectivity A Thread Border Router is required to connect mesh network to the internet. An open source implementation of Thread Border Router is provided by the OpenThread community. See OpenThread Border Router guide for instructions on how to set up a Border Router.

Sample usage You can try using OpenThread with the Zephyr Echo server and Echo client samples, which provide out-of-the-box configuration for OpenThread. To enable OpenThread support in these samples, build them with `overlay-ot.conf` overlay config file. See sockets-echo-server and sockets-echo-client samples for details.

Thread related APIs

OpenThread Driver API OpenThread L2 uses Zephyr's protocol agnostic IEEE 802.15.4 driver API internally. This API is of interest to **driver developers** that want to support OpenThread.

The driver API is part of the *IEEE 802.15.4 Driver API* subsystem and documented there.

OpenThread L2 Adaptation Layer API Zephyr's OpenThread L2 platform adaptation layer glues the external OpenThread stack together with Zephyr's IEEE 802.15.4 protocol agnostic driver API. This API is of interest to OpenThread L2 **subsystem contributors** only.

Related code samples

OpenThread co-processor
Build a Thread border-router using OpenThread's co-processor designs.

group `openthread`

OpenThread Layer 2 abstraction layer.

Functions

int `openthread_state_changed_cb_register`(struct openthread_context *ot_context, struct *openthread_state_changed_cb* *cb)

Registers callbacks which will be called when certain configuration or state changes occur within OpenThread.

Parameters

- `ot_context` – the OpenThread context to register the callback with.
- `cb` – callback struct to register.

int `openthread_state_changed_cb_unregister`(struct openthread_context *ot_context, struct *openthread_state_changed_cb* *cb)

Unregisters OpenThread configuration or state changed callbacks.

Parameters

- `ot_context` – the OpenThread context to unregister the callback from.
- `cb` – callback struct to unregister.

k_tid_t `openthread_thread_id_get`(void)

Get OpenThread thread identification.

struct openthread_context *`openthread_get_default_context`(void)

Get pointer to default OpenThread context.

Return values

- `!NULL` – On success.
- `NULL` – On failure.

struct otInstance *`openthread_get_default_instance`(void)

Get pointer to default OpenThread instance.

Return values

- `!NULL` – On success.

- `NULL` – On failure.

int `openthread_start`(struct openthread_context *ot_context)

Starts the OpenThread network.

Depends on active settings: it uses stored network configuration, start joining procedure or uses default network configuration. Additionally when the device is MTD, it sets the SED mode to properly attach the network.

Parameters

- `ot_context`

void `openthread_api_mutex_lock`(struct openthread_context *ot_context)

Lock internal mutex before accessing OT API.

OpenThread API is not thread-safe, therefore before accessing any API function, it's needed to lock the internal mutex, to prevent the OpenThread thread from preempting the API call.

Parameters

- `ot_context` – Context to lock.

int `openthread_api_mutex_try_lock`(struct openthread_context *ot_context)

Try to lock internal mutex before accessing OT API.

This function behaves like *openthread_api_mutex_lock()* provided that the internal mutex is unlocked. Otherwise, it exists immediately and returns a negative value.

Parameters

- `ot_context` – Context to lock.

Return values

- `0` – On success.
- `<0` – On failure.

void `openthread_api_mutex_unlock`(struct openthread_context *ot_context)

Unlock internal mutex after accessing OT API.

Parameters

- `ot_context` – Context to unlock.

struct `openthread_state_changed_cb`

#include <*openthread.h*> OpenThread state change callback

OpenThread state change callback structure

Used to register a callback in the callback list. As many callbacks as needed can be added as long as each of them are unique pointers of struct *openthread_state_changed_cb*. Beware such structure should not be allocated on stack.

Public Members

void (*`state_changed_cb`)(otChangedFlags flags, struct openthread_context *ot_context, void *user_data)

Callback for notifying configuration or state changes.

Param flags
as per OpenThread otStateChangedCallback() aFlags parameter. See https://openthread.io/reference/group/api-instance#otstatechangedcallback
Param ot_context
the OpenThread context the callback is registered with.
Param user_data
Data to pass to the callback.

void *`user_data`

User data if required.

sys_snode_t `node`

Internally used field for list handling.

- user must not directly modify

Point-to-Point Protocol (PPP) Support

- *Overview*
- *Testing*

Overview Point-to-Point Protocol (PPP) is a data link layer (layer 2) communications protocol used to establish a direct connection between two nodes. PPP is used over many types of serial links since IP packets cannot be transmitted over a modem line on their own, without some data link protocol.

In Zephyr, each individual PPP link is modelled as a network interface. This is similar to how Linux implements PPP.

PPP support must be enabled at compile time by setting option `CONFIG_NET_L2_PPP`. The PPP implementation supports only these protocols:

- LCP (Link Control Protocol, RFC1661)
- HDLC (High-level data link control, RFC1662)
- IPCP (IP Control Protocol, RFC1332)
- IPV6CP (IPv6 Control Protocol, RFC5072)

For using PPP with a cellular modem, see cellular-modem sample for additional information.

Testing See the net-tools README file for more details on how to test the Zephyr PPP against pppd running in Linux.

Wi-Fi Management

Overview The Wi-Fi management API is used to manage Wi-Fi networks. It supports below modes:

- IEEE802.11 Station (STA)

- IEEE802.11 Access Point (AP)

Only personal mode security is supported with below types:

- Open
- WPA2-PSK
- WPA3-PSK-256
- WPA3-SAE

The Wi-Fi management API is implemented in the *wifi_mgmt* module as a part of the networking L2 stack. Currently, two types of Wi-Fi drivers are supported:

- Networking or socket offloaded drivers
- Native L2 Ethernet drivers

API Reference

group `wifi_mgmt`

Wi-Fi Management API.

Wi-Fi utility functions.

Utility functions for the Wi-Fi subsystem.

int `wifi_utils_parse_scan_bands`(char *scan_bands_str, uint8_t *band_map)

Convert a band specification string to a bitmap representing the bands.

The function will parse a string which specifies Wi-Fi frequency band values as a comma separated string and convert it to a bitmap. The string can use the following characters to represent the bands:

- 2: 2.4 GHz
- 5: 5 GHz
- 6: 6 GHz

For the bitmap generated refer to *wifi_frequency_bands* for bit position of each band.

E.g. a string “2,5,6” will be converted to a bitmap value of 0x7

Parameters

- `scan_bands_str` – String which spe.
- `band_map` – Pointer to the bitmap variable to be updated.

Return values

- `0` – on success.
- `-errno` – value in case of failure.

int `wifi_utils_parse_scan_ssids`(char *scan_ssids_str, const char *ssids[], uint8_t num_ssids)

Append a string containing an SSID to an array of SSID strings.

Parameters

- `scan_ssids_str` – string to be appended in the list of scanned SSIDs.
- `ssids` – Pointer to an array where the SSIDs pointers are to be stored.

- `num_ssids` – Maximum number of SSIDs that can be stored.

Return values

- `0` – on success.
- `-errno` – value in case of failure.

int `wifi_utils_parse_scan_chan`(char *scan_chan_str, struct *wifi_band_channel* *chan, uint8_t max_channels)

Convert a string containing a specification of scan channels to an array.

The function will parse a string which specifies channels to be scanned as a string and convert it to an array.

The channel string has to be formatted using the colon (:), comma(,), hyphen (-) and underscore (_) delimiters as follows:

- A colon identifies the value preceding it as a band. A band value (2: 2.4 GHz, 5: 5 GHz 6: 6 GHz) has to precede the channels in that band (e.g. 2: etc)
- Hyphens (-) are used to identify channel ranges (e.g. 2-7, 32-48 etc)
- Commas are used to separate channel values within a band. Channels can be specified as individual values (2,6,48 etc) or channel ranges using hyphens (1-14, 32-48 etc)
- Underscores (_) are used to specify multiple band-channel sets (e.g. 2:1,2_5:36,40 etc)
- No spaces should be used anywhere, i.e. before/after commas, before/after hyphens etc.

An example channel specification specifying channels in the 2.4 GHz and 5 GHz bands is as below: 2:1,5,7,9-11_5:36-48,100,163-167

Parameters

- `scan_chan_str` – List of channels expressed in the format described above.
- `chan` – Pointer to an array where the parsed channels are to be stored.
- `max_channels` – Maximum number of channels to store

Return values

- `0` – on success.
- `-errno` – value in case of failure.

bool `wifi_utils_validate_chan`(uint8_t band, uint16_t chan)

Validate a channel against a band.

Parameters

- `band` – Band to validate the channel against.
- `chan` – Channel to validate.

Return values

- `true` – if the channel is valid for the band.
- `false` – if the channel is not valid for the band.

bool `wifi_utils_validate_chan_2g`(uint16_t chan)

Validate a channel against the 2.4 GHz band.

Parameters

- `chan` – Channel to validate.

Return values

- `true` – if the channel is valid for the band.
- `false` – if the channel is not valid for the band.

bool `wifi_utils_validate_chan_5g`(uint16_t chan)

Validate a channel against the 5 GHz band.

Parameters

- `chan` – Channel to validate.

Return values

- `true` – if the channel is valid for the band.
- `false` – if the channel is not valid for the band.

bool `wifi_utils_validate_chan_6g`(uint16_t chan)

Validate a channel against the 6 GHz band.

Parameters

- `chan` – Channel to validate.

Return values

- `true` – if the channel is valid for the band.
- `false` – if the channel is not valid for the band.

`WIFI_UTILS_MAX_BAND_STR_LEN`

Maximum length of the band specification string.

`WIFI_UTILS_MAX_CHAN_STR_LEN`

Maximum length of the channel specification string.

Defines

`WIFI_COUNTRY_CODE_LEN`

Length of the country code string.

`WIFI_SSID_MAX_LEN`

Max SSID length.

`WIFI_PSK_MIN_LEN`

Minimum PSK length.

`WIFI_PSK_MAX_LEN`

Maximum PSK length.

`WIFI_SAE_PSWD_MAX_LEN`

Max SAW password length.

`WIFI_MAC_ADDR_LEN`

MAC address length.

`WIFI_CHANNEL_MIN`
: Minimum channel number.

`WIFI_CHANNEL_MAX`
: Maximum channel number.

`WIFI_CHANNEL_ANY`
: Any channel number.

`WIFI_INTERFACE_INDEX_MIN`
: Network interface index min value.

`WIFI_INTERFACE_INDEX_MAX`
: Network interface index max value.

`NET_REQUEST_WIFI_SCAN`
: Request a Wi-Fi scan.

`NET_REQUEST_WIFI_CONNECT`
: Request a Wi-Fi connect.

`NET_REQUEST_WIFI_DISCONNECT`
: Request a Wi-Fi disconnect.

`NET_REQUEST_WIFI_AP_ENABLE`
: Request a Wi-Fi access point enable.

`NET_REQUEST_WIFI_AP_DISABLE`
: Request a Wi-Fi access point disable.

`NET_REQUEST_WIFI_IFACE_STATUS`
: Request a Wi-Fi network interface status.

`NET_REQUEST_WIFI_PS`
: Request a Wi-Fi power save.

`NET_REQUEST_WIFI_TWT`
: Request a Wi-Fi TWT.

`NET_REQUEST_WIFI_PS_CONFIG`
: Request a Wi-Fi power save configuration.

`NET_REQUEST_WIFI_REG_DOMAIN`
: Request a Wi-Fi regulatory domain.

`NET_REQUEST_WIFI_MODE`
: Request current Wi-Fi mode.

NET_REQUEST_WIFI_PACKET_FILTER
: Request Wi-Fi packet filter.

NET_REQUEST_WIFI_CHANNEL
: Request a Wi-Fi channel.

NET_REQUEST_WIFI_AP_STA_DISCONNECT
: Request a Wi-Fi access point to disconnect a station.

NET_REQUEST_WIFI_VERSION
: Request a Wi-Fi version.

NET_REQUEST_WIFI_RTS_THRESHOLD
: Request a Wi-Fi RTS threshold.

NET_REQUEST_WIFI_AP_CONFIG_PARAM
: Request a Wi-Fi AP parameters configuration.

NET_EVENT_WIFI_SCAN_RESULT
: Event emitted for Wi-Fi scan result.

NET_EVENT_WIFI_SCAN_DONE
: Event emitted when Wi-Fi scan is done.

NET_EVENT_WIFI_CONNECT_RESULT
: Event emitted for Wi-Fi connect result.

NET_EVENT_WIFI_DISCONNECT_RESULT
: Event emitted for Wi-Fi disconnect result.

NET_EVENT_WIFI_IFACE_STATUS
: Event emitted for Wi-Fi network interface status.

NET_EVENT_WIFI_TWT
: Event emitted for Wi-Fi TWT information.

NET_EVENT_WIFI_TWT_SLEEP_STATE
: Event emitted for Wi-Fi TWT sleep state.

NET_EVENT_WIFI_RAW_SCAN_RESULT
: Event emitted for Wi-Fi raw scan result.

NET_EVENT_WIFI_DISCONNECT_COMPLETE
: Event emitted Wi-Fi disconnect is completed.

NET_EVENT_WIFI_AP_ENABLE_RESULT
: Event emitted for Wi-Fi access point enable result.

NET_EVENT_WIFI_AP_DISABLE_RESULT
: Event emitted for Wi-Fi access point disable result.

NET_EVENT_WIFI_AP_STA_CONNECTED
: Event emitted when Wi-Fi station is connected in AP mode.

NET_EVENT_WIFI_AP_STA_DISCONNECTED
: Event emitted Wi-Fi station is disconnected from AP.

MAX_REG_CHAN_NUM
: Max regulatory channel number.

Typedefs

typedef void (*scan_result_cb_t)(struct *net_if* *iface, int status, struct *wifi_scan_result* *entry)

Scan result callback.

Param iface
: Network interface

Param status
: Scan result status

Param entry
: Scan result entry

Enums

enum wifi_security_type

IEEE 802.11 security types.

Values:

enumerator WIFI_SECURITY_TYPE_NONE = 0
: No security.

enumerator WIFI_SECURITY_TYPE_PSK
: WPA2-PSK security.

enumerator WIFI_SECURITY_TYPE_PSK_SHA256
: WPA2-PSK-SHA256 security.

enumerator WIFI_SECURITY_TYPE_SAE
: WPA3-SAE security.

enumerator WIFI_SECURITY_TYPE_WAPI
: GB 15629.11-2003 WAPI security.

enumerator WIFI_SECURITY_TYPE_EAP
EAP security - Enterprise.

enumerator WIFI_SECURITY_TYPE_WEP
WEP security.

enumerator WIFI_SECURITY_TYPE_WPA_PSK
WPA-PSK security.

enumerator WIFI_SECURITY_TYPE_WPA_AUTO_PERSONAL
WPA/WPA2/WPA3 PSK security.

enum wifi_mfp_options
IEEE 802.11w - Management frame protection.
Values:

enumerator WIFI_MFP_DISABLE = 0
MFP disabled.

enumerator WIFI_MFP_OPTIONAL
MFP optional.

enumerator WIFI_MFP_REQUIRED
MFP required.

enum wifi_frequency_bands
IEEE 802.11 operational frequency bands (not exhaustive).
Values:

enumerator WIFI_FREQ_BAND_2_4_GHZ = 0
2.4 GHz band.

enumerator WIFI_FREQ_BAND_5_GHZ
5 GHz band.

enumerator WIFI_FREQ_BAND_6_GHZ
6 GHz band (Wi-Fi 6E, also extends to 7GHz).

enumerator __WIFI_FREQ_BAND_AFTER_LAST
Number of frequency bands available.

enumerator WIFI_FREQ_BAND_MAX = *__WIFI_FREQ_BAND_AFTER_LAST* - 1
Highest frequency band available.

enumerator WIFI_FREQ_BAND_UNKNOWN
Invalid frequency band.

enum `wifi_iface_state`

Wi-Fi interface states.

Based on https://w1.fi/wpa_supplicant/devel/defs_8h.html#a4aeb27c1e4abd046df3064ea9756f0bc

Values:

enumerator `WIFI_STATE_DISCONNECTED` = 0

Interface is disconnected.

enumerator `WIFI_STATE_INTERFACE_DISABLED`

Interface is disabled (administratively).

enumerator `WIFI_STATE_INACTIVE`

No enabled networks in the configuration.

enumerator `WIFI_STATE_SCANNING`

Interface is scanning for networks.

enumerator `WIFI_STATE_AUTHENTICATING`

Authentication with a network is in progress.

enumerator `WIFI_STATE_ASSOCIATING`

Association with a network is in progress.

enumerator `WIFI_STATE_ASSOCIATED`

Association with a network completed.

enumerator `WIFI_STATE_4WAY_HANDSHAKE`

4-way handshake with a network is in progress.

enumerator `WIFI_STATE_GROUP_HANDSHAKE`

Group Key exchange with a network is in progress.

enumerator `WIFI_STATE_COMPLETED`

All authentication completed, ready to pass data.

enum `wifi_iface_mode`

Wi-Fi interface modes.

Based on https://w1.fi/wpa_supplicant/devel/defs_8h.html#a4aeb27c1e4abd046df3064ea9756f0bc

Values:

enumerator `WIFI_MODE_INFRA` = 0

Infrastructure station mode.

enumerator `WIFI_MODE_IBSS` = 1

IBSS (ad-hoc) station mode.

enumerator WIFI_MODE_AP = 2
AP mode.

enumerator WIFI_MODE_P2P_GO = 3
P2P group owner mode.

enumerator WIFI_MODE_P2P_GROUP_FORMATION = 4
P2P group formation mode.

enumerator WIFI_MODE_MESH = 5
802.11s Mesh mode.

enum wifi_link_mode
Wi-Fi link operating modes.
As per https://en.wikipedia.org/wiki/Wi-Fi#Versions_and_generations.
Values:

enumerator WIFI_0 = 0
802.11 (legacy).

enumerator WIFI_1
802.11b.

enumerator WIFI_2
802.11a.

enumerator WIFI_3
802.11g.

enumerator WIFI_4
802.11n.

enumerator WIFI_5
802.11ac.

enumerator WIFI_6
802.11ax.

enumerator WIFI_6E
802.11ax 6GHz.

enumerator WIFI_7
802.11be.

enum wifi_scan_type
Wi-Fi scanning types.
Values:

enumerator WIFI_SCAN_TYPE_ACTIVE = 0
Active scanning (default).

enumerator WIFI_SCAN_TYPE_PASSIVE
Passive scanning.

enum wifi_ps
Wi-Fi power save states.
Values:

enumerator WIFI_PS_DISABLED = 0
Power save disabled.

enumerator WIFI_PS_ENABLED
Power save enabled.

enum wifi_ps_mode
Wi-Fi power save modes.
Values:

enumerator WIFI_PS_MODE_LEGACY = 0
Legacy power save mode.

enumerator WIFI_PS_MODE_WMM
WMM power save mode.

enum wifi_operational_modes
Wifi operational mode.
Values:

enumerator WIFI_STA_MODE = *BIT*(0)
STA mode setting enable.

enumerator WIFI_MONITOR_MODE = *BIT*(1)
Monitor mode setting enable.

enumerator WIFI_TX_INJECTION_MODE = *BIT*(2)
TX injection mode setting enable.

enumerator WIFI_PROMISCUOUS_MODE = *BIT*(3)
Promiscuous mode setting enable.

enumerator WIFI_AP_MODE = *BIT*(4)
AP mode setting enable.

enumerator WIFI_SOFTAP_MODE = *BIT*(5)
Softap mode setting enable.

enum **wifi_filter**

Mode filter settings.

Values:

enumerator **WIFI_PACKET_FILTER_ALL** = *BIT*(0)

Support management, data and control packet sniffing.

enumerator **WIFI_PACKET_FILTER_MGMT** = *BIT*(1)

Support only sniffing of management packets.

enumerator **WIFI_PACKET_FILTER_DATA** = *BIT*(2)

Support only sniffing of data packets.

enumerator **WIFI_PACKET_FILTER_CTRL** = *BIT*(3)

Support only sniffing of control packets.

enum **wifi_twt_operation**

Wi-Fi Target Wake Time (TWT) operations.

Values:

enumerator **WIFI_TWT_SETUP** = 0

TWT setup operation.

enumerator **WIFI_TWT_TEARDOWN**

TWT teardown operation.

enum **wifi_twt_negotiation_type**

Wi-Fi Target Wake Time (TWT) negotiation types.

Values:

enumerator **WIFI_TWT_INDIVIDUAL** = 0

TWT individual negotiation.

enumerator **WIFI_TWT_BROADCAST**

TWT broadcast negotiation.

enumerator **WIFI_TWT_WAKE_TBTT**

TWT wake TBTT negotiation.

enum **wifi_twt_setup_cmd**

Wi-Fi Target Wake Time (TWT) setup commands.

Values:

enumerator **WIFI_TWT_SETUP_CMD_REQUEST** = 0

TWT setup request.

enumerator **WIFI_TWT_SETUP_CMD_SUGGEST**

TWT setup suggest (parameters can be changed by AP)

enumerator WIFI_TWT_SETUP_CMD_DEMAND
TWT setup demand (parameters can not be changed by AP)

enumerator WIFI_TWT_SETUP_CMD_GROUPING
TWT setup grouping (grouping of TWT flows)

enumerator WIFI_TWT_SETUP_CMD_ACCEPT
TWT setup accept (parameters accepted by AP)

enumerator WIFI_TWT_SETUP_CMD_ALTERNATE
TWT setup alternate (alternate parameters suggested by AP)

enumerator WIFI_TWT_SETUP_CMD_DICTATE
TWT setup dictate (parameters dictated by AP)

enumerator WIFI_TWT_SETUP_CMD_REJECT
TWT setup reject (parameters rejected by AP)

enum wifi_twt_setup_resp_status
Wi-Fi Target Wake Time (TWT) negotiation status.
Values:

enumerator WIFI_TWT_RESP_RECEIVED = 0
TWT response received for TWT request.

enumerator WIFI_TWT_RESP_NOT_RECEIVED
TWT response not received for TWT request.

enum wifi_twt_fail_reason
Target Wake Time (TWT) error codes.
Values:

enumerator WIFI_TWT_FAIL_UNSPECIFIED
Unspecified error.

enumerator WIFI_TWT_FAIL_CMD_EXEC_FAIL
Command execution failed.

enumerator WIFI_TWT_FAIL_OPERATION_NOT_SUPPORTED
Operation not supported.

enumerator WIFI_TWT_FAIL_UNABLE_TO_GET_IFACE_STATUS
Unable to get interface status.

enumerator WIFI_TWT_FAIL_DEVICE_NOT_CONNECTED
Device not connected to AP.

enumerator WIFI_TWT_FAIL_PEER_NOT_HE_CAPAB
Peer not HE (802.11ax/Wi-Fi 6) capable.

enumerator WIFI_TWT_FAIL_PEER_NOT_TWT_CAPAB
Peer not TWT capable.

enumerator WIFI_TWT_FAIL_OPERATION_IN_PROGRESS
A TWT flow is already in progress.

enumerator WIFI_TWT_FAIL_INVALID_FLOW_ID
Invalid negotiated flow id.

enumerator WIFI_TWT_FAIL_IP_NOT_ASSIGNED
IP address not assigned or configured.

enumerator WIFI_TWT_FAIL_FLOW_ALREADY_EXISTS
Flow already exists.

enum wifi_twt_teardown_status
Wi-Fi Target Wake Time (TWT) teradown status.
Values:

enumerator WIFI_TWT_TEARDOWN_SUCCESS = 0
TWT teardown success.

enumerator WIFI_TWT_TEARDOWN_FAILED
TWT teardown failure.

enum wifi_ps_param_type
Wi-Fi power save parameters.
Values:

enumerator WIFI_PS_PARAM_STATE
Power save state.

enumerator WIFI_PS_PARAM_LISTEN_INTERVAL
Power save listen interval.

enumerator WIFI_PS_PARAM_WAKEUP_MODE
Power save wakeup mode.

enumerator WIFI_PS_PARAM_MODE
Power save mode.

enumerator WIFI_PS_PARAM_TIMEOUT
Power save timeout.

enum `wifi_ps_wakeup_mode`

Wi-Fi power save modes.

Values:

enumerator `WIFI_PS_WAKEUP_MODE_DTIM` = 0

DTIM based wakeup.

enumerator `WIFI_PS_WAKEUP_MODE_LISTEN_INTERVAL`

Listen interval based wakeup.

enum `wifi_config_ps_param_fail_reason`

Wi-Fi power save error codes.

Values:

enumerator `WIFI_PS_PARAM_FAIL_UNSPECIFIED`

Unspecified error.

enumerator `WIFI_PS_PARAM_FAIL_CMD_EXEC_FAIL`

Command execution failed.

enumerator `WIFI_PS_PARAM_FAIL_OPERATION_NOT_SUPPORTED`

Parameter not supported.

enumerator `WIFI_PS_PARAM_FAIL_UNABLE_TO_GET_IFACE_STATUS`

Unable to get interface status.

enumerator `WIFI_PS_PARAM_FAIL_DEVICE_NOT_CONNECTED`

Device not connected to AP.

enumerator `WIFI_PS_PARAM_FAIL_DEVICE_CONNECTED`

Device already connected to AP.

enumerator `WIFI_PS_PARAM_LISTEN_INTERVAL_RANGE_INVALID`

Listen interval out of range.

enum `wifi_ap_config_param`

Wi-Fi AP mode configuration parameter.

Values:

enumerator `WIFI_AP_CONFIG_PARAM_MAX_INACTIVITY` = *BIT*(0)

Used for AP mode configuration parameter ap_max_inactivity.

enumerator `WIFI_AP_CONFIG_PARAM_MAX_NUM_STA` = *BIT*(1)

Used for AP mode configuration parameter max_num_sta.

enum `net_request_wifi_cmd`

Wi-Fi management commands.

Values:

enumerator NET_REQUEST_WIFI_CMD_SCAN = 1
Scan for Wi-Fi networks.

enumerator NET_REQUEST_WIFI_CMD_CONNECT
Connect to a Wi-Fi network.

enumerator NET_REQUEST_WIFI_CMD_DISCONNECT
Disconnect from a Wi-Fi network.

enumerator NET_REQUEST_WIFI_CMD_AP_ENABLE
Enable AP mode.

enumerator NET_REQUEST_WIFI_CMD_AP_DISABLE
Disable AP mode.

enumerator NET_REQUEST_WIFI_CMD_IFACE_STATUS
Get interface status.

enumerator NET_REQUEST_WIFI_CMD_PS
Set power save status.

enumerator NET_REQUEST_WIFI_CMD_TWT
Setup or teardown TWT flow.

enumerator NET_REQUEST_WIFI_CMD_PS_CONFIG
Get power save config.

enumerator NET_REQUEST_WIFI_CMD_REG_DOMAIN
Set or get regulatory domain.

enumerator NET_REQUEST_WIFI_CMD_MODE
Set or get Mode of operation.

enumerator NET_REQUEST_WIFI_CMD_PACKET_FILTER
Set or get packet filter setting for current mode.

enumerator NET_REQUEST_WIFI_CMD_CHANNEL
Set or get Wi-Fi channel for Monitor or TX-Injection mode.

enumerator NET_REQUEST_WIFI_CMD_AP_STA_DISCONNECT
Disconnect a STA from AP.

enumerator NET_REQUEST_WIFI_CMD_VERSION
Get Wi-Fi driver and Firmware versions.

enumerator NET_REQUEST_WIFI_CMD_RTS_THRESHOLD
Set RTS threshold.

enumerator NET_REQUEST_WIFI_CMD_AP_CONFIG_PARAM
Configure AP parameter.

enum net_event_wifi_cmd
Wi-Fi management events.
Values:

enumerator NET_EVENT_WIFI_CMD_SCAN_RESULT = 1
Scan results available.

enumerator NET_EVENT_WIFI_CMD_SCAN_DONE
Scan done.

enumerator NET_EVENT_WIFI_CMD_CONNECT_RESULT
Connect result.

enumerator NET_EVENT_WIFI_CMD_DISCONNECT_RESULT
Disconnect result.

enumerator NET_EVENT_WIFI_CMD_IFACE_STATUS
Interface status.

enumerator NET_EVENT_WIFI_CMD_TWT
TWT events.

enumerator NET_EVENT_WIFI_CMD_TWT_SLEEP_STATE
TWT sleep status: awake or sleeping, can be used by application to determine if it can send data or not.

enumerator NET_EVENT_WIFI_CMD_RAW_SCAN_RESULT
Raw scan results available.

enumerator NET_EVENT_WIFI_CMD_DISCONNECT_COMPLETE
Disconnect complete.

enumerator NET_EVENT_WIFI_CMD_AP_ENABLE_RESULT
AP mode enable result.

enumerator NET_EVENT_WIFI_CMD_AP_DISABLE_RESULT
AP mode disable result.

enumerator NET_EVENT_WIFI_CMD_AP_STA_CONNECTED
STA connected to AP.

enumerator NET_EVENT_WIFI_CMD_AP_STA_DISCONNECTED
STA disconnected from AP.

enum `wifi_conn_status`

Wi-Fi connect result codes.

To be overlaid on top of *wifi_status* in the connect result event for detailed status.

Values:

enumerator `WIFI_STATUS_CONN_SUCCESS` = 0

Connection successful.

enumerator `WIFI_STATUS_CONN_FAIL`

Connection failed - generic failure.

enumerator `WIFI_STATUS_CONN_WRONG_PASSWORD`

Connection failed - wrong password Few possible reasons for 4-way handshake failure that we can guess are as follows: 1) Incorrect key 2) EAPoL frames lost causing timeout.

#1 is the likely cause, so, we convey to the user that it is due to Wrong passphrase/password.

enumerator `WIFI_STATUS_CONN_TIMEOUT`

Connection timed out.

enumerator `WIFI_STATUS_CONN_AP_NOT_FOUND`

Connection failed - AP not found.

enumerator `WIFI_STATUS_CONN_LAST_STATUS`

Last connection status.

enumerator `WIFI_STATUS_DISCONN_FIRST_STATUS` = *WIFI_STATUS_CONN_LAST_STATUS*

Connection disconnected status.

enum `wifi_disconn_reason`

Wi-Fi disconnect reason codes.

To be overlaid on top of *wifi_status* in the disconnect result event for detailed reason.

Values:

enumerator `WIFI_REASON_DISCONN_SUCCESS` = 0

Success, overload status as reason.

enumerator `WIFI_REASON_DISCONN_UNSPECIFIED`

Unspecified reason.

enumerator `WIFI_REASON_DISCONN_USER_REQUEST`

Disconnected due to user request.

enumerator `WIFI_REASON_DISCONN_AP_LEAVING`

Disconnected due to AP leaving.

enumerator WIFI_REASON_DISCONN_INACTIVITY

Disconnected due to inactivity.

enum wifi_ap_status

Wi-Fi AP mode result codes.

To be overlaid on top of *wifi_status* in the AP mode enable or disable result event for detailed status.

Values:

enumerator WIFI_STATUS_AP_SUCCESS = 0

AP mode enable or disable successful.

enumerator WIFI_STATUS_AP_FAIL

AP mode enable or disable failed - generic failure.

enumerator WIFI_STATUS_AP_CHANNEL_NOT_SUPPORTED

AP mode enable failed - channel not supported.

enumerator WIFI_STATUS_AP_CHANNEL_NOT_ALLOWED

AP mode enable failed - channel not allowed.

enumerator WIFI_STATUS_AP_SSID_NOT_ALLOWED

AP mode enable failed - SSID not allowed.

enumerator WIFI_STATUS_AP_AUTH_TYPE_NOT_SUPPORTED

AP mode enable failed - authentication type not supported.

enumerator WIFI_STATUS_AP_OP_NOT_SUPPORTED

AP mode enable failed - operation not supported.

enumerator WIFI_STATUS_AP_OP_NOT_PERMITTED

AP mode enable failed - operation not permitted.

enum wifi_mgmt_op

Generic get/set operation for any command.

Values:

enumerator WIFI_MGMT_GET = 0

Get operation.

enumerator WIFI_MGMT_SET = 1

Set operation.

enum wifi_twt_sleep_state

Wi-Fi TWT sleep states.

Values:

enumerator **WIFI_TWT_STATE_SLEEP** = 0
TWT sleep state: sleeping.

enumerator **WIFI_TWT_STATE_AWAKE** = 1
TWT sleep state: awake.

Functions

const char ***wifi_security_txt**(enum *wifi_security_type* security)
Helper function to get user-friendly security type name.

const char ***wifi_mfp_txt**(enum *wifi_mfp_options* mfp)
Helper function to get user-friendly MFP name.

const char ***wifi_band_txt**(enum *wifi_frequency_bands* band)
Helper function to get user-friendly frequency band name.

const char ***wifi_state_txt**(enum *wifi_iface_state* state)
Helper function to get user-friendly interface state name.

const char ***wifi_mode_txt**(enum *wifi_iface_mode* mode)
Helper function to get user-friendly interface mode name.

const char ***wifi_link_mode_txt**(enum *wifi_link_mode* link_mode)
Helper function to get user-friendly link mode name.

const char ***wifi_ps_txt**(enum *wifi_ps* ps_name)
Helper function to get user-friendly ps name.

const char ***wifi_ps_mode_txt**(enum *wifi_ps_mode* ps_mode)
Helper function to get user-friendly ps mode name.

const char ***wifi_twt_operation_txt**(enum *wifi_twt_operation* twt_operation)
Helper function to get user-friendly twt operation name.

const char ***wifi_twt_negotiation_type_txt**(enum *wifi_twt_negotiation_type* twt_negotiation)
Helper function to get user-friendly twt negotiation type name.

const char ***wifi_twt_setup_cmd_txt**(enum *wifi_twt_setup_cmd* twt_setup)
Helper function to get user-friendly twt setup cmd name.

static inline const char ***wifi_twt_get_err_code_str**(int16_t err_no)
Helper function to get user-friendly TWT error code name.

const char ***wifi_ps_wakeup_mode_txt**(enum *wifi_ps_wakeup_mode* ps_wakeup_mode)
Helper function to get user-friendly ps wakeup mode name.

static inline const char ***wifi_ps_get_config_err_code_str**(int16_t err_no)
Helper function to get user-friendly power save error code name.

void **wifi_mgmt_raise_connect_result_event**(struct *net_if* *iface, int status)
Wi-Fi management connect result event.

Parameters

- **iface** – Network interface
- **status** – Connect result status

void **wifi_mgmt_raise_disconnect_result_event**(struct *net_if* *iface, int status)

Wi-Fi management disconnect result event.

Parameters

- **iface** – Network interface
- **status** – Disconnect result status

void **wifi_mgmt_raise_iface_status_event**(struct *net_if* *iface, struct *wifi_iface_status* *iface_status)

Wi-Fi management interface status event.

Parameters

- **iface** – Network interface
- **iface_status** – Interface status

void **wifi_mgmt_raise_twt_event**(struct *net_if* *iface, struct *wifi_twt_params* *twt_params)

Wi-Fi management TWT event.

Parameters

- **iface** – Network interface
- **twt_params** – TWT parameters

void **wifi_mgmt_raise_twt_sleep_state**(struct *net_if* *iface, int twt_sleep_state)

Wi-Fi management TWT sleep state event.

Parameters

- **iface** – Network interface
- **twt_sleep_state** – TWT sleep state

void **wifi_mgmt_raise_raw_scan_result_event**(struct *net_if* *iface, struct *wifi_raw_scan_result* *raw_scan_info)

Wi-Fi management raw scan result event.

Parameters

- **iface** – Network interface
- **raw_scan_info** – Raw scan result

void **wifi_mgmt_raise_disconnect_complete_event**(struct *net_if* *iface, int status)

Wi-Fi management disconnect complete event.

Parameters

- **iface** – Network interface
- **status** – Disconnect complete status

void **wifi_mgmt_raise_ap_enable_result_event**(struct *net_if* *iface, enum *wifi_ap_status* status)

Wi-Fi management AP mode enable result event.

Parameters

- **iface** – Network interface
- **status** – AP mode enable result status

void **wifi_mgmt_raise_ap_disable_result_event**(struct *net_if* *iface, enum *wifi_ap_status* status)

Wi-Fi management AP mode disable result event.

Parameters

- **iface** – Network interface
- **status** – AP mode disable result status

void **wifi_mgmt_raise_ap_sta_connected_event**(struct *net_if* *iface, struct *wifi_ap_sta_info* *sta_info)

Wi-Fi management AP mode STA connected event.

Parameters

- **iface** – Network interface
- **sta_info** – STA information

void **wifi_mgmt_raise_ap_sta_disconnected_event**(struct *net_if* *iface, struct *wifi_ap_sta_info* *sta_info)

Wi-Fi management AP mode STA disconnected event.

Parameters

- **iface** – Network interface
- **sta_info** – STA information

struct **wifi_version**

#include <*wifi_mgmt.h*> Wi-Fi version.

Public Members

const char ***drv_version**

Driver version.

const char ***fw_version**

Firmware version.

struct **wifi_band_channel**

#include <*wifi_mgmt.h*> Wi-Fi structure to uniquely identify a band-channel pair.

Public Members

uint8_t **band**

Frequency band.

uint8_t **channel**

Channel.

struct **wifi_scan_params**

#include <*wifi_mgmt.h*> Wi-Fi scan parameters structure.

Used to specify parameters which can control how the Wi-Fi scan is performed.

Public Members

enum *wifi_scan_type* **scan_type**

Scan type, see enum wifi_scan_type.

The scan_type is only a hint to the underlying Wi-Fi chip for the preferred mode of scan. The actual mode of scan can depend on factors such as the Wi-Fi chip implementation support, regulatory domain restrictions etc.

uint8_t **bands**

Bitmap of bands to be scanned.

Refer to *wifi_frequency_bands* for bit position of each band.

uint16_t **dwell_time_active**

Active scan dwell time (in ms) on a channel.

uint16_t **dwell_time_passive**

Passive scan dwell time (in ms) on a channel.

const char ***ssids**[WIFI_MGMT_SCAN_SSID_FILT_MAX]

Array of SSID strings to scan.

uint16_t **max_bss_cnt**

Specifies the maximum number of scan results to return.

These results would be the BSSIDS with the best RSSI values, in all the scanned channels. This should only be used to limit the number of returned scan results, and cannot be counted upon to limit the scan time, since the underlying Wi-Fi chip might have to scan all the channels to find the max_bss_cnt number of APs with the best signal strengths. A value of 0 signifies that there is no restriction on the number of scan results to be returned.

struct *wifi_band_channel* **band_chan**[WIFI_MGMT_SCAN_CHAN_MAX_MANUAL]

Channel information array indexed on Wi-Fi frequency bands and channels within that band.

E.g. to scan channel 6 and 11 on the 2.4 GHz band, channel 36 on the 5 GHz band:

```
chan[0] = {WIFI_FREQ_BAND_2_4_GHZ, 6};
chan[1] = {WIFI_FREQ_BAND_2_4_GHZ, 11};
chan[2] = {WIFI_FREQ_BAND_5_GHZ, 36};
```

This list specifies the channels to be **considered for scan**. The underlying Wi-Fi chip can silently omit some channels due to various reasons such as channels not conforming to regulatory restrictions etc. The invoker of the API should ensure that the channels specified follow regulatory rules.

struct **wifi_scan_result**

#include <wifi_mgmt.h> Wi-Fi scan result, each result is provided to the *net_mgmt_event_callback* via its info attribute (see net_mgmt.h)

Public Members

uint8_t **ssid**[*WIFI_SSID_MAX_LEN*]
SSID.

uint8_t **ssid_length**
SSID length.

uint8_t **band**
Frequency band.

uint8_t **channel**
Channel.

enum *wifi_security_type* **security**
Security type.

enum *wifi_mfp_options* **mfp**
MFP options.

int8_t **rssi**
RSSI.

uint8_t **mac**[*WIFI_MAC_ADDR_LEN*]
BSSID.

uint8_t **mac_length**
BSSID length.

struct **wifi_connect_req_params**
#include <*wifi_mgmt.h*> Wi-Fi connect request parameters.

Public Members

const uint8_t ***ssid**
SSID.

uint8_t **ssid_length**
SSID length.

const uint8_t ***psk**
Pre-shared key.

uint8_t **psk_length**
Pre-shared key length.

const uint8_t ***sae_password**
SAE password (same as PSK but with no length restrictions), optional.

uint8_t **sae_password_length**
SAE password length.

uint8_t **band**
Frequency band.

uint8_t **channel**
Channel.

enum *wifi_security_type* **security**
Security type.

enum *wifi_mfp_options* **mfp**
MFP options.

uint8_t **bssid**[*WIFI_MAC_ADDR_LEN*]
BSSID.

int **timeout**
Connect timeout in seconds, SYS_FOREVER_MS for no timeout.

struct **wifi_status**
#include <*wifi_mgmt.h*> Generic Wi-Fi status for commands and events.

Public Members

int **status**
Status value.

enum *wifi_conn_status* **conn_status**
Connection status.

enum *wifi_disconn_reason* **disconn_reason**
Disconnection reason status.

enum *wifi_ap_status* **ap_status**
Access point status.

struct **wifi_iface_status**
#include <*wifi_mgmt.h*> Wi-Fi interface status.

Public Members

int **state**
Interface state, see enum wifi_iface_state.

unsigned int **ssid_len**
SSID length.

char **ssid**[*WIFI_SSID_MAX_LEN*]
SSID.

char **bssid**[*WIFI_MAC_ADDR_LEN*]
BSSID.

enum *wifi_frequency_bands* **band**
Frequency band.

unsigned int **channel**
Channel.

enum *wifi_iface_mode* **iface_mode**
Interface mode, see enum wifi_iface_mode.

enum *wifi_link_mode* **link_mode**
Link mode, see enum wifi_link_mode.

enum *wifi_security_type* **security**
Security type, see enum wifi_security_type.

enum *wifi_mfp_options* **mfp**
MFP options, see enum wifi_mfp_options.

int **rssi**
RSSI.

unsigned char **dtim_period**
DTIM period.

unsigned short **beacon_interval**
Beacon interval.

bool **twt_capable**
is TWT capable?

struct **wifi_ps_params**
#include <*wifi_mgmt.h*> Wi-Fi power save parameters.

Public Members

enum *wifi_ps* **enabled**
Power save state.

unsigned short **listen_interval**

Listen interval.

enum *wifi_ps_wakeup_mode* **wakeup_mode**

Wi-Fi power save wakeup mode.

enum *wifi_ps_mode* **mode**

Wi-Fi power save mode.

unsigned int **timeout_ms**

Wi-Fi power save timeout.

This is the time out to wait after sending a TX packet before going back to power save (in ms) to receive any replies from the AP. Zero means this feature is disabled.

It's a tradeoff between power consumption and latency.

enum *wifi_ps_param_type* **type**

Wi-Fi power save type.

enum *wifi_config_ps_param_fail_reason* **fail_reason**

Wi-Fi power save fail reason.

struct **wifi_twt_params**

#include <wifi_mgmt.h> Wi-Fi TWT parameters.

Public Members

enum *wifi_twt_operation* **operation**

TWT operation, see enum wifi_twt_operation.

enum *wifi_twt_negotiation_type* **negotiation_type**

TWT negotiation type, see enum wifi_twt_negotiation_type.

enum *wifi_twt_setup_cmd* **setup_cmd**

TWT setup command, see enum wifi_twt_setup_cmd.

enum *wifi_twt_setup_resp_status* **resp_status**

TWT setup response status, see enum wifi_twt_setup_resp_status.

enum *wifi_twt_teardown_status* **teardown_status**

TWT teardown cmd status, see enum wifi_twt_teardown_status.

uint8_t **dialog_token**

Dialog token, used to map requests to responses.

uint8_t **flow_id**

Flow ID, used to map setup with teardown.

uint64_t **twt_interval**
: Interval = Wake up time + Sleeping time.

bool **responder**
: Requestor or responder.

bool **trigger**
: Trigger enabled or disabled.

bool **implicit**
: Implicit or explicit.

bool **announce**
: Announced or unannounced.

uint32_t **twt_wake_interval**
: Wake up time.

uint32_t **twt_wake_ahead_duration**
: Wake ahead notification is sent earlier than TWT Service period (SP) start based on this duration.

  This should give applications ample time to prepare the data before TWT SP starts.

struct *wifi_twt_params* **setup**
: Setup specific parameters.

bool **teardown_all**
: Teardown all flows.

struct *wifi_twt_params* **teardown**
: Teardown specific parameters.

enum *wifi_twt_fail_reason* **fail_reason**
: TWT fail reason, see enum wifi_twt_fail_reason.

struct **wifi_twt_flow_info**
: *#include <wifi_mgmt.h>* Wi-Fi TWT flow information.

Public Members

uint64_t **twt_interval**
: Interval = Wake up time + Sleeping time.

uint8_t **dialog_token**
: Dialog token, used to map requests to responses.

uint8_t **flow_id**
: Flow ID, used to map setup with teardown.

enum *wifi_twt_negotiation_type* **negotiation_type**
TWT negotiation type, see enum wifi_twt_negotiation_type.

bool **responder**
Requestor or responder.

bool **trigger**
Trigger enabled or disabled.

bool **implicit**
Implicit or explicit.

bool **announce**
Announced or unannounced.

uint32_t **twt_wake_interval**
Wake up time.

uint32_t **twt_wake_ahead_duration**
Wake ahead duration.

struct **wifi_ps_config**
#include <*wifi_mgmt.h*> Wi-Fi power save configuration.

Public Members

char **num_twt_flows**
Number of TWT flows.

struct *wifi_twt_flow_info* **twt_flows**[WIFI_MAX_TWT_FLOWS]
TWT flow details.

struct *wifi_ps_params* **ps_params**
Power save configuration.

struct **wifi_reg_chan_info**
#include <*wifi_mgmt.h*> Per-channel regulatory attributes.

Public Members

unsigned short **center_frequency**
Center frequency in MHz.

unsigned short **max_power**
Maximum transmission power (in dBm)

unsigned short **supported**
Is channel supported or not.

unsigned short **passive_only**
Passive transmissions only.

unsigned short **dfs**
Is a DFS channel.

struct **wifi_reg_domain**
#include <*wifi_mgmt.h*> Regulatory domain information or configuration.

Public Members

enum *wifi_mgmt_op* **oper**
Regulatory domain operation.

bool **force**
Ignore all other regulatory hints over this one.

uint8_t **country_code**[*WIFI_COUNTRY_CODE_LEN*]
Country code: ISO/IEC 3166-1 alpha-2.

unsigned int **num_channels**
Number of channels supported.

struct *wifi_reg_chan_info* ***chan_info**
Channels information.

struct **wifi_raw_scan_result**
#include <*wifi_mgmt.h*> Wi-Fi raw scan result.

Public Members

int8_t **rssi**
RSSI.

int **frame_length**
Frame length.

unsigned short **frequency**
Frequency.

uint8_t **data**[CONFIG_WIFI_MGMT_RAW_SCAN_RESULT_LENGTH]
Raw scan data.

struct **wifi_ap_sta_info**
#include <*wifi_mgmt.h*> AP mode - connected STA details.

Public Members

enum *wifi_link_mode* **link_mode**
Link mode, see enum wifi_link_mode.

uint8_t **mac**[*WIFI_MAC_ADDR_LEN*]
MAC address.

uint8_t **mac_length**
MAC address length.

bool **twt_capable**
is TWT capable ?

struct **wifi_mode_info**
#include <*wifi_mgmt.h*> Wi-Fi mode setup.

Public Members

uint8_t **mode**
Mode setting for a specific mode of operation.

uint8_t **if_index**
Interface index.

enum *wifi_mgmt_op* **oper**
Get or set operation.

struct **wifi_filter_info**
#include <*wifi_mgmt.h*> Wi-Fi filter setting for monitor, prmoiscuous, TX-injection modes.

Public Members

uint8_t **filter**
Filter setting.

uint8_t **if_index**
Interface index.

uint16_t **buffer_size**
Filter buffer size.

enum *wifi_mgmt_op* **oper**
Get or set operation.

struct **wifi_channel_info**
#include <*wifi_mgmt.h*> Wi-Fi channel setting for monitor and TX-injection modes.

Public Members

uint16_t **channel**
Channel value to set.

uint8_t **if_index**
Interface index.

enum *wifi_mgmt_op* **oper**
Get or set operation.

struct **wifi_ap_config_params**
#include <*wifi_mgmt.h*> Wi-Fi AP configuration parameter.

Public Members

enum *wifi_ap_config_param* **type**
Parameter used to identify the different AP parameters.

uint32_t **max_inactivity**
Parameter used for setting maximum inactivity duration for stations.

uint32_t **max_num_sta**
Parameter used for setting maximum number of stations.

struct **wifi_mgmt_ops**
#include <*wifi_mgmt.h*> Wi-Fi management API.

Public Members

int (***scan**)(const struct *device* *dev, struct *wifi_scan_params* *params, *scan_result_cb_t* cb)
Scan for Wi-Fi networks.
Param dev
Pointer to the device structure for the driver instance.
Param params
Scan parameters
Param cb
Callback to be called for each result cb parameter is the cb that should be called for each result by the driver. The wifi mgmt part will take care of raising the necessary event etc.
Return
0 if ok, < 0 if error

int (***connect**)(const struct *device* *dev, struct *wifi_connect_req_params* *params)
Connect to a Wi-Fi network.
Param dev
Pointer to the device structure for the driver instance.

Param params
Connect parameters
Return
0 if ok, < 0 if error

int (*`disconnect`)(const struct *device* *dev)

Disconnect from a Wi-Fi network.
Param dev
Pointer to the device structure for the driver instance.
Return
0 if ok, < 0 if error

int (*`ap_enable`)(const struct *device* *dev, struct *wifi_connect_req_params* *params)

Enable AP mode.
Param dev
Pointer to the device structure for the driver instance.
Param params
AP mode parameters
Return
0 if ok, < 0 if error

int (*`ap_disable`)(const struct *device* *dev)

Disable AP mode.
Param dev
Pointer to the device structure for the driver instance.
Return
0 if ok, < 0 if error

int (*`ap_sta_disconnect`)(const struct *device* *dev, const uint8_t *mac)

Disconnect a STA from AP.
Param dev
Pointer to the device structure for the driver instance.
Param mac
MAC address of the STA to disconnect
Return
0 if ok, < 0 if error

int (*`iface_status`)(const struct *device* *dev, struct *wifi_iface_status* *status)

Get interface status.
Param dev
Pointer to the device structure for the driver instance.
Param status
Interface status
Return
0 if ok, < 0 if error

int (*`get_stats`)(const struct *device* *dev, struct *net_stats_wifi* *stats)

Get Wi-Fi statistics.
Param dev
Pointer to the device structure for the driver instance.
Param stats
Wi-Fi statistics
Return
0 if ok, < 0 if error

int (*`set_power_save`)(const struct *device* *dev, struct *wifi_ps_params* *params)

Set power save status.

Param dev
Pointer to the device structure for the driver instance.

Param params
Power save parameters

Return
0 if ok, < 0 if error

int (*`set_twt`)(const struct *device* *dev, struct *wifi_twt_params* *params)

Setup or teardown TWT flow.

Param dev
Pointer to the device structure for the driver instance.

Param params
TWT parameters

Return
0 if ok, < 0 if error

int (*`get_power_save_config`)(const struct *device* *dev, struct *wifi_ps_config* *config)

Get power save config.

Param dev
Pointer to the device structure for the driver instance.

Param config
Power save config

Return
0 if ok, < 0 if error

int (*`reg_domain`)(const struct *device* *dev, struct *wifi_reg_domain* *reg_domain)

Set or get regulatory domain.

Param dev
Pointer to the device structure for the driver instance.

Param reg_domain
Regulatory domain

Return
0 if ok, < 0 if error

int (*`filter`)(const struct *device* *dev, struct *wifi_filter_info* *filter)

Set or get packet filter settings for monitor and promiscuous modes.

Param dev
Pointer to the device structure for the driver instance.

Param packet
filter settings

Return
0 if ok, < 0 if error

int (*`mode`)(const struct *device* *dev, struct *wifi_mode_info* *mode)

Set or get mode of operation.

Param dev
Pointer to the device structure for the driver instance.

Param mode
settings

Return
0 if ok, < 0 if error

int (*`channel`)(const struct *device* *dev, struct *wifi_channel_info* *channel)

Set or get current channel of operation.

Param dev
Pointer to the device structure for the driver instance.

Param channel
settings

Return
0 if ok, < 0 if error

int (*`get_version`)(const struct *device* *dev, struct *wifi_version* *params)

Get Version of WiFi driver and Firmware.

The driver that implements the get_version function must not use stack to allocate the version information pointers that are returned as params struct members. The version pointer parameters should point to a static memory either in ROM (preferred) or in RAM.

Param dev
Pointer to the device structure for the driver instance

Param params
Version parameters

Return
0 if ok, < 0 if error

int (*`set_rts_threshold`)(const struct *device* *dev, unsigned int rts_threshold)

Set RTS threshold value.

Param dev
Pointer to the device structure for the driver instance.

Param RTS
threshold value

Return
0 if ok, < 0 if error

int (*`ap_config_params`)(const struct *device* *dev, struct *wifi_ap_config_params* *params)

Configure AP parameter.

Param dev
Pointer to the device structure for the driver instance.

Param params
AP mode parameter configuration parameter info

Return
0 if ok, < 0 if error

struct `net_wifi_mgmt_offload`

#include <*wifi_mgmt.h*> Wi-Fi management offload API.

Public Members

struct *ethernet_api* `wifi_iface`

Mandatory to get in first position.

A network device should indeed provide a pointer on such net_if_api structure. So we make current structure pointer that can be casted to a net_if_api structure pointer. Ethernet API

const struct *wifi_mgmt_ops* *const **wifi_mgmt_api**
Wi-Fi management API.

const void ***wifi_drv_ops**
Wi-Fi supplicant driver API.

Protocols

CoAP

- *Overview*
- *Sample Usage*
  - *CoAP Server*
  - *CoAP Client*
- *Testing*
  - *libcoap*
  - *TTCN3*
- *API Reference*

Overview The Constrained Application Protocol (CoAP) is a specialized web transfer protocol for use with constrained nodes and constrained (e.g., low-power, lossy) networks. It provides a convenient API for RESTful Web services that support CoAP's features. For more information about the protocol itself, see IETF RFC7252 The Constrained Application Protocol.

Zephyr provides a CoAP library which supports client and server roles. The library can be enabled with `CONFIG_COAP` Kconfig option and is configurable as per user needs. The Zephyr CoAP library is implemented using plain buffers. Users of the API create sockets for communication and pass the buffer to the library for parsing and other purposes. The library itself doesn't create any sockets for users.

On top of CoAP, Zephyr has support for LWM2M "Lightweight Machine 2 Machine" protocol, a simple, low-cost remote management and service enablement mechanism. See *Lightweight M2M (LWM2M)* for more information.

Supported RFCs:

- RFC7252: The Constrained Application Protocol (CoAP)
- RFC6690: Constrained RESTful Environments (CoRE) Link Format
- RFC7959: Block-Wise Transfers in the Constrained Application Protocol (CoAP)
- RFC7641: Observing Resources in the Constrained Application Protocol (CoAP)

Note

Not all parts of these RFCs are supported. Features are supported based on Zephyr requirements.

Sample Usage

Note

A *CoAP server* subsystem is available, the following is for creating a custom server implementation.

CoAP Server To create a CoAP server, resources for the server need to be defined. The `.well-known/core` resource should be added before all other resources that should be included in the responses of the `.well-known/core` resource.

```
static struct coap_resource resources[] = {
    { .get = well_known_core_get,
      .path = COAP_WELL_KNOWN_CORE_PATH,
    },
    { .get  = sample_get,
      .post = sample_post,
      .del  = sample_del,
      .put  = sample_put,
      .path = sample_path
    },
    { },
};
```

An application reads data from the socket and passes the buffer to the CoAP library to parse the message. If the CoAP message is proper, the library uses the buffer along with resources defined above to call the correct callback function to handle the CoAP request from the client. It's the callback function's responsibility to either reply or act according to CoAP request.

```
coap_packet_parse(&request, data, data_len, options, opt_num);
...
coap_handle_request(&request, resources, options, opt_num,
                    client_addr, client_addr_len);
```

If `CONFIG_COAP_URI_WILDCARD` enabled, server may accept multiple resources using MQTT-like wildcard style:

- the plus symbol represents a single-level wild card in the path;
- the hash symbol represents the multi-level wild card in the path.

```
static const char * const led_set[] = { "led","+","set", NULL };
static const char * const btn_get[] = { "button","#", NULL };
static const char * const no_wc[] = { "test","+1", NULL };
```

It accepts /led/0/set, led/1234/set, led/any/set, /button/door/1, /test/+1, but returns -ENOENT for /led/1, /test/21, /test/1.

This option is enabled by default, disable it to avoid unexpected behaviour with resource path like '/some_resource/+/#'.

Note

A *CoAP client* subsystem is available, the following is for creating a custom client implementation.

CoAP Client If the CoAP client knows about resources in the CoAP server, the client can start prepare CoAP requests and wait for responses. If the client doesn't know about resources in the CoAP server, it can request resources through the `.well-known/core` CoAP message.

```
/* Initialize the CoAP message */
char *path = "test";
struct coap_packet request;
uint8_t data[100];
uint8_t payload[20];

coap_packet_init(&request, data, sizeof(data),
                 1, COAP_TYPE_CON, 8, coap_next_token(),
                 COAP_METHOD_GET, coap_next_id());

/* Append options */
coap_packet_append_option(&request, COAP_OPTION_URI_PATH,
                          path, strlen(path));

/* Append Payload marker if you are going to add payload */
coap_packet_append_payload_marker(&request);

/* Append payload */
coap_packet_append_payload(&request, (uint8_t *)payload,
                           sizeof(payload) - 1);

/* send over sockets */
```

Testing There are various ways to test Zephyr CoAP library.

libcoap libcoap implements a lightweight application-protocol for devices that are resource constrained, such as by computing power, RF range, memory, bandwidth, or network packet sizes. Sources can be found here libcoap. libcoap has a script (`examples/etsi_coaptest.sh`) to test coap-server functionality in Zephyr.

See the net-tools project for more details

The coap-server sample can be built and executed on QEMU as described in *Networking with QEMU*.

Use this command on the host to run the libcoap implementation of the ETSI test cases:

```
sudo ./libcoap/examples/etsi_coaptest.sh -i tap0 2001:db8::1
```

TTCN3 Eclipse has TTCN3 based tests to run against CoAP implementations.

Install eclipse-titan and set symbolic links for titan tools

```
sudo apt-get install eclipse-titan

cd /usr/share/titan

sudo ln -s /usr/bin bin
sudo ln /usr/bin/titanver bin
sudo ln -s /usr/bin/mctr_cli bin
sudo ln -s /usr/include/titan include
sudo ln -s /usr/lib/titan lib

export TTCN3_DIR=/usr/share/titan

git clone https://gitlab.eclipse.org/eclipse/titan/titan.misc.git

cd titan.misc
```

Follow the instruction to setup CoAP test suite from here:

- https://gitlab.eclipse.org/eclipse/titan/titan.misc
- https://gitlab.eclipse.org/eclipse/titan/titan.misc/-/tree/master/CoAP_Conf

After the build is complete, the coap-server sample can be built and executed on QEMU as described in *Networking with QEMU*.

Change the client (test suite) and server (Zephyr coap-server sample) addresses in coap.cfg file as per your setup.

Execute the test cases with following command.

```
ttcn3_start coaptests coap.cfg
```

Sample output of ttcn3 tests looks like this.

```
Verdict statistics: 0 none (0.00 %), 10 pass (100.00 %), 0 inconc (0.00 %), 0 fail (0.00 %),
↪ 0 error (0.00 %).
Test execution summary: 10 test cases were executed. Overall verdict: pass
```

Related code samples

CoAP client
Use the CoAP library to implement a client that fetches a resource.

CoAP service
Use the CoAP server subsystem to register CoAP resources.

API Reference

group coap

COAP library.

Since
1.10

Version
0.8.0

Defines

COAP_MAKE_RESPONSE_CODE(class, det)

Utility macro to create a CoAP response code.

Parameters

- **class** – Class of the response code (ex. 2, 4, 5, ...)
- **det** – Detail of the response code

Returns
Response code literal

COAP_WELL_KNOWN_CORE_PATH

This resource should be added before all other resources that should be included in the responses of the .well-known/core resource if is to be used with coap_well_known_core_get.

Typedefs

typedef int (*`coap_method_t`)(struct *coap_resource* *resource, struct *coap_packet* *request, struct *sockaddr* *addr, *socklen_t* addr_len)

Type of the callback being called when a resource's method is invoked by the remote entity.

typedef void (*`coap_notify_t`)(struct *coap_resource* *resource, struct *coap_observer* *observer)

Type of the callback being called when a resource's has observers to be informed when an update happens.

typedef int (*`coap_reply_t`)(const struct *coap_packet* *response, struct *coap_reply* *reply, const struct *sockaddr* *from)

Helper function to be called when a response matches the a pending request.

When sending blocks, the callback is only executed when the reply of the last block is received. i.e. it is not called when the code of the reply is 'continue' (2.31).

Enums

enum `coap_option_num`

Set of CoAP packet options we are aware of.

Users may add options other than these to their packets, provided they know how to format them correctly. The only restriction is that all options must be added to a packet in numeric order.

Refer to RFC 7252, section 12.2 for more information.

Values:

enumerator `COAP_OPTION_IF_MATCH` = 1

If-Match.

enumerator `COAP_OPTION_URI_HOST` = 3

Uri-Host.

enumerator `COAP_OPTION_ETAG` = 4

ETag.

enumerator `COAP_OPTION_IF_NONE_MATCH` = 5

If-None-Match.

enumerator `COAP_OPTION_OBSERVE` = 6

Observe (RFC 7641)

enumerator `COAP_OPTION_URI_PORT` = 7

Uri-Port.

enumerator `COAP_OPTION_LOCATION_PATH` = 8

Location-Path.

enumerator COAP_OPTION_URI_PATH = 11
Uri-Path.

enumerator COAP_OPTION_CONTENT_FORMAT = 12
Content-Format.

enumerator COAP_OPTION_MAX_AGE = 14
Max-Age.

enumerator COAP_OPTION_URI_QUERY = 15
Uri-Query.

enumerator COAP_OPTION_ACCEPT = 17
Accept.

enumerator COAP_OPTION_LOCATION_QUERY = 20
Location-Query.

enumerator COAP_OPTION_BLOCK2 = 23
Block2 (RFC 7959)

enumerator COAP_OPTION_BLOCK1 = 27
Block1 (RFC 7959)

enumerator COAP_OPTION_SIZE2 = 28
Size2 (RFC 7959)

enumerator COAP_OPTION_PROXY_URI = 35
Proxy-Uri.

enumerator COAP_OPTION_PROXY_SCHEME = 39
Proxy-Scheme.

enumerator COAP_OPTION_SIZE1 = 60
Size1.

enumerator COAP_OPTION_ECHO = 252
Echo (RFC 9175)

enumerator COAP_OPTION_REQUEST_TAG = 292
Request-Tag (RFC 9175)

enum coap_method
Available request methods.

To be used when creating a request or a response.

Values:

enumerator COAP_METHOD_GET = 1
GET.

enumerator COAP_METHOD_POST = 2
POST.

enumerator COAP_METHOD_PUT = 3
PUT.

enumerator COAP_METHOD_DELETE = 4
DELETE.

enumerator COAP_METHOD_FETCH = 5
FETCH.

enumerator COAP_METHOD_PATCH = 6
PATCH.

enumerator COAP_METHOD_IPATCH = 7
IPATCH.

enum coap_msgtype
CoAP packets may be of one of these types.
Values:

enumerator COAP_TYPE_CON = 0
Confirmable message.
The packet is a request or response the destination end-point must acknowledge.

enumerator COAP_TYPE_NON_CON = 1
Non-confirmable message.
The packet is a request or response that doesn't require acknowledgements.

enumerator COAP_TYPE_ACK = 2
Acknowledge.
Response to a confirmable message.

enumerator COAP_TYPE_RESET = 3
Reset.
Rejecting a packet for any reason is done by sending a message of this type.

enum coap_response_code
Set of response codes available for a response packet.
To be used when creating a response.
Values:

enumerator COAP_RESPONSE_CODE_OK = ((2 « 5) | (0))
2.00 - OK

enumerator COAP_RESPONSE_CODE_CREATED = ((2 « 5) | (1))
2.01 - Created

enumerator COAP_RESPONSE_CODE_DELETED = ((2 « 5) | (2))
2.02 - Deleted

enumerator COAP_RESPONSE_CODE_VALID = ((2 « 5) | (3))
2.03 - Valid

enumerator COAP_RESPONSE_CODE_CHANGED = ((2 « 5) | (4))
2.04 - Changed

enumerator COAP_RESPONSE_CODE_CONTENT = ((2 « 5) | (5))
2.05 - Content

enumerator COAP_RESPONSE_CODE_CONTINUE = ((2 « 5) | (31))
2.31 - Continue

enumerator COAP_RESPONSE_CODE_BAD_REQUEST = ((4 « 5) | (0))
4.00 - Bad Request

enumerator COAP_RESPONSE_CODE_UNAUTHORIZED = ((4 « 5) | (1))
4.01 - Unauthorized

enumerator COAP_RESPONSE_CODE_BAD_OPTION = ((4 « 5) | (2))
4.02 - Bad Option

enumerator COAP_RESPONSE_CODE_FORBIDDEN = ((4 « 5) | (3))
4.03 - Forbidden

enumerator COAP_RESPONSE_CODE_NOT_FOUND = ((4 « 5) | (4))
4.04 - Not Found

enumerator COAP_RESPONSE_CODE_NOT_ALLOWED = ((4 « 5) | (5))
4.05 - Method Not Allowed

enumerator COAP_RESPONSE_CODE_NOT_ACCEPTABLE = ((4 « 5) | (6))
4.06 - Not Acceptable

enumerator COAP_RESPONSE_CODE_INCOMPLETE = ((4 « 5) | (8))
4.08 - Request Entity Incomplete

enumerator COAP_RESPONSE_CODE_CONFLICT = ((4 « 5) | (9))
4.12 - Precondition Failed

enumerator COAP_RESPONSE_CODE_PRECONDITION_FAILED = ((4 « 5) | (12))
4.12 - Precondition Failed

enumerator COAP_RESPONSE_CODE_REQUEST_TOO_LARGE = ((4 « 5) | (13))
4.13 - Request Entity Too Large

enumerator COAP_RESPONSE_CODE_UNSUPPORTED_CONTENT_FORMAT = ((4 « 5) | (15))
4.15 - Unsupported Content-Format

enumerator COAP_RESPONSE_CODE_UNPROCESSABLE_ENTITY = ((4 « 5) | (22))
4.22 - Unprocessable Entity

enumerator COAP_RESPONSE_CODE_TOO_MANY_REQUESTS = ((4 « 5) | (29))
4.29 - Too Many Requests

enumerator COAP_RESPONSE_CODE_INTERNAL_ERROR = ((5 « 5) | (0))
5.00 - Internal Server Error

enumerator COAP_RESPONSE_CODE_NOT_IMPLEMENTED = ((5 « 5) | (1))
5.01 - Not Implemented

enumerator COAP_RESPONSE_CODE_BAD_GATEWAY = ((5 « 5) | (2))
5.02 - Bad Gateway

enumerator COAP_RESPONSE_CODE_SERVICE_UNAVAILABLE = ((5 « 5) | (3))
5.03 - Service Unavailable

enumerator COAP_RESPONSE_CODE_GATEWAY_TIMEOUT = ((5 « 5) | (4))
5.04 - Gateway Timeout

enumerator COAP_RESPONSE_CODE_PROXYING_NOT_SUPPORTED = ((5 « 5) | (5))
5.05 - Proxying Not Supported

enum coap_content_format
Set of Content-Format option values for CoAP.

To be used when encoding or decoding a Content-Format option.

Values:

enumerator COAP_CONTENT_FORMAT_TEXT_PLAIN = 0
text/plain;charset=utf-8

enumerator COAP_CONTENT_FORMAT_APP_LINK_FORMAT = 40
application/link-format

enumerator COAP_CONTENT_FORMAT_APP_XML = 41
application/xml

enumerator COAP_CONTENT_FORMAT_APP_OCTET_STREAM = 42
application/octet-stream

enumerator COAP_CONTENT_FORMAT_APP_EXI = 47
application/exi

enumerator COAP_CONTENT_FORMAT_APP_JSON = 50
application/json

enumerator COAP_CONTENT_FORMAT_APP_JSON_PATCH_JSON = 51
application/json-patch+json

enumerator COAP_CONTENT_FORMAT_APP_MERGE_PATCH_JSON = 52
application/merge-patch+json

enumerator COAP_CONTENT_FORMAT_APP_CBOR = 60
application/cbor

enum coap_block_size
Represents the size of each block that will be transferred using block-wise transfers [RFC7959]:

Each entry maps directly to the value that is used in the wire.

https://tools.ietf.org/html/rfc7959

Values:

enumerator COAP_BLOCK_16
16-byte block size

enumerator COAP_BLOCK_32
32-byte block size

enumerator COAP_BLOCK_64
64-byte block size

enumerator COAP_BLOCK_128
128-byte block size

enumerator COAP_BLOCK_256
256-byte block size

enumerator COAP_BLOCK_512
512-byte block size

enumerator COAP_BLOCK_1024
1024-byte block size

Functions

uint8_t **coap_header_get_version**(const struct *coap_packet* *cpkt)

Returns the version present in a CoAP packet.

Parameters

- **cpkt** – CoAP packet representation

Returns
the CoAP version in packet

uint8_t **coap_header_get_type**(const struct *coap_packet* *cpkt)

Returns the type of the CoAP packet.

Parameters

- **cpkt** – CoAP packet representation

Returns
the type of the packet

uint8_t **coap_header_get_token**(const struct *coap_packet* *cpkt, uint8_t *token)

Returns the token (if any) in the CoAP packet.

Parameters

- **cpkt** – CoAP packet representation
- **token** – Where to store the token, must point to a buffer containing at least COAP_TOKEN_MAX_LEN bytes

Returns
Token length in the CoAP packet (0 - COAP_TOKEN_MAX_LEN).

uint8_t **coap_header_get_code**(const struct *coap_packet* *cpkt)

Returns the code of the CoAP packet.

Parameters

- **cpkt** – CoAP packet representation

Returns
the code present in the packet

int **coap_header_set_code**(const struct *coap_packet* *cpkt, uint8_t code)

Modifies the code of the CoAP packet.

Parameters

- **cpkt** – CoAP packet representation
- **code** – CoAP code

Returns
0 on success, -EINVAL on failure

uint16_t **coap_header_get_id**(const struct *coap_packet* *cpkt)

Returns the message id associated with the CoAP packet.

Parameters

- **cpkt** – CoAP packet representation

Returns
the message id present in the packet

const uint8_t *coap_packet_get_payload(const struct *coap_packet* *cpkt, uint16_t *len)

Returns the data pointer and length of the CoAP packet.

Parameters

- `cpkt` – CoAP packet representation
- `len` – Total length of CoAP payload

Returns

data pointer and length if payload exists NULL pointer and length set to 0 in case there is no payload

bool coap_uri_path_match(const char *const *path, struct *coap_option* *options, uint8_t opt_num)

Verify if CoAP URI path matches with provided options.

Parameters

- `path` – Null-terminated array of strings.
- `options` – Parsed options from *coap_packet_parse()*
- `opt_num` – Number of options

Returns

true if the CoAP URI path matches, false otherwise.

int coap_packet_parse(struct *coap_packet* *cpkt, uint8_t *data, uint16_t len, struct *coap_option* *options, uint8_t opt_num)

Parses the CoAP packet in data, validating it and initializing *cpkt*.

data must remain valid while *cpkt* is used.

Parameters

- `cpkt` – Packet to be initialized from received *data*.
- `data` – Data containing a CoAP packet, its *data* pointer is positioned on the start of the CoAP packet.
- `len` – Length of the data
- `options` – Parse options and cache its details.
- `opt_num` – Number of options

Return values

- `0` – in case of success.
- `-EINVAL` – in case of invalid input args.
- `-EBADMSG` – in case of malformed coap packet header.
- `-EILSEQ` – in case of malformed coap options.

int coap_packet_set_path(struct *coap_packet* *cpkt, const char *path)

Parses provided coap path (with/without query) or query and appends that as options to the *cpkt*.

Parameters

- `cpkt` – Packet to append path and query options for.
- `path` – Null-terminated string of coap path, query or both.

Return values

`0` – in case of success or negative in case of error.

int **coap_packet_init**(struct *coap_packet* *cpkt, uint8_t *data, uint16_t max_len, uint8_t ver, uint8_t type, uint8_t token_len, const uint8_t *token, uint8_t code, uint16_t id)

Creates a new CoAP Packet from input data.

Parameters

- **cpkt** – New packet to be initialized using the storage from *data*.
- **data** – Data that will contain a CoAP packet information
- **max_len** – Maximum allowable length of data
- **ver** – CoAP header version
- **type** – CoAP header type
- **token_len** – CoAP header token length
- **token** – CoAP header token
- **code** – CoAP header code
- **id** – CoAP header message id

Returns

0 in case of success or negative in case of error.

int **coap_ack_init**(struct *coap_packet* *cpkt, const struct *coap_packet* *req, uint8_t *data, uint16_t max_len, uint8_t code)

Create a new CoAP Acknowledgment message for given request.

This function works like *coap_packet_init*, filling CoAP header type, CoAP header token, and CoAP header message id fields according to acknowledgment rules.

Parameters

- **cpkt** – New packet to be initialized using the storage from *data*.
- **req** – CoAP request packet that is being acknowledged
- **data** – Data that will contain a CoAP packet information
- **max_len** – Maximum allowable length of data
- **code** – CoAP header code

Returns

0 in case of success or negative in case of error.

uint8_t ***coap_next_token**(void)

Returns a randomly generated array of 8 bytes, that can be used as a message's token.

Returns

a 8-byte pseudo-random token.

uint16_t **coap_next_id**(void)

Helper to generate message ids.

Returns

a new message id

int **coap_find_options**(const struct *coap_packet* *cpkt, uint16_t code, struct *coap_option* *options, uint16_t veclen)

Return the values associated with the option of value *code*.

Parameters

- **cpkt** – CoAP packet representation
- **code** – Option number to look for

- **options** – Array of *coap_option* where to store the value of the options found
- **veclen** – Number of elements in the options array

Returns
The number of options found in packet matching code, negative on error.

int **coap_packet_append_option**(struct *coap_packet* *cpkt, uint16_t code, const uint8_t *value, uint16_t len)

Appends an option to the packet.

Note: options can be added out of numeric order of their codes. But it's more efficient to add them in order.

Parameters

- **cpkt** – Packet to be updated
- **code** – Option code to add to the packet, see *coap_option_num*
- **value** – Pointer to the value of the option, will be copied to the packet
- **len** – Size of the data to be added

Returns
0 in case of success or negative in case of error.

int **coap_packet_remove_option**(struct *coap_packet* *cpkt, uint16_t code)

Remove an option from the packet.

Parameters

- **cpkt** – Packet to be updated
- **code** – Option code to remove from the packet, see *coap_option_num*

Returns
0 in case of success or negative in case of error.

unsigned int **coap_option_value_to_int**(const struct *coap_option* *option)

Converts an option to its integer representation.

Assumes that the number is encoded in the network byte order in the option.

Parameters

- **option** – Pointer to the option value, retrieved by *coap_find_options()*

Returns
The integer representation of the option

int **coap_append_option_int**(struct *coap_packet* *cpkt, uint16_t code, unsigned int val)

Appends an integer value option to the packet.

The option must be added in numeric order of their codes, and the least amount of bytes will be used to encode the value.

Parameters

- **cpkt** – Packet to be updated
- **code** – Option code to add to the packet, see *coap_option_num*
- **val** – Integer value to be added

Returns
0 in case of success or negative in case of error.

int `coap_packet_append_payload_marker`(struct *coap_packet* *cpkt)

Append payload marker to CoAP packet.

Parameters

- `cpkt` – Packet to append the payload marker (0xFF)

Returns
0 in case of success or negative in case of error.

int `coap_packet_append_payload`(struct *coap_packet* *cpkt, const uint8_t *payload, uint16_t payload_len)

Append payload to CoAP packet.

Parameters

- `cpkt` – Packet to append the payload
- `payload` – CoAP packet payload
- `payload_len` – CoAP packet payload len

Returns
0 in case of success or negative in case of error.

bool `coap_packet_is_request`(const struct *coap_packet* *cpkt)

Check if a CoAP packet is a CoAP request.

Parameters

- `cpkt` – Packet to be checked.

Returns
true if the packet is a request, false otherwise.

int `coap_handle_request_len`(struct *coap_packet* *cpkt, struct *coap_resource* *resources, size_t resources_len, struct *coap_option* *options, uint8_t opt_num, struct *sockaddr* *addr, *socklen_t* addr_len)

When a request is received, call the appropriate methods of the matching resources.

Parameters

- `cpkt` – Packet received
- `resources` – Array of known resources
- `resources_len` – Number of resources in the array
- `options` – Parsed options from *coap_packet_parse()*
- `opt_num` – Number of options
- `addr` – Peer address
- `addr_len` – Peer address length

Return values

- `>=` – 0 in case of success.
- `-ENOTSUP` – in case of invalid request code.
- `-EPERM` – in case resource handler is not implemented.
- `-ENOENT` – in case the resource is not found.

int `coap_handle_request`(struct *coap_packet* *cpkt, struct *coap_resource* *resources, struct *coap_option* *options, uint8_t opt_num, struct *sockaddr* *addr, *socklen_t* addr_len)

When a request is received, call the appropriate methods of the matching resources.

Parameters

- `cpkt` – Packet received
- `resources` – Array of known resources (terminated with empty resource)
- `options` – Parsed options from *coap_packet_parse()*
- `opt_num` – Number of options
- `addr` – Peer address
- `addr_len` – Peer address length

Return values

- `>=` – 0 in case of success.
- `-ENOTSUP` – in case of invalid request code.
- `-EPERM` – in case resource handler is not implemented.
- `-ENOENT` – in case the resource is not found.

static inline uint16_t **coap_block_size_to_bytes**(enum *coap_block_size* block_size)

Helper for converting the enumeration to the size expressed in bytes.

Parameters

- `block_size` – The block size to be converted

Returns

The size in bytes that the block_size represents

static inline enum *coap_block_size* **coap_bytes_to_block_size**(uint16_t bytes)

Helper for converting block size in bytes to enumeration.

NOTE: Only valid CoAP block sizes map correctly.

Parameters

- `bytes` – CoAP block size in bytes.

Returns

enum coap_block_size

int **coap_block_transfer_init**(struct *coap_block_context* *ctx, enum *coap_block_size* block_size, size_t total_size)

Initializes the context of a block-wise transfer.

Parameters

- `ctx` – The context to be initialized
- `block_size` – The size of the block
- `total_size` – The total size of the transfer, if known

Returns

0 in case of success or negative in case of error.

int **coap_append_descriptive_block_option**(struct *coap_packet* *cpkt, struct *coap_block_context* *ctx)

Append BLOCK1 or BLOCK2 option to the packet.

If the CoAP packet is a request then BLOCK1 is appended otherwise BLOCK2 is appended.

Parameters

- `cpkt` – Packet to be updated

- **ctx** – Block context from which to retrieve the information for the block option

Returns
0 in case of success or negative in case of error.

bool **coap_has_descriptive_block_option**(struct *coap_packet* *cpkt)

Check if a descriptive block option is set in the packet.

If the CoAP packet is a request then an available BLOCK1 option would be checked otherwise a BLOCK2 option would be checked.

Parameters

- **cpkt** – Packet to be checked.

Returns
true if the corresponding block option is set, false otherwise.

int **coap_remove_descriptive_block_option**(struct *coap_packet* *cpkt)

Remove BLOCK1 or BLOCK2 option from the packet.

If the CoAP packet is a request then BLOCK1 is removed otherwise BLOCK2 is removed.

Parameters

- **cpkt** – Packet to be updated.

Returns
0 in case of success or negative in case of error.

bool **coap_block_has_more**(struct *coap_packet* *cpkt)

Check if BLOCK1 or BLOCK2 option has more flag set.

Parameters

- **cpkt** – Packet to be checked.

Returns
true If more flag is set in BLOCK1 or BLOCK2

Returns
false If MORE flag is not set or BLOCK header not found.

int **coap_append_block1_option**(struct *coap_packet* *cpkt, struct *coap_block_context* *ctx)

Append BLOCK1 option to the packet.

Parameters

- **cpkt** – Packet to be updated
- **ctx** – Block context from which to retrieve the information for the Block1 option

Returns
0 in case of success or negative in case of error.

int **coap_append_block2_option**(struct *coap_packet* *cpkt, struct *coap_block_context* *ctx)

Append BLOCK2 option to the packet.

Parameters

- **cpkt** – Packet to be updated
- **ctx** – Block context from which to retrieve the information for the Block2 option

Returns
0 in case of success or negative in case of error.

int `coap_append_size1_option`(struct *coap_packet* *cpkt, struct *coap_block_context* *ctx)

Append SIZE1 option to the packet.

Parameters

- `cpkt` – Packet to be updated
- `ctx` – Block context from which to retrieve the information for the Size1 option

Returns

0 in case of success or negative in case of error.

int `coap_append_size2_option`(struct *coap_packet* *cpkt, struct *coap_block_context* *ctx)

Append SIZE2 option to the packet.

Parameters

- `cpkt` – Packet to be updated
- `ctx` – Block context from which to retrieve the information for the Size2 option

Returns

0 in case of success or negative in case of error.

int `coap_get_option_int`(const struct *coap_packet* *cpkt, uint16_t code)

Get the integer representation of a CoAP option.

Parameters

- `cpkt` – Packet to be inspected
- `code` – CoAP option code

Returns

Integer value >= 0 in case of success or negative in case of error.

int `coap_get_block1_option`(const struct *coap_packet* *cpkt, bool *has_more, uint8_t *block_number)

Get the block size, more flag and block number from the CoAP block1 option.

Parameters

- `cpkt` – Packet to be inspected
- `has_more` – Is set to the value of the more flag
- `block_number` – Is set to the number of the block

Returns

Integer value of the block size in case of success or negative in case of error.

int `coap_get_block2_option`(const struct *coap_packet* *cpkt, uint8_t *block_number)

Get values from CoAP block2 option.

Decode block number and block size from option. Ignore the has_more flag as it should always be zero on queries.

Parameters

- `cpkt` – Packet to be inspected
- `block_number` – Is set to the number of the block

Returns

Integer value of the block size in case of success or negative in case of error.

int **coap_update_from_block**(const struct *coap_packet* *cpkt, struct *coap_block_context* *ctx)

Retrieves BLOCK{1,2} and SIZE{1,2} from *cpkt* and updates *ctx* accordingly.

Parameters

- **cpkt** – Packet in which to look for block-wise transfers options
- **ctx** – Block context to be updated

Returns
0 in case of success or negative in case of error.

int **coap_next_block_for_option**(const struct *coap_packet* *cpkt, struct *coap_block_context* *ctx, enum *coap_option_num* option)

Updates *ctx* according to *option* set in *cpkt* so after this is called the current entry indicates the correct offset in the body of data being transferred.

Parameters

- **cpkt** – Packet in which to look for block-wise transfers options
- **ctx** – Block context to be updated
- **option** – Either COAP_OPTION_BLOCK1 or COAP_OPTION_BLOCK2

Returns
The offset in the block-wise transfer, 0 if the transfer has finished or a negative value in case of an error.

size_t **coap_next_block**(const struct *coap_packet* *cpkt, struct *coap_block_context* *ctx)

Updates *ctx* so after this is called the current entry indicates the correct offset in the body of data being transferred.

Parameters

- **cpkt** – Packet in which to look for block-wise transfers options
- **ctx** – Block context to be updated

Returns
The offset in the block-wise transfer, 0 if the transfer has finished.

void **coap_observer_init**(struct *coap_observer* *observer, const struct *coap_packet* *request, const struct *sockaddr* *addr)

Indicates that the remote device referenced by *addr*, with *request*, wants to observe a resource.

Parameters

- **observer** – Observer to be initialized
- **request** – Request on which the observer will be based
- **addr** – Address of the remote device

bool **coap_register_observer**(struct *coap_resource* *resource, struct *coap_observer* *observer)

After the observer is initialized, associate the observer with an resource.

Parameters

- **resource** – Resource to add an observer
- **observer** – Observer to be added

Returns
true if this is the first observer added to this resource.

bool **coap_remove_observer**(struct *coap_resource* *resource, struct *coap_observer* *observer)

Remove this observer from the list of registered observers of that resource.

Parameters

- **resource** – Resource in which to remove the observer
- **observer** – Observer to be removed

Returns
true if the observer was found and removed.

struct *coap_observer* ***coap_find_observer**(struct *coap_observer* *observers, size_t len, const struct *sockaddr* *addr, const uint8_t *token, uint8_t token_len)

Returns the observer that matches address *addr* and has token *token*.

Parameters

- **observers** – Pointer to the array of observers
- **len** – Size of the array of observers
- **addr** – Address of the endpoint observing a resource
- **token** – Pointer to the token
- **token_len** – Length of valid bytes in the token

Returns
A pointer to a observer if a match is found, NULL otherwise.

struct *coap_observer* ***coap_find_observer_by_addr**(struct *coap_observer* *observers, size_t len, const struct *sockaddr* *addr)

Returns the observer that matches address *addr*.

Note

The function *coap_find_observer()* should be preferred if both the observer's address and token are known.

Parameters

- **observers** – Pointer to the array of observers
- **len** – Size of the array of observers
- **addr** – Address of the endpoint observing a resource

Returns
A pointer to a observer if a match is found, NULL otherwise.

struct *coap_observer* ***coap_find_observer_by_token**(struct *coap_observer* *observers, size_t len, const uint8_t *token, uint8_t token_len)

Returns the observer that has token *token*.

Note

The function *coap_find_observer()* should be preferred if both the observer's address and token are known.

Parameters

- `observers` – Pointer to the array of observers
- `len` – Size of the array of observers
- `token` – Pointer to the token
- `token_len` – Length of valid bytes in the token

Returns
A pointer to a observer if a match is found, NULL otherwise.

struct *coap_observer* *`coap_observer_next_unused`(struct *coap_observer* *observers, size_t len)

Returns the next available observer representation.

Parameters

- `observers` – Pointer to the array of observers
- `len` – Size of the array of observers

Returns
A pointer to a observer if there's an available observer, NULL otherwise.

void `coap_reply_init`(struct *coap_reply* *reply, const struct *coap_packet* *request)

Indicates that a reply is expected for *request*.

Parameters

- `reply` – Reply structure to be initialized
- `request` – Request from which *reply* will be based

int `coap_pending_init`(struct *coap_pending* *pending, const struct *coap_packet* *request, const struct *sockaddr* *addr, const struct *coap_transmission_parameters* *params)

Initialize a pending request with a request.

The request's fields are copied into the pending struct, so *request* doesn't have to live for as long as the pending struct lives, but "data" that needs to live for at least that long.

Parameters

- `pending` – Structure representing the waiting for a confirmation message, initialized with data from *request*
- `request` – Message waiting for confirmation
- `addr` – Address to send the retransmission
- `params` – Pointer to the CoAP transmission parameters struct, or NULL to use default values

Returns
0 in case of success or negative in case of error.

struct *coap_pending* *`coap_pending_next_unused`(struct *coap_pending* *pendings, size_t len)

Returns the next available pending struct, that can be used to track the retransmission status of a request.

Parameters

- `pendings` – Pointer to the array of *coap_pending* structures
- `len` – Size of the array of *coap_pending* structures

Returns
pointer to a free *coap_pending* structure, NULL in case none could be found.

struct *coap_reply* ***coap_reply_next_unused**(struct *coap_reply* *replies, size_t len)

Returns the next available reply struct, so it can be used to track replies and notifications received.

Parameters

- **replies** – Pointer to the array of *coap_reply* structures
- **len** – Size of the array of *coap_reply* structures

Returns
pointer to a free *coap_reply* structure, NULL in case none could be found.

struct *coap_pending* ***coap_pending_received**(const struct *coap_packet* *response, struct *coap_pending* *pendings, size_t len)

After a response is received, returns if there is any matching pending request exits.

User has to clear all pending retransmissions related to that response by calling *coap_pending_clear()*.

Parameters

- **response** – The received response
- **pendings** – Pointer to the array of *coap_reply* structures
- **len** – Size of the array of *coap_reply* structures

Returns
pointer to the associated *coap_pending* structure, NULL in case none could be found.

struct *coap_reply* ***coap_response_received**(const struct *coap_packet* *response, const struct *sockaddr* *from, struct *coap_reply* *replies, size_t len)

After a response is received, call coap_reply_t handler registered in *coap_reply* structure.

Parameters

- **response** – A response received
- **from** – Address from which the response was received
- **replies** – Pointer to the array of *coap_reply* structures
- **len** – Size of the array of *coap_reply* structures

Returns
Pointer to the reply matching the packet received, NULL if none could be found.

struct *coap_pending* ***coap_pending_next_to_expire**(struct *coap_pending* *pendings, size_t len)

Returns the next pending about to expire, pending->timeout informs how many ms to next expiration.

Parameters

- **pendings** – Pointer to the array of *coap_pending* structures
- **len** – Size of the array of *coap_pending* structures

Returns
The next *coap_pending* to expire, NULL if none is about to expire.

bool **coap_pending_cycle**(struct *coap_pending* *pending)

After a request is sent, user may want to cycle the pending retransmission so the timeout is updated.

Parameters

- **pending** – Pending representation to have its timeout updated

Returns

false if this is the last retransmission.

void **coap_pending_clear**(struct *coap_pending* *pending)

Cancels the pending retransmission, so it again becomes available.

Parameters

- **pending** – Pending representation to be canceled

void **coap_pendings_clear**(struct *coap_pending* *pendings, size_t len)

Cancels all pending retransmissions, so they become available again.

Parameters

- **pendings** – Pointer to the array of *coap_pending* structures
- **len** – Size of the array of *coap_pending* structures

size_t **coap_pendings_count**(struct *coap_pending* *pendings, size_t len)

Count number of pending requests.

Parameters

- **len** – Number of elements in array.
- **pendings** – Array of pending requests.

Returns

count of elements where timeout is not zero.

void **coap_reply_clear**(struct *coap_reply* *reply)

Cancels awaiting for this reply, so it becomes available again.

User responsibility to free the memory associated with data.

Parameters

- **reply** – The reply to be canceled

void **coap_replies_clear**(struct *coap_reply* *replies, size_t len)

Cancels all replies, so they become available again.

Parameters

- **replies** – Pointer to the array of *coap_reply* structures
- **len** – Size of the array of *coap_reply* structures

int **coap_resource_notify**(struct *coap_resource* *resource)

Indicates that this resource was updated and that the *notify* callback should be called for every registered observer.

Parameters

- **resource** – Resource that was updated

Returns

0 in case of success or negative in case of error.

bool **coap_request_is_observe**(const struct *coap_packet* *request)

Returns if this request is enabling observing a resource.

Parameters

- **request** – Request to be checked

Returns

True if the request is enabling observing a resource, False otherwise

struct *coap_transmission_parameters* **coap_get_transmission_parameters**(void)

Get currently active CoAP transmission parameters.

Returns

CoAP transmission parameters structure.

void **coap_set_transmission_parameters**(const struct *coap_transmission_parameters* *params)

Set CoAP transmission parameters.

Parameters

- **params** – Pointer to the transmission parameters structure.

int **coap_well_known_core_get**(struct *coap_resource* *resource, const struct *coap_packet* *request, struct *coap_packet* *response, uint8_t *data, uint16_t data_len)

Build a CoAP response for a .well-known/core CoAP request.

Parameters

- **resource** – Array of known resources, terminated with an empty resource
- **request** – A pointer to the .well-known/core CoAP request
- **response** – A pointer to a CoAP response, will be initialized
- **data** – A data pointer to be used to build the CoAP response
- **data_len** – The maximum length of the data buffer

Returns

0 in case of success or negative in case of error.

int **coap_well_known_core_get_len**(struct *coap_resource* *resources, size_t resources_len, const struct *coap_packet* *request, struct *coap_packet* *response, uint8_t *data, uint16_t data_len)

Build a CoAP response for a .well-known/core CoAP request.

Parameters

- **resources** – Array of known resources
- **resources_len** – Number of resources in the array
- **request** – A pointer to the .well-known/core CoAP request
- **response** – A pointer to a CoAP response, will be initialized
- **data** – A data pointer to be used to build the CoAP response
- **data_len** – The maximum length of the data buffer

Returns

0 in case of success or negative in case of error.

struct `coap_resource`

#include <*coap.h*> Description of CoAP resource.

CoAP servers often want to register resources, so that clients can act on them, by fetching their state or requesting updates to them.

Public Members

coap_method_t `get`

Which function to be called for each CoAP method.

coap_notify_t `notify`

Notify function to call.

const char *const *`path`

Resource path.

void *`user_data`

User specific opaque data.

sys_slist_t `observers`

List of resource observers.

int `age`

Resource age.

struct `coap_observer`

#include <*coap.h*> Represents a remote device that is observing a local resource.

Public Members

sys_snode_t `list`

Observer list node.

struct *sockaddr* `addr`

Observer connection end point information.

uint8_t `token`[8]

Observer token.

uint8_t `tkl`

Extended token length.

struct `coap_packet`

#include <*coap.h*> Representation of a CoAP Packet.

Public Members

uint8_t *data
User allocated buffer.

uint16_t offset
CoAP lib maintains offset while adding data.

uint16_t max_len
Max CoAP packet data length.

uint8_t hdr_len
CoAP header length.

uint16_t opt_len
Total options length (delta + len + value)

uint16_t delta
Used for delta calculation in CoAP packet.

struct coap_option
#include <*coap.h*> Representation of a CoAP option.

Public Members

uint16_t delta
Option delta.

uint8_t len
Option length.

uint8_t value[12]
Option value.

struct coap_transmission_parameters
#include <*coap.h*> CoAP transmission parameters.

Public Members

uint32_t ack_timeout
Initial ACK timeout.

Value is used as a base value to retry pending CoAP packets.

uint16_t coap_backoff_percent
Set CoAP retry backoff factor.

A value of 200 means a factor of 2.0.

uint8_t **max_retransmission**
Maximum number of retransmissions.

struct **coap_pending**
#include <*coap.h*> Represents a request awaiting for an acknowledgment (ACK).

Public Members

struct *sockaddr* **addr**
Remote address.

int64_t **t0**
Time when the request was sent.

uint32_t **timeout**
Timeout in ms.

uint16_t **id**
Message id.

uint8_t ***data**
User allocated buffer.

uint16_t **len**
Length of the CoAP packet.

uint8_t **retries**
Number of times the request has been sent.

struct *coap_transmission_parameters* **params**
Transmission parameters.

struct **coap_reply**
#include <*coap.h*> Represents the handler for the reply of a request, it is also used when observing resources.

Public Members

coap_reply_t **reply**
CoAP reply callback.

void ***user_data**
User specific opaque data.

int **age**
Reply age.

uint16_t **id**
: Reply id.

uint8_t **token**[8]
: Reply token.

uint8_t **tkl**
: Extended token length.

struct **coap_block_context**
: *#include <coap.h>* Represents the current state of a block-wise transaction.

Public Members

size_t **total_size**
: Total size of the block-wise transaction.

size_t **current**
: Current size of the block-wise transaction.

enum *coap_block_size* **block_size**
: Block size.

struct **coap_core_metadata**
: *#include <coap_link_format.h>* In case you want to add attributes to the resources included in the 'well-known/core' "virtual" resource, the 'user_data' field should point to a valid *coap_core_metadata* structure.

Public Members

const char *const ***attributes**
: List of attributes to add.

void ***user_data**
: User specific data.

CoAP client

- *Overview*
- *Sample Usage*
- *API Reference*

Overview The CoAP client library allows application to send CoAP requests and parse CoAP responses. The library can be enabled with CONFIG_COAP_CLIENT Kconfig option. The application is notified about the response via a callback that is provided to the API in the request. The CoAP client handles the communication over sockets. As the CoAP client doesn't create socket it is using, the application is responsible for creating the socket. Plain UDP or DTLS sockets are supported.

Sample Usage The following is an example of a CoAP client initialization and request sending:

```
static struct coap_client;
struct coap_client_request req = { 0 };

coap_client_init(&client, NULL);

req.method = COAP_METHOD_GET;
req.confirmable = true;
req.path = "test";
req.fmt = COAP_CONTENT_FORMAT_TEXT_PLAIN;
req.cb = response_cb;
req.payload = NULL;
req.len = 0;

/* Sock is a file descriptor referencing a socket, address is the sockaddr struct for the
 * destination address of the request or NULL if the socket is already connected.
 */
ret = coap_client_req(&client, sock, &address, &req, -1);
```

Before any requests can be sent, the CoAP client needs to be initialized. After initialization, the application can send a CoAP request and wait for the response. Currently only one request can be sent for a single CoAP client at a time. There can be multiple CoAP clients.

The callback provided in the callback will be called in following cases:

- There is a response for the request
- The request failed for some reason

The callback contains a flag *last_block*, which indicates if there is more data to come in the response and means that the current response is part of a blockwise transfer. When the *last_block* is set to true, the response is finished and the client is ready for the next request after returning from the callback.

If the server responds to the request, the library provides the response to the application through the response callback registered in the request structure. As the response can be a blockwise transfer and the client calls the callback once per each block, the application should be to process all of the blocks to be able to process the response.

The following is an example of a very simple response handling function:

```
void response_cb(int16_t code, size_t offset, const uint8_t *payload, size_t len,
                 bool last_block, void *user_data)
{
    if (code >= 0) {
            LOG_INF("CoAP response from server %d", code);
            if (last_block) {
                    LOG_INF("Last packet received");
            }
    } else {
            LOG_ERR("Error in sending request %d", code);
    }
}
```

API Reference

group `coap_client`

CoAP client API.

Defines

`MAX_COAP_MSG_LEN`

Maximum size of a CoAP message.

Typedefs

typedef void (*`coap_client_response_cb_t`)(int16_t result_code, size_t offset, const uint8_t *payload, size_t len, bool last_block, void *user_data)

Callback for CoAP request.

This callback is called for responses to CoAP client requests. It is used to indicate errors, response codes from server or to deliver payload. Blockwise transfers cause this callback to be called sequentially with increasing payload offset and only partial content in buffer pointed by payload parameter.

Param result_code
Result code of the response. Negative if there was a failure in send. *coap_response_code* for positive.

Param offset
Payload offset from the beginning of a blockwise transfer.

Param payload
Buffer containing the payload from the response. NULL for empty payload.

Param len
Size of the payload.

Param last_block
Indicates the last block of the response.

Param user_data
User provided context.

Functions

int `coap_client_init`(struct coap_client *client, const char *info)

Initialize the CoAP client.

Parameters

- `client` – **[in]** Client instance.
- `info` – **[in]** Name for the receiving thread of the client. Setting this NULL will result as default name of “coap_client”.

Returns
int Zero on success, otherwise a negative error code.

int **coap_client_req**(struct coap_client *client, int sock, const struct *sockaddr* *addr, struct *coap_client_request* *req, struct *coap_transmission_parameters* *params)

Send CoAP request.

Operation is handled asynchronously using a background thread. If the socket isn't connected to a destination address, user must provide a destination address, otherwise the address should be set as NULL. Once the callback is called with last block set as true, socket can be closed or used for another query.

Parameters

- **client** – Client instance.
- **sock** – Open socket file descriptor.
- **addr** – the destination address of the request, NULL if socket is already connected.
- **req** – CoAP request structure
- **params** – Pointer to transmission parameters structure or NULL to use default values.

Returns
zero when operation started successfully or negative error code otherwise.

void **coap_client_cancel_requests**(struct coap_client *client)

Cancel all current requests.

This is intended for canceling long-running requests (e.g. GETs with the OBSERVE option set) which has gone stale for some reason.

Parameters

- **client** – Client instance.

struct **coap_client_request**

#include <*coap_client.h*> Representation of a CoAP client request.

Public Members

enum *coap_method* **method**

Method of the request.

bool **confirmable**

CoAP Confirmable/Non-confirmable message.

const char ***path**

Path of the requested resource.

enum *coap_content_format* **fmt**

Content format to be used.

uint8_t ***payload**

User allocated buffer for send request.

size_t **len**
: Length of the payload.

coap_client_response_cb_t **cb**
: Callback when response received.

struct *coap_client_option* ***options**
: Extra options to be added to request.

uint8_t **num_options**
: Number of extra options.

void ***user_data**
: User provided context.

struct **coap_client_option**
#include <coap_client.h> Representation of extra options for the CoAP client request.

Public Members

uint16_t **code**
: Option code.

uint8_t **len**
: Option len.

uint8_t **value**[12]
: Buffer for the length.

CoAP server

- *Overview*
- *Setup*
- *Sample Usage*
- *Observable resources*
- *CoAP Events*
- *CoRE Link Format*
- *API Reference*

Overview Zephyr comes with a batteries-included CoAP server, which uses services to listen for CoAP requests. The CoAP services handle communication over sockets and pass requests to registered CoAP resources.

Setup Some configuration is required to make sure services can be started using the CoAP server. The `CONFIG_COAP_SERVER` option should be enabled in your project:

Listing 1: `prj.conf`

```
CONFIG_COAP_SERVER=y
```

All services are added to a predefined linker section and all resources for each service also get their respective linker sections. If you would have a service `my_service` it has to be prefixed with `coap_resource_` and added to a linker file:

Listing 2: `sections-ram.ld`

```
#include <zephyr/linker/iterable_sections.h>

ITERABLE_SECTION_RAM(coap_resource_my_service, 4)
```

Add this linker file to your application using CMake:

Listing 3: `CMakeLists.txt`

```
zephyr_linker_sources(DATA_SECTIONS sections-ram.ld)
```

You can now define your service as part of the application:

```
#include <zephyr/net/coap_service.h>

static const uint16_t my_service_port = 5683;

COAP_SERVICE_DEFINE(my_service, "0.0.0.0", &my_service_port, COAP_SERVICE_AUTOSTART);
```

Note

Services defined with the `COAP_SERVICE_AUTOSTART` flag will be started together with the CoAP server thread. Services can be manually started and stopped with `coap_service_start` and `coap_service_stop` respectively.

Sample Usage The following is an example of a CoAP resource registered with our service:

```
#include <zephyr/net/coap_service.h>

static int my_get(struct coap_resource *resource, struct coap_packet *request,
                  struct sockaddr *addr, socklen_t addr_len)
{
    static const char *msg = "Hello, world!";
    uint8_t data[CONFIG_COAP_SERVER_MESSAGE_SIZE];
    struct coap_packet response;
    uint16_t id;
    uint8_t token[COAP_TOKEN_MAX_LEN];
    uint8_t tkl, type;

    type = coap_header_get_type(request);
    id = coap_header_get_id(request);
    tkl = coap_header_get_token(request, token);

    /* Determine response type */
    type = (type == COAP_TYPE_CON) ? COAP_TYPE_ACK : COAP_TYPE_NON_CON;

    coap_packet_init(&response, data, sizeof(data), COAP_VERSION_1, type, tkl, token,
```

(continues on next page)

(continued from previous page)

```
                    COAP_RESPONSE_CODE_CONTENT, id);

    /* Set content format */
    coap_append_option_int(&response, COAP_OPTION_CONTENT_FORMAT,
                           COAP_CONTENT_FORMAT_TEXT_PLAIN);

    /* Append payload */
    coap_packet_append_payload_marker(&response);
    coap_packet_append_payload(&response, (uint8_t *)msg, sizeof(msg));

    /* Send to response back to the client */
    return coap_resource_send(resource, &response, addr, addr_len, NULL);
}

static int my_put(struct coap_resource *resource, struct coap_packet *request,
                  struct sockaddr *addr, socklen_t addr_len)
{
    /* ... Handle the incoming request ... */

    /* Return a CoAP response code as a shortcut for an empty ACK message */
    return COAP_RESPONSE_CODE_CHANGED;
}

static const char * const my_resource_path[] = { "test", NULL };
COAP_RESOURCE_DEFINE(my_resource, my_service, {
    .path = my_resource_path,
    .get = my_get,
    .put = my_put,
});
```

Note

As demonstrated in the example above, a CoAP resource handler can return response codes to let the server respond with an empty ACK response.

Observable resources The CoAP server provides logic for parsing observe requests and stores these using the runtime data of CoAP services. An example using a temperature sensor can look like:

```
#include <zephyr/kernel.h>
#include <zephyr/drivers/sensor.h>
#include <zephyr/net/coap_service.h>

static void notify_observers(struct k_work *work);
K_WORK_DELAYABLE_DEFINE(temp_work, notify_observers);

static int send_temperature(struct coap_resource *resource,
                            const struct sockaddr *addr, socklen_t addr_len,
                            uint16_t age, uint16_t id, const uint8_t *token, uint8_t tkl,
                            bool is_response)
{
    const struct device *dev = DEVICE_DT_GET(DT_ALIAS(ambient_temp0));
    uint8_t data[CONFIG_COAP_SERVER_MESSAGE_SIZE];
    struct coap_packet response;
    char payload[14];
    struct sensor_value value;
    double temp;
    uint8_t type;
```

(continues on next page)

(continued from previous page)

```
    /* Determine response type */
    type = is_response ? COAP_TYPE_ACK : COAP_TYPE_CON;

    if (!is_response) {
        id = coap_next_id();
    }

    coap_packet_init(&response, data, sizeof(data), COAP_VERSION_1, type, tkl, token,
                     COAP_RESPONSE_CODE_CONTENT, id);

    if (age >= 2U) {
        coap_append_option_int(&response, COAP_OPTION_OBSERVE, age);
    }

    /* Set content format */
    coap_append_option_int(&response, COAP_OPTION_CONTENT_FORMAT,
                           COAP_CONTENT_FORMAT_TEXT_PLAIN);

    /* Get the sensor date */
    sensor_sample_fetch_chan(dev, SENSOR_CHAN_AMBIENT_TEMP);
    sensor_channel_get(dev, SENSOR_CHAN_AMBIENT_TEMP, &value);
    temp = sensor_value_to_double(&value);

    snprintk(payload, sizeof(payload), "%0.2f°C", temp);

    /* Append payload */
    coap_packet_append_payload_marker(&response);
    coap_packet_append_payload(&response, (uint8_t *)payload, strlen(payload));

    return coap_resource_send(resource, &response, addr, addr_len, NULL);
}

static int temp_get(struct coap_resource *resource, struct coap_packet *request,
                    struct sockaddr *addr, socklen_t addr_len)
{
    uint8_t token[COAP_TOKEN_MAX_LEN];
    uint16_t id;
    uint8_t tkl;
    int r;

    /* Let the CoAP server parse the request and add/remove observers if needed */
    r = coap_resource_parse_observe(resource, request, addr);

    id = coap_header_get_id(request);
    tkl = coap_header_get_token(request, token);

    return send_temperature(resource, addr, addr_len, r == 0 ? resource->age : 0,
                            id, token, tkl, true);
}

static void temp_notify(struct coap_resource *resource, struct coap_observer *observer)
{
    send_temperature(resource, &observer->addr, sizeof(observer->addr), resource->age, 0,
                     observer->token, observer->tkl, false);
}

static const char * const temp_resource_path[] = { "sensors", "temp1", NULL };
COAP_RESOURCE_DEFINE(temp_resource, my_service, {
    .path = temp_resource_path,
    .get = temp_get,
```

(continues on next page)

(continued from previous page)

```
    .notify = temp_notify,
});

static void notify_observers(struct k_work *work)
{
    if (sys_slist_is_empty(&temp_resource.observers)) {
        return;
    }

    coap_resource_notify(&temp_resource);
    k_work_reschedule(&temp_work, K_SECONDS(1));
}
```

CoAP Events By enabling `CONFIG_NET_MGMT_EVENT` the user can register for CoAP events. The following example simply prints when an event occurs.

```
#include <zephyr/sys/printk.h>
#include <zephyr/net/coap_mgmt.h>
#include <zephyr/net/coap_service.h>

#define COAP_EVENTS_SET (NET_EVENT_COAP_OBSERVER_ADDED | NET_EVENT_COAP_OBSERVER_REMOVED | \
                         NET_EVENT_COAP_SERVICE_STARTED | NET_EVENT_COAP_SERVICE_STOPPED)

void coap_event_handler(uint32_t mgmt_event, struct net_if *iface,
                        void *info, size_t info_length, void *user_data)
{
    switch (mgmt_event) {
    case NET_EVENT_COAP_OBSERVER_ADDED:
        printk("CoAP observer added");
        break;
    case NET_EVENT_COAP_OBSERVER_REMOVED:
        printk("CoAP observer removed");
        break;
    case NET_EVENT_COAP_SERVICE_STARTED:
        if (info != NULL && info_length == sizeof(struct net_event_coap_service)) {
            struct net_event_coap_service *net_event = info;

            printk("CoAP service %s started", net_event->service->name);
        } else {
            printk("CoAP service started");
        }
        break;
    case NET_EVENT_COAP_SERVICE_STOPPED:
        if (info != NULL && info_length == sizeof(struct net_event_coap_service)) {
            struct net_event_coap_service *net_event = info;

            printk("CoAP service %s stopped", net_event->service->name);
        } else {
            printk("CoAP service stopped");
        }
        break;
    }
}

NET_MGMT_REGISTER_EVENT_HANDLER(coap_events, COAP_EVENTS_SET, coap_event_handler, NULL);
```

CoRE Link Format The `CONFIG_COAP_SERVER_WELL_KNOWN_CORE` option enables handling the `.well-known/core` GET requests by the server. This allows clients to get a list of hypermedia links to other resources hosted in that server.

Related code samples

CoAP service
Use the CoAP server subsystem to register CoAP resources.

API Reference

group `coap_service`

CoAP Service API.

CoAP Service configuration flags

`COAP_SERVICE_AUTOSTART`

Start the service on boot.

Defines

`COAP_RESOURCE_DEFINE`(_name, _service, ...)

Define a static CoAP resource owned by the service named `_service` .

```
static const struct gpio_dt_spec led = GPIO_DT_SPEC_GET(DT_ALIAS(led0), gpios);

static int led_put(struct coap_resource *resource, struct coap_packet *request,
                   struct sockaddr *addr, socklen_t addr_len)
{
        const uint8_t *payload;
        uint16_t payload_len;

        payload = coap_packet_get_payload(request, &payload_len);
        if (payload_len != 1) {
                return COAP_RESPONSE_CODE_BAD_REQUEST;
        }

        if (gpio_pin_set_dt(&led, payload[0]) < 0) {
                return COAP_RESPONSE_CODE_INTERNAL_ERROR;
        }

        return COAP_RESPONSE_CODE_CHANGED;
}

COAP_RESOURCE_DEFINE(my_resource, my_service, {
        .put = led_put,
});
```

Note

The handlers registered with the resource can return a CoAP response code to reply with an acknowledge without any payload, nothing is sent if the return value is 0 or negative. As seen in the example.

Parameters

- **_name** – Name of the resource.
- **_service** – Name of the associated service.

COAP_SERVICE_DEFINE(_name, _host, _port, _flags)

Define a CoAP service with static resources.

See also

COAP_SERVICE_FLAGS.

Note

The _host parameter can be NULL. If not, it is used to specify an IP address either in IPv4 or IPv6 format a fully-qualified hostname or a virtual host, otherwise the any address is used.

Note

The _port parameter must be non-NULL. It points to a location that specifies the port number to use for the service. If the specified port number is zero, then an ephemeral port number will be used and the actual port number assigned will be written back to memory. For ephemeral port numbers, the memory pointed to by _port must be writeable.

Parameters

- **_name** – Name of the service.
- **_host** – IP address or hostname associated with the service.
- **_port** – **[inout]** Pointer to port associated with the service.
- **_flags** – Configuration flags

COAP_SERVICE_COUNT(_dst)

Count the number of CoAP services.

Parameters

- **_dst** – **[out]** Pointer to location where result is written.

COAP_SERVICE_RESOURCE_COUNT(_service)

Count CoAP service static resources.

Parameters

- **_service** – Pointer to a service.

COAP_SERVICE_HAS_RESOURCE(_service, _resource)

Check if service has the specified resource.

Parameters

- **_service** – Pointer to a service.
- **_resource** – Pointer to a resource.

COAP_SERVICE_FOREACH(_it)

Iterate over all CoAP services.

Parameters

- **_it** – Name of iterator (of type *CoAP service API*)

COAP_RESOURCE_FOREACH(_service, _it)

Iterate over static CoAP resources associated with a given _service.

> **Note**
>
> This macro requires that _service is defined with *COAP_SERVICE_DEFINE*.

Parameters

- **_service** – Name of CoAP service
- **_it** – Name of iterator (of type *coap_resource*)

COAP_SERVICE_FOREACH_RESOURCE(_service, _it)

Iterate over all static resources associated with _service .

> **Note**
>
> This macro is suitable for a _service defined with *COAP_SERVICE_DEFINE*.

Parameters

- **_service** – Pointer to COAP service
- **_it** – Name of iterator (of type *coap_resource*)

Functions

int coap_service_start(const struct coap_service *service)

Start the provided service .

> **Note**
>
> This function is suitable for a service defined with *COAP_SERVICE_DEFINE*.

Parameters

- **service** – Pointer to CoAP service

Return values

- **0** – in case of success.
- **-EALREADY** – in case of an already running service.
- **-ENOTSUP** – in case the server has no valid host and port configuration.

int **coap_service_stop**(const struct coap_service *service)

Stop the provided `service` .

> **Note**
>
> This function is suitable for a `service` defined with *COAP_SERVICE_DEFINE*.

Parameters

- `service` – Pointer to CoAP service

Return values

- `0` – in case of success.
- `-EALREADY` – in case the service isn't running.

int **coap_service_is_running**(const struct coap_service *service)

Query the provided `service` running state.

> **Note**
>
> This function is suitable for a `service` defined with *COAP_SERVICE_DEFINE*.

Parameters

- `service` – Pointer to CoAP service

Return values

- `1` – if the service is running
- `0` – if the service is stopped
- `negative` – in case of an error.

int **coap_service_send**(const struct coap_service *service, const struct *coap_packet* *cpkt, const struct *sockaddr* *addr, *socklen_t* addr_len, const struct *coap_transmission_parameters* *params)

Send a CoAP message from the provided `service` .

> **Note**
>
> This function is suitable for a `service` defined with *COAP_SERVICE_DEFINE*.

Parameters

- `service` – Pointer to CoAP service
- `cpkt` – CoAP Packet to send
- `addr` – Peer address
- `addr_len` – Peer address length
- `params` – Pointer to transmission parameters structure or NULL to use default values.

Returns

0 in case of success or negative in case of error.

int `coap_resource_send`(const struct *coap_resource* *resource, const struct *coap_packet* *cpkt, const struct *sockaddr* *addr, *socklen_t* addr_len, const struct *coap_transmission_parameters* *params)

Send a CoAP message from the provided `resource` .

> **Note**
>
> This function is suitable for a `resource` defined with *COAP_RESOURCE_DEFINE*.

Parameters

- `resource` – Pointer to CoAP resource
- `cpkt` – CoAP Packet to send
- `addr` – Peer address
- `addr_len` – Peer address length
- `params` – Pointer to transmission parameters structure or NULL to use default values.

Returns

0 in case of success or negative in case of error.

int `coap_resource_parse_observe`(struct *coap_resource* *resource, const struct *coap_packet* *request, const struct *sockaddr* *addr)

Parse a CoAP observe request for the provided `resource` .

If the observe option value is equal to 0, an observer will be added, if the value is equal to 1, an existing observer will be removed.

> **Note**
>
> This function is suitable for a `resource` defined with *COAP_RESOURCE_DEFINE*.

Parameters

- `resource` – Pointer to CoAP resource
- `request` – CoAP request to parse
- `addr` – Peer address

Returns

the observe option value in case of success or negative in case of error.

int `coap_resource_remove_observer_by_addr`(struct *coap_resource* *resource, const struct *sockaddr* *addr)

Lookup an observer by address and remove it from the `resource` .

> **Note**
>
> This function is suitable for a `resource` defined with *COAP_RESOURCE_DEFINE*.

Parameters

- `resource` – Pointer to CoAP resource

- **addr** – Peer address

Returns
0 in case of success or negative in case of error.

int **coap_resource_remove_observer_by_token**(struct *coap_resource* *resource, const uint8_t *token, uint8_t token_len)

Lookup an observer by token and remove it from the `resource` .

> **Note**
> This function is suitable for a `resource` defined with *COAP_RESOURCE_DEFINE*.

Parameters

- **resource** – Pointer to CoAP resource
- **token** – Pointer to the token
- **token_len** – Length of valid bytes in the token

Returns
0 in case of success or negative in case of error.

group **coap_mgmt**

CoAP Manager Events.

Defines

NET_EVENT_COAP_SERVICE_STARTED

coap_mgmt event raised when a service has started

NET_EVENT_COAP_SERVICE_STOPPED

coap_mgmt event raised when a service has stopped

NET_EVENT_COAP_OBSERVER_ADDED

coap_mgmt event raised when an observer has been added to a resource

NET_EVENT_COAP_OBSERVER_REMOVED

coap_mgmt event raised when an observer has been removed from a resource

struct **net_event_coap_service**

#include <coap_mgmt.h> CoAP Service event structure.

Public Members

const struct coap_service ***service**

The CoAP service for which the event is emitted.

struct **net_event_coap_observer**

#include <coap_mgmt.h> CoAP Observer event structure.

Public Members

struct *coap_resource* ***resource**

The CoAP resource for which the event is emitted.

struct *coap_observer* ***observer**

The observer that is added/removed.

HTTP Client

- *Overview*
- *Sample Usage*
- *API Reference*

Overview The HTTP client library allows you to send HTTP requests and parse HTTP responses. The library communicates over the sockets API but it does not create sockets on its own. It can be enabled with `CONFIG_HTTP_CLIENT` Kconfig option.

The application must be responsible for creating a socket and passing it to the library. Therefore, depending on the application's needs, the library can communicate over either a plain TCP socket (HTTP) or a TLS socket (HTTPS).

Sample Usage The API of the HTTP client library has a single function.

The following is an example of a request structure created correctly:

```
struct http_request req = { 0 };
static uint8_t recv_buf[512];

req.method = HTTP_GET;
req.url = "/";
req.host = "localhost";
req.protocol = "HTTP/1.1";
req.response = response_cb;
req.recv_buf = recv_buf;
req.recv_buf_len = sizeof(recv_buf);

/* sock is a file descriptor referencing a socket that has been connected
 * to the HTTP server.
 */
ret = http_client_req(sock, &req, 5000, NULL);
```

If the server responds to the request, the library provides the response to the application through the response callback registered in the request structure. As the library can provide the response in chunks, the application must be able to process these.

Together with the structure containing the response data, the callback function also provides information about whether the library expects to receive more data.

The following is an example of a very simple response handling function:

```
static void response_cb(struct http_response *rsp,
                        enum http_final_call final_data,
                        void *user_data)
{
    if (final_data == HTTP_DATA_MORE) {
        LOG_INF("Partial data received (%zd bytes)", rsp->data_len);
    } else if (final_data == HTTP_DATA_FINAL) {
        LOG_INF("All the data received (%zd bytes)", rsp->data_len);
    }

    LOG_INF("Response status %s", rsp->http_status);
}
```

See HTTP client sample application for more information about the library usage.

Related code samples

HTTP Client
Implement an HTTP(S) client that issues a variety of HTTP requests.

TagoIO HTTP Post
Send random temperature values to TagoIO IoT Cloud Platform using HTTP.

API Reference

group `http_client`

HTTP client API.

Since
2.1

Version
0.2.0

Typedefs

typedef int (***http_payload_cb_t**)(int sock, struct *http_request* *req, void *user_data)

Callback used when data needs to be sent to the server.

Param sock
Socket id of the connection

Param req
HTTP request information

Param user_data
User specified data specified in *http_client_req()*

Return
>=0 amount of data sent, in this case *http_client_req()* should continue sending data, <0 if *http_client_req()* should return the error code to the caller.

typedef int (***http_header_cb_t**)(int sock, struct *http_request* *req, void *user_data)

Callback can be used if application wants to construct additional HTTP headers when the HTTP request is sent.

Usage of this is optional.

Param sock
Socket id of the connection

Param req
HTTP request information

Param user_data
User specified data specified in *http_client_req()*

Return
>=0 amount of data sent, in this case *http_client_req()* should continue sending data, <0 if *http_client_req()* should return the error code to the caller.

typedef void (*`http_response_cb_t`)(struct *http_response* *rsp, enum *http_final_call* final_data, void *user_data)

Callback used when data is received from the server.

Param rsp
HTTP response information

Param final_data
Does this data buffer contain all the data or is there still more data to come.

Param user_data
User specified data specified in *http_client_req()*

Enums

enum `http_final_call`

Is there more data to come.

Values:

enumerator `HTTP_DATA_MORE` = 0

More data will come.

enumerator `HTTP_DATA_FINAL` = 1

End of data.

Functions

int `http_client_req`(int sock, struct *http_request* *req, int32_t timeout, void *user_data)

Do a HTTP request.

The callback is called when data is received from the HTTP server. The caller must have created a connection to the server before calling this function so *connect()* call must have be done successfully for the socket.

Parameters

- `sock` – Socket id of the connection.
- `req` – HTTP request information

- **timeout** – Max timeout to wait for the data. The timeout value cannot be 0 as there would be no time to receive the data. The timeout value is in milliseconds.
- **user_data** – User specified data that is passed to the callback.

Returns
<0 if error, >=0 amount of data sent to the server

struct **http_response**

#include <*client.h*> HTTP response from the server.

Public Members

const struct http_parser_settings ***http_cb**

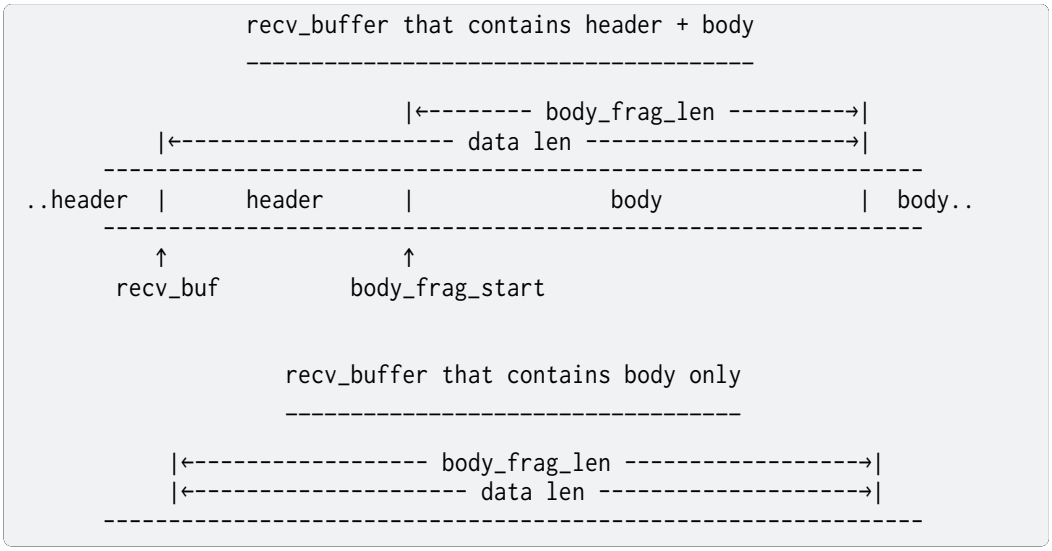
HTTP parser settings for the application usage.

http_response_cb_t **cb**

User provided HTTP response callback which is called when a response is received to a sent HTTP request.

uint8_t ***body_frag_start**

Start address of the body fragment contained in the recv_buf.



size_t **body_frag_len**

Length of the body fragment contained in the recv_buf.

uint8_t ***recv_buf**

Where the response is stored, this is to be provided by the user.

size_t **recv_buf_len**

Response buffer maximum length.

size_t **data_len**

Length of the data in the result buf.

If the value is larger than recv_buf_len, then it means that the data is truncated and could not be fully copied into recv_buf. This can only happen if the user did not set the response callback. If the callback is set, then the HTTP client API will call response callback many times so that all the data is delivered to the user. Will be zero in the event of a null response.

size_t **content_length**

HTTP Content-Length field value.

Will be set to zero in the event of a null response.

size_t **processed**

Amount of data given to the response callback so far, including the current data given to the callback.

This should be equal to the content_length field once the entire body has been received. Will be zero if a null response is given.

char **http_status**[HTTP_STATUS_STR_SIZE]

See https://tools.ietf.org/html/rfc7230#section-3.1.2 for more information.

The status-code element is a 3-digit integer code

The reason-phrase element exists for the sole purpose of providing a textual description associated with the numeric status code. A client SHOULD ignore the reason-phrase content.

Will be blank if a null HTTP response is given.

uint16_t **http_status_code**

Numeric HTTP status code which corresponds to the textual description.

Set to zero if null response is given. Otherwise, will be a 3-digit integer code if valid HTTP response is given.

uint8_t **cl_present**

Is Content-Length field present.

uint8_t **body_found**

Is message body found.

uint8_t **message_complete**

Is HTTP message parsing complete.

struct **http_client_internal_data**

#include <*client.h*> HTTP client internal data that the application should not touch.

Public Members

struct http_parser **parser**

HTTP parser context.

struct http_parser_settings **parser_settings**

HTTP parser settings.

struct *http_response* **response**

HTTP response specific data (filled by *http_client_req()* when data is received)

void ***user_data**

User data.

int **sock**

HTTP socket.

struct **http_request**

#include <*client.h*> HTTP client request.

This contains all the data that is needed when doing a HTTP request.

Public Members

struct *http_client_internal_data* **internal**

HTTP client request internal data.

enum http_method **method**

The HTTP method: GET, HEAD, OPTIONS, POST, ...

http_response_cb_t **response**

User supplied callback function to call when response is received.

const struct http_parser_settings ***http_cb**

User supplied list of HTTP callback functions if the calling application wants to know the parsing status or the HTTP fields.

This is optional and normally not needed.

uint8_t ***recv_buf**

User supplied buffer where received data is stored.

size_t **recv_buf_len**

Length of the user supplied receive buffer.

const char ***url**

The URL for this request, for example: /index.html.

const char ***protocol**

The HTTP protocol, for example “HTTP/1.1”.

const char ****header_fields**

The HTTP header fields (application specific) The Content-Type may be specified here or in the next field.

Depending on your application, the Content-Type may vary, however some header fields may remain constant through the application’s life cycle. This is a NULL terminated list of header fields.

const char *content_type_value

The value of the Content-Type header field, may be NULL.

const char *host

Hostname to be used in the request.

const char *port

Port number to be used in the request.

http_payload_cb_t payload_cb

User supplied callback function to call when payload needs to be sent.

This can be NULL in which case the payload field in *http_request* is used. The idea of this payload callback is to allow user to send more data that is practical to store in allocated memory.

const char *payload

Payload, may be NULL.

size_t payload_len

Payload length is used to calculate Content-Length.

Set to 0 for chunked transfers.

http_header_cb_t optional_headers_cb

User supplied callback function to call when optional headers need to be sent.

This can be NULL, in which case the optional_headers field in *http_request* is used. The idea of this optional_headers callback is to allow user to send more HTTP header data that is practical to store in allocated memory.

const char **optional_headers

A NULL terminated list of any optional headers that should be added to the HTTP request.

May be NULL. If the optional_headers_cb is specified, then this field is ignored. Note that there are two similar fields that contain headers, the header_fields above and this optional_headers. This is done like this to support Websocket use case where Websocket will use header_fields variable and any optional application specific headers will be placed into this field.

HTTP Server

- *Overview*
- *Server Setup*
- *Sample Usage*
  - *Services*
  - *Static resources*
  - *Dynamic resources*

  - *Websocket resources*
- *API Reference*

Overview Zephyr provides an HTTP server library, which allows to register HTTP services and HTTP resources associated with those services. The server creates a listening socket for every registered service, and handles incoming client connections. It's possible to communicate over a plain TCP socket (HTTP) or a TLS socket (HTTPS). Both, HTTP/1.1 (RFC 2616) and HTTP/2 (RFC 9113) protocol versions are supported.

The server operation is generally transparent for the application, running in a background thread. The application can control the server activity with respective API functions.

Certain resource types (for example dynamic resource) provide resource-specific application callbacks, allowing the server to interact with the application (for instance provide resource content, or process request payload).

Currently, the following resource types are supported:

- Static resources - content defined compile-time, cannot be modified at runtime (*HTTP_RESOURCE_TYPE_STATIC*).
- Dynamic resources - content provided at runtime by respective application callback (*HTTP_RESOURCE_TYPE_DYNAMIC*).
- Websocket resources - allowing to establish Websocket connections with the server (*HTTP_RESOURCE_TYPE_WEBSOCKET*).

Zephyr provides a sample demonstrating HTTP(s) server operation and various resource types usage. See sockets-http-server for more information.

Server Setup A few prerequisites are needed in order to enable HTTP server functionality in the application.

First of all, the HTTP server has to be enabled in applications configuration file with `CONFIG_HTTP_SERVER` Kconfig option:

Listing 4: `prj.conf`

```
CONFIG_HTTP_SERVER=y
```

All HTTP services and HTTP resources are placed in a dedicated linker section. The linker section for services is predefined locally, however the application is responsible for defining linker sections for resources associated with respective services. Linker section names for resources should be prefixed with `http_resource_desc_`, appended with the service name.

Linker sections for resources should be defined in a linker file. For example, for a service named `my_service`, the linker section shall be defined as follows:

Listing 5: `sections-rom.ld`

```
#include <zephyr/linker/iterable_sections.h>

ITERABLE_SECTION_ROM(http_resource_desc_my_service, Z_LINK_ITERABLE_SUBALIGN)
```

Finally, the linker file and linker section have to be added to your application using CMake:

Listing 6: CMakeLists.txt

```
zephyr_linker_sources(SECTIONS sections-rom.ld)
zephyr_linker_section(NAME http_resource_desc_my_service
                      KVMA RAM_REGION GROUP RODATA_REGION
                      SUBALIGN Z_LINK_ITERABLE_SUBALIGN)
```

Note

You need to define a separate linker section for each HTTP service registered in the system.

Sample Usage

Services The application needs to define an HTTP service (or multiple services), with the same name as used for the linker section with *HTTP_SERVICE_DEFINE* macro:

```
#include <zephyr/net/http/service.h>

static uint16_t http_service_port = 80;

HTTP_SERVICE_DEFINE(my_service, "0.0.0.0", &http_service_port, 1, 10, NULL);
```

Alternatively, an HTTPS service can be defined with *HTTPS_SERVICE_DEFINE*:

```
#include <zephyr/net/http/service.h>
#include <zephyr/net/tls_credentials.h>

#define HTTP_SERVER_CERTIFICATE_TAG 1

static uint16_t https_service_port = 443;
static const sec_tag_t sec_tag_list[] = {
    HTTP_SERVER_CERTIFICATE_TAG,
};

HTTPS_SERVICE_DEFINE(my_service, "0.0.0.0", &https_service_port, 1, 10,
                     NULL, sec_tag_list, sizeof(sec_tag_list));
```

Note

HTTPS services rely on TLS credentials being registered in the system. See *TLS credentials subsystem* for information on how to configure TLS credentials in the system.

Once HTTP(s) service is defined, resources can be registered for it with *HTTP_RESOURCE_DEFINE* macro.

Application can enable resource wildcard support by enabling CONFIG_HTTP_SERVER_RESOURCE_WILDCARD option. When this option is set, then it is possible to match several incoming HTTP requests with just one resource handler. The fnmatch() POSIX API function is used to match the pattern in the URL paths.

Example:

```
HTTP_RESOURCE_DEFINE(my_resource, my_service, "/foo*", &resource_detail);
```

This would match all URLs that start with a string foo. See POSIX.2 chapter 2.13 for pattern matching syntax description.

Static resources Static resource content is defined build-time and is immutable. The following example shows how gzip compressed webpage can be defined as a static resource in the application:

```
static const uint8_t index_html_gz[] = {
    #include "index.html.gz.inc"
};

struct http_resource_detail_static index_html_gz_resource_detail = {
    .common = {
        .type = HTTP_RESOURCE_TYPE_STATIC,
        .bitmask_of_supported_http_methods = BIT(HTTP_GET),
        .content_encoding = "gzip",
    },
    .static_data = index_html_gz,
    .static_data_len = sizeof(index_html_gz),
};

HTTP_RESOURCE_DEFINE(index_html_gz_resource, my_service, "/",
                     &index_html_gz_resource_detail);
```

The resource content and content encoding is application specific. For the above example, a gzip compressed webpage can be generated during build, by adding the following code to the application's `CMakeLists.txt` file:

Listing 7: `CMakeLists.txt`

```
set(gen_dir ${ZEPHYR_BINARY_DIR}/include/generated/)
set(source_file_index src/index.html)
generate_inc_file_for_target(app ${source_file_index} ${gen_dir}/index.html.gz.inc --gzip)
```

where `src/index.html` is the location of the webpage to be compressed.

Dynamic resources For dynamic resource, a resource callback is registered to exchange data between the server and the application. The application defines a resource buffer used to pass the request payload data from the server, and to provide response payload to the server. The following example code shows how to register a dynamic resource with a simple resource handler, which echoes received data back to the client:

```
static uint8_t recv_buffer[1024];

static int dyn_handler(struct http_client_ctx *client,
                       enum http_data_status status, uint8_t *buffer,
                       size_t len, void *user_data)
{
#define MAX_TEMP_PRINT_LEN 32
    static char print_str[MAX_TEMP_PRINT_LEN];
    enum http_method method = client->method;
    static size_t processed;

    __ASSERT_NO_MSG(buffer != NULL);

    if (status == HTTP_SERVER_DATA_ABORTED) {
        LOG_DBG("Transaction aborted after %zd bytes.", processed);
        processed = 0;
        return 0;
    }

    processed += len;

    snprintf(print_str, sizeof(print_str), "%s received (%zd bytes)",
```

(continues on next page)

(continued from previous page)

```
            http_method_str(method), len);
    LOG_HEXDUMP_DBG(buffer, len, print_str);

    if (status == HTTP_SERVER_DATA_FINAL) {
        LOG_DBG("All data received (%zd bytes).", processed);
        processed = 0;
    }

    /* This will echo data back to client as the buffer and recv_buffer
     * point to same area.
     */
    return len;
}

struct http_resource_detail_dynamic dyn_resource_detail = {
    .common = {
        .type = HTTP_RESOURCE_TYPE_DYNAMIC,
        .bitmask_of_supported_http_methods =
            BIT(HTTP_GET) | BIT(HTTP_POST),
    },
    .cb = dyn_handler,
    .data_buffer = recv_buffer,
    .data_buffer_len = sizeof(recv_buffer),
    .user_data = NULL,
};

HTTP_RESOURCE_DEFINE(dyn_resource, my_service, "/dynamic",
                     &dyn_resource_detail);
```

The resource callback may be called multiple times for a single request, hence the application should be able to keep track of the received data progress.

The `status` field informs the application about the progress in passing request payload from the server to the application. As long as the status reports *HTTP_SERVER_DATA_MORE*, the application should expect more data to be provided in a consecutive callback calls. Once all request payload has been passed to the application, the server reports *HTTP_SERVER_DATA_FINAL* status. In case of communication errors during request processing (for example client closed the connection before complete payload has been received), the server reports *HTTP_SERVER_DATA_ABORTED*. Either of the two events indicate that the application shall reset any progress recorded for the resource, and await a new request to come. The server guarantees that the resource can only be accessed by single client at a time.

The resource callback returns the number of bytes to be replied in the response payload to the server (provided in the resource data buffer). In case there is no more data to be included in the response, the callback should return 0.

The server will call the resource callback until it provided all request data to the application, and the application reports there is no more data to include in the reply.

Websocket resources Websocket resources register an application callback, which is called when a Websocket connection upgrade takes place. The callback is provided with a socket descriptor corresponding to the underlying TCP/TLS connection. Once called, the application takes full control over the socket, i. e. is responsible to release it when done.

```
static int ws_socket;
static uint8_t ws_recv_buffer[1024];

int ws_setup(int sock, void *user_data)
{
    ws_socket = sock;
```

(continues on next page)

(continued from previous page)

```
    return 0;
}

struct http_resource_detail_websocket ws_resource_detail = {
    .common = {
        .type = HTTP_RESOURCE_TYPE_WEBSOCKET,
        /* We need HTTP/1.1 Get method for upgrading */
        .bitmask_of_supported_http_methods = BIT(HTTP_GET),
    },
    .cb = ws_setup,
    .data_buffer = ws_recv_buffer,
    .data_buffer_len = sizeof(ws_recv_buffer),
    .user_data = NULL, /* Fill this for any user specific data */
};

HTTP_RESOURCE_DEFINE(ws_resource, my_service, "/", &ws_resource_detail);
```

The above minimalistic example shows how to register a Websocket resource with a simple callback, used only to store the socket descriptor provided. Further processing of the Websocket connection is application-specific, hence outside of scope of this guide. See sockets-http-server for an example Websocket-based echo service implementation.

Related code samples

HTTP Server
Implement an HTTP(s) Server demonstrating various resource types.

API Reference

group `http_service`

Defines

`HTTP_RESOURCE_DEFINE`(_name, _service, _resource, _detail)

Define a static HTTP resource.

A static HTTP resource is one that is known prior to system initialization. In contrast, dynamic resources may be discovered upon system initialization. Dynamic resources may also be inserted, or removed by events originating internally or externally to the system at runtime.

Note

The `_resource` is the URL without the associated protocol, host, or URL parameters. E.g. the resource for `#param1=value1` would be `/bar/baz.html`. It is often referred to as the "path" of the URL. Every `(service, resource)` pair should be unique. The `_resource` must be non-NULL.

Parameters

- **`_name`** – Name of the resource.
- **`_service`** – Name of the associated service.

- **_resource** – Pathname-like string identifying the resource.
- **_detail** – Implementation-specific detail associated with the resource.

HTTP_SERVICE_DEFINE_EMPTY(_name, _host, _port, _concurrent, _backlog, _detail)

Define an HTTP service without static resources.

> **Note**
>
> The _host parameter must be non-NULL. It is used to specify an IP address either in IPv4 or IPv6 format a fully-qualified hostname or a virtual host.

> **Note**
>
> The _port parameter must be non-NULL. It points to a location that specifies the port number to use for the service. If the specified port number is zero, then an ephemeral port number will be used and the actual port number assigned will be written back to memory. For ephemeral port numbers, the memory pointed to by _port must be writeable.

Parameters

- **_name** – Name of the service.
- **_host** – IP address or hostname associated with the service.
- **_port** – **[inout]** Pointer to port associated with the service.
- **_concurrent** – Maximum number of concurrent clients.
- **_backlog** – Maximum number queued connections.
- **_detail** – Implementation-specific detail associated with the service.

HTTPS_SERVICE_DEFINE_EMPTY(_name, _host, _port, _concurrent, _backlog, _detail, _sec_tag_list, _sec_tag_list_size)

Define an HTTPS service without static resources.

> **Note**
>
> The _host parameter must be non-NULL. It is used to specify an IP address either in IPv4 or IPv6 format a fully-qualified hostname or a virtual host.

> **Note**
>
> The _port parameter must be non-NULL. It points to a location that specifies the port number to use for the service. If the specified port number is zero, then an ephemeral port number will be used and the actual port number assigned will be written back to memory. For ephemeral port numbers, the memory pointed to by _port must be writeable.

Parameters

- **_name** – Name of the service.
- **_host** – IP address or hostname associated with the service.

- **_port** – **[inout]** Pointer to port associated with the service.
- **_concurrent** – Maximum number of concurrent clients.
- **_backlog** – Maximum number queued connections.
- **_detail** – Implementation-specific detail associated with the service.
- **_sec_tag_list** – TLS security tag list used to setup a HTTPS socket.
- **_sec_tag_list_size** – TLS security tag list size used to setup a HTTPS socket.

HTTP_SERVICE_DEFINE(_name, _host, _port, _concurrent, _backlog, _detail)

Define an HTTP service with static resources.

Note

The _host parameter must be non-NULL. It is used to specify an IP address either in IPv4 or IPv6 format a fully-qualified hostname or a virtual host.

Note

The _port parameter must be non-NULL. It points to a location that specifies the port number to use for the service. If the specified port number is zero, then an ephemeral port number will be used and the actual port number assigned will be written back to memory. For ephemeral port numbers, the memory pointed to by _port must be writeable.

Parameters

- **_name** – Name of the service.
- **_host** – IP address or hostname associated with the service.
- **_port** – **[inout]** Pointer to port associated with the service.
- **_concurrent** – Maximum number of concurrent clients.
- **_backlog** – Maximum number queued connections.
- **_detail** – Implementation-specific detail associated with the service.

HTTPS_SERVICE_DEFINE(_name, _host, _port, _concurrent, _backlog, _detail, _sec_tag_list, _sec_tag_list_size)

Define an HTTPS service with static resources.

Note

The _host parameter must be non-NULL. It is used to specify an IP address either in IPv4 or IPv6 format a fully-qualified hostname or a virtual host.

Note

The _port parameter must be non-NULL. It points to a location that specifies the port number to use for the service. If the specified port number is zero, then an ephemeral port number will be used and the actual port number assigned will be

written back to memory. For ephemeral port numbers, the memory pointed to by `_port` must be writeable.

Parameters

- **_name** – Name of the service.
- **_host** – IP address or hostname associated with the service.
- **_port** – **[inout]** Pointer to port associated with the service.
- **_concurrent** – Maximum number of concurrent clients.
- **_backlog** – Maximum number queued connections.
- **_detail** – Implementation-specific detail associated with the service.
- **_sec_tag_list** – TLS security tag list used to setup a HTTPS socket.
- **_sec_tag_list_size** – TLS security tag list size used to setup a HTTPS socket.

`HTTP_SERVICE_COUNT`(_dst)

Count the number of HTTP services.

Parameters

- **_dst** – **[out]** Pointer to location where result is written.

`HTTP_SERVICE_RESOURCE_COUNT`(_service)

Count HTTP service static resources.

Parameters

- **_service** – Pointer to a service.

`HTTP_SERVICE_FOREACH`(_it)

Iterate over all HTTP services.

Parameters

- **_it** – Name of http_service_desc iterator

`HTTP_RESOURCE_FOREACH`(_service, _it)

Iterate over static HTTP resources associated with a given `_service`.

Note

This macro requires that `_service` is defined with *HTTP_SERVICE_DEFINE*.

Parameters

- **_service** – Name of HTTP service
- **_it** – Name of iterator (of type *http_resource_desc*)

`HTTP_SERVICE_FOREACH_RESOURCE`(_service, _it)

Iterate over all static resources associated with `_service` .

Note

This macro is suitable for a `_service` defined with either *HTTP_SERVICE_DEFINE* or *HTTP_SERVICE_DEFINE_EMPTY*.

Parameters

- `_service` – Pointer to HTTP service
- `_it` – Name of iterator (of type *http_resource_desc*)

struct `http_resource_desc`

#include <service.h> HTTP resource description.

Public Members

const char *`resource`

Resource name.

void *`detail`

Detail associated with this resource.

Related code samples

HTTP Server
Implement an HTTP(s) Server demonstrating various resource types.

group `http_server`

Typedefs

typedef int (*`http_resource_dynamic_cb_t`)(struct *http_client_ctx* *client, enum *http_data_status* status, uint8_t *data_buffer, size_t data_len, void *user_data)

Callback used when data is received.

Data to be sent to client can be specified.

Param client
HTTP context information for this client connection.

Param status
HTTP data status, indicate whether more data is expected or not.

Param data_buffer
Data received.

Param data_len
Amount of data received.

Param user_data
User specified data.

Return
>0 amount of data to be sent to client, let server to call this function again when new data is received. 0 nothing to sent to client, close the connection <0 error, close the connection.

typedef int (*`http_resource_websocket_cb_t`)(int ws_socket, void *user_data)

Callback used when a Websocket connection is setup.

The application will need to handle all functionality related to the connection like reading and writing websocket data, and closing the connection.

Param ws_socket
A socket for the Websocket data.

Param user_data
User specified data.

Return
0 Accepting the connection, HTTP server library will no longer handle data to/from the socket and it is application responsibility to send and receive data to/from the supplied socket. <0 error, close the connection.

Enums

enum `http_resource_type`

HTTP server resource type.

Values:

enumerator `HTTP_RESOURCE_TYPE_STATIC`

Static resource, cannot be modified on runtime.

enumerator `HTTP_RESOURCE_TYPE_DYNAMIC`

Dynamic resource, server interacts with the application via registered *http_resource_dynamic_cb_t*.

enumerator `HTTP_RESOURCE_TYPE_WEBSOCKET`

Websocket resource, application takes control over Websocket connection after and upgrade.

enum `http_data_status`

Indicates the status of the currently processed piece of data.

Values:

enumerator `HTTP_SERVER_DATA_ABORTED` = -1

Transaction aborted, data incomplete.

enumerator `HTTP_SERVER_DATA_MORE` = 0

Transaction incomplete, more data expected.

enumerator `HTTP_SERVER_DATA_FINAL` = 1

Final data fragment in current transaction.

Functions

int **http_server_start**(void)

Start the HTTP2 server.

The server runs in a background thread. Once started, the server will create a server socket for all HTTP services registered in the system and accept connections from clients (see *HTTP_SERVICE_DEFINE*).

int **http_server_stop**(void)

Stop the HTTP2 server.

All server sockets are closed and the server thread is suspended.

struct **http_resource_detail**

#include <*server.h*> Representation of a server resource, common for all resource types.

Public Members

uint32_t **bitmask_of_supported_http_methods**

Bitmask of supported HTTP methods (http_method).

enum *http_resource_type* **type**

Resource type.

int **path_len**

Length of the URL path.

const char ***content_encoding**

Content encoding of the resource.

const char ***content_type**

Content type of the resource.

struct **http_resource_detail_static**

#include <*server.h*> Representation of a static server resource.

Public Members

struct *http_resource_detail* **common**

Common resource details.

const void ***static_data**

Content of the static resource.

size_t **static_data_len**

Size of the static resource.

struct **http_resource_detail_dynamic**

#include <*server.h*> Representation of a dynamic server resource.

Public Members

struct *http_resource_detail* **common**

Common resource details.

http_resource_dynamic_cb_t **cb**

Resource callback used by the server to interact with the application.

uint8_t ***data_buffer**

Data buffer used to exchanged data between server and the, application.

size_t **data_buffer_len**

Length of the data in the data buffer.

struct *http_client_ctx* ***holder**

A pointer to the client currently processing resource, used to prevent concurrent access to the resource from multiple clients.

void ***user_data**

A pointer to the user data registered by the application.

struct **http_resource_detail_websocket**

#include <server.h> Representation of a websocket server resource.

Public Members

struct *http_resource_detail* **common**

Common resource details.

int **ws_sock**

Websocket socket value.

http_resource_websocket_cb_t **cb**

Resource callback used by the server to interact with the application.

uint8_t ***data_buffer**

Data buffer used to exchanged data between server and the, application.

size_t **data_buffer_len**

Length of the data in the data buffer.

void ***user_data**

A pointer to the user data registered by the application.

struct **http2_stream_ctx**

#include <server.h> HTTP/2 stream representation.

Public Members

int **stream_id**
Stream identifier.

enum http2_stream_state **stream_state**
Stream state.

int **window_size**
Stream-level window size.

bool **headers_sent**
Flag indicating that headers were sent in the reply.

bool **end_stream_sent**
Flag indicating that END_STREAM flag was sent.

struct **http2_frame**
#include <*server.h*> HTTP/2 frame representation.

Public Members

uint32_t **length**
Frame payload length.

uint32_t **stream_identifier**
Stream ID the frame belongs to.

uint8_t **type**
Frame type.

uint8_t **flags**
Frame flags.

uint8_t **padding_len**
Frame padding length.

struct **http_client_ctx**
#include <*server.h*> Representation of an HTTP client connected to the server.

Public Members

int **fd**
Socket descriptor associated with the server.

unsigned char **buffer**[HTTP_SERVER_CLIENT_BUFFER_SIZE]
Client data buffer.

unsigned char *cursor
: Cursor indicating currently processed byte.

size_t data_len
: Data left to process in the buffer.

int window_size
: Connection-level window size.

enum http_server_state server_state
: Server state for the associated client.

struct *http2_frame* current_frame
: Currently processed HTTP/2 frame.

struct *http_resource_detail* *current_detail
: Currently processed resource detail.

struct *http2_stream_ctx* *current_stream
: Currently processed stream.

struct http_hpack_header_buf header_field
: HTTP/2 header parser context.

struct *http2_stream_ctx* streams[HTTP_SERVER_MAX_STREAMS]
: HTTP/2 streams context.

struct http_parser_settings parser_settings
: HTTP/1 parser configuration.

struct http_parser parser
: HTTP/1 parser context.

unsigned char url_buffer[HTTP_SERVER_MAX_URL_LENGTH]
: Request URL.

unsigned char content_type[HTTP_SERVER_MAX_CONTENT_TYPE_LEN]
: Request content type.

unsigned char header_buffer[HTTP_SERVER_MAX_HEADER_LEN]
: Temp buffer for currently processed header (HTTP/1 only).

size_t content_len
: Request content length.

enum http_method method
: Request method.

enum http1_parser_state **parser_state**
HTTP/1 parser state.

int **http1_frag_data_len**
Length of the payload length in the currently processed request fragment (HTTP/1 only).

struct *k_work_delayable* **inactivity_timer**
Client inactivity timer.

The client connection is closed by the server when it expires.

bool **preface_sent**
Flag indicating that HTTP2 preface was sent.

bool **http1_headers_sent**
Flag indicating that HTTP1 headers were sent.

bool **has_upgrade_header**
Flag indicating that upgrade header was present in the request.

bool **http2_upgrade**
Flag indicating HTTP/2 upgrade takes place.

bool **websocket_upgrade**
Flag indicating Websocket upgrade takes place.

bool **websocket_sec_key_next**
Flag indicating Websocket key is being processed.

bool **expect_continuation**
The next frame on the stream is expectd to be a continuation frame.

Lightweight M2M (LWM2M)

- *Overview*
- *Example LwM2M object and resources: Device*
- *Sample usage*
- *LwM2M security modes*
- *Multi-thread usage*
- *Support for time series data*
  - *Enabling and configuring*
  - *Read and Write operations*
  - *Limitations*
- *LwM2M engine and application events*

- *Configuring lifetime and activity period*
- *LwM2M shell*
- *API Reference*

Overview Lightweight Machine to Machine (LwM2M) is an application layer protocol designed with device management, data reporting and device actuation in mind. Based on CoAP/UDP, LwM2M is a standard defined by the Open Mobile Alliance and suitable for constrained devices by its use of CoAP packet-size optimization and a simple, stateless flow that supports a REST API.

One of the key differences between LwM2M and CoAP is that an LwM2M client initiates the connection to an LwM2M server. The server can then use the REST API to manage various interfaces with the client.

LwM2M uses a simple resource model with the core set of objects and resources defined in the specification.

The LwM2M library can be enabled with `CONFIG_LWM2M` Kconfig option.

Example LwM2M object and resources: Device *Object definition*

Object ID	Name	Instance	Mandatory
3	Device	Single	Mandatory

Resource definitions

`* R=Read, W=Write, E=Execute`

ID	Name	OP*	Instance	Mandatory	Type
0	Manufacturer	R	Single	Optional	String
1	Model	R	Single	Optional	String
2	Serial number	R	Single	Optional	String
3	Firmware version	R	Single	Optional	String
4	Reboot	E	Single	Mandatory	
5	Factory Reset	E	Single	Optional	
6	Available Power Sources	R	Multiple	Optional	Integer 0-7
7	Power Source Voltage (mV)	R	Multiple	Optional	Integer
8	Power Source Current (mA)	R	Multiple	Optional	Integer
9	Battery Level %	R	Single	Optional	Integer
10	Memory Free (Kb)	R	Single	Optional	Integer
11	Error Code	R	Multiple	Optional	Integer 0-8
12	Reset Error	E	Single	Optional	
13	Current Time	RW	Single	Optional	Time
14	UTC Offset	RW	Single	Optional	String
15	Timezone	RW	Single	Optional	String
16	Supported Binding	R	Single	Mandatory	String
17	Device Type	R	Single	Optional	String
18	Hardware Version	R	Single	Optional	String
19	Software Version	R	Single	Optional	String
20	Battery Status	R	Single	Optional	Integer 0-6
21	Memory Total (Kb)	R	Single	Optional	Integer
22	ExtDevInfo	R	Multiple	Optional	ObjLnk

The server could query the `Manufacturer` resource for `Device` object instance 0 (the default and only instance) by sending a `READ 3/0/0` operation to the client.

The full list of registered objects and resource IDs can be found in the LwM2M registry.

Zephyr's LwM2M library lives in the subsys/net/lib/lwm2m, with a client sample in samples/net/lwm2m_client. For more information about the provided sample see: lwm2m-client. The sample can be configured to use normal unsecure network sockets or sockets secured via DTLS.

The Zephyr LwM2M library implements the following items:

- engine to process networking events and core functions
- RD client which performs BOOTSTRAP and REGISTRATION functions
- SenML CBOR, SenML JSON, CBOR, TLV, JSON, and plain text formatting functions
- LwM2M Technical Specification Enabler objects such as Security, Server, Device, Firmware Update, etc.
- Extended IPSO objects such as Light Control, Temperature Sensor, and Timer

By default, the library implements LwM2M specification 1.0.2 and can be set to LwM2M specification 1.1.1 with a Kconfig option.

For more information about LwM2M visit OMA Specworks LwM2M.

Sample usage To use the LwM2M library, start by creating an LwM2M client context *lwm2m_ctx* structure:

```
/* LwM2M client context */
static struct lwm2m_ctx client;
```

Create callback functions for LwM2M resource executions:

```
static int device_reboot_cb(uint16_t obj_inst_id, uint8_t *args,
                            uint16_t args_len)
{
        LOG_INF("Device rebooting.");
        LOG_PANIC();
        sys_reboot(0);
        return 0; /* won't reach this */
}
```

The LwM2M RD client can send events back to the sample. To receive those events, setup a callback function:

```
static void rd_client_event(struct lwm2m_ctx *client,
                            enum lwm2m_rd_client_event client_event)
{
        switch (client_event) {

        case LWM2M_RD_CLIENT_EVENT_NONE:
                /* do nothing */
                break;

        case LWM2M_RD_CLIENT_EVENT_BOOTSTRAP_REG_FAILURE:
                LOG_DBG("Bootstrap registration failure!");
                break;

        case LWM2M_RD_CLIENT_EVENT_BOOTSTRAP_REG_COMPLETE:
                LOG_DBG("Bootstrap registration complete");
                break;
```

(continues on next page)

(continued from previous page)

```
        case LWM2M_RD_CLIENT_EVENT_BOOTSTRAP_TRANSFER_COMPLETE:
                LOG_DBG("Bootstrap transfer complete");
                break;

        case LWM2M_RD_CLIENT_EVENT_REGISTRATION_FAILURE:
                LOG_DBG("Registration failure!");
                break;

        case LWM2M_RD_CLIENT_EVENT_REGISTRATION_COMPLETE:
                LOG_DBG("Registration complete");
                break;

        case LWM2M_RD_CLIENT_EVENT_REG_TIMEOUT:
                LOG_DBG("Registration timeout!");
                break;

        case LWM2M_RD_CLIENT_EVENT_REG_UPDATE_COMPLETE:
                LOG_DBG("Registration update complete");
                break;

        case LWM2M_RD_CLIENT_EVENT_DEREGISTER_FAILURE:
                LOG_DBG("Deregister failure!");
                break;

        case LWM2M_RD_CLIENT_EVENT_DISCONNECT:
                LOG_DBG("Disconnected");
                break;

        case LWM2M_RD_CLIENT_EVENT_REG_UPDATE:
                LOG_DBG("Registration update");
                break;

        case LWM2M_RD_CLIENT_EVENT_DEREGISTER:
                LOG_DBG("Deregistration client");
                break;

        case LWM2M_RD_CLIENT_EVENT_SERVER_DISABLED:
                LOG_DBG("LwM2M server disabled");
          break;
        }
}
```

Next we assign `Security` resource values to let the client know where and how to connect as well as set the `Manufacturer` and `Reboot` resources in the `Device` object with some data and the callback we defined above:

```
/*
 * Server URL of default Security object = 0/0/0
 * Use leshan.eclipse.org server IP (5.39.83.206) for connection
 */
lwm2m_set_string(&LWM2M_OBJ(0, 0, 0), "coap://5.39.83.206");

/*
 * Security Mode of default Security object = 0/0/2
 * 3 = NoSec mode (no security beware!)
 */
lwm2m_set_u8(&LWM2M_OBJ(0, 0, 2), 3);

#define CLIENT_MANUFACTURER "Zephyr Manufacturer"
```

(continues on next page)

(continued from previous page)

```
/*
 * Manufacturer resource of Device object = 3/0/0
 * We use lwm2m_set_res_data() function to set a pointer to the
 * CLIENT_MANUFACTURER string.
 * Note the LWM2M_RES_DATA_FLAG_RO flag which stops the engine from
 * trying to assign a new value to the buffer.
 */
lwm2m_set_res_data(&LWM2M_OBJ(3, 0, 0), CLIENT_MANUFACTURER,
                   sizeof(CLIENT_MANUFACTURER),
                   LWM2M_RES_DATA_FLAG_RO);

/* Reboot resource of Device object = 3/0/4 */
lwm2m_register_exec_callback(&LWM2M_OBJ(3, 0, 4), device_reboot_cb);
```

Lastly, we start the LwM2M RD client (which in turn starts the LwM2M engine). The second parameter of *lwm2m_rd_client_start()* is the client endpoint name. This is important as it needs to be unique per LwM2M server:

```
(void)memset(&client, 0x0, sizeof(client));
lwm2m_rd_client_start(&client, "unique-endpoint-name", 0, rd_client_event);
```

LwM2M security modes The Zephyr LwM2M library can be used either without security or use DTLS to secure the communication channel. When using DTLS with the LwM2M engine, PSK (Pre-Shared Key) and X.509 certificates are the security modes that can be used to secure the communication. The engine uses LwM2M Security object (Id 0) to read the stored credentials and feed keys from the security object into the TLS credential subsystem, see *secure sockets documentation*. Enable the `CONFIG_LWM2M_DTLS_SUPPORT` Kconfig option to use the security.

Depending on the selected mode, the security object must contain following data:

PSK
: Security Mode (Resource ID 2) set to zero (Pre-Shared Key mode). Identity (Resource ID 3) contains PSK ID in binary form. Secret key (Resource ID 5) contains the PSK key in binary form. If the key or identity is provided as a hex string, it must be converted to binary before storing into the security object.

X509
: When X509 certificates are used, set Security Mode (ID 2) to `2` (Certificate mode). Identity (ID 3) is used to store the client certificate and Secret key (ID 5) must have a private key associated with the certificate. Server Public Key resource (ID 4) must contain a server certificate or CA certificate used to sign the certificate chain. If the `CONFIG_MBEDTLS_PEM_CERTIFICATE_FORMAT` Kconfig option is enabled, certificates and private key can be entered in PEM format. Otherwise, they must be in binary DER format.

NoSec
: When no security is used, set Security Mode (Resource ID 2) to `3` (NoSec).

In all modes, Server URI resource (ID 0) must contain the full URI for the target server. When DNS names are used, the DNS resolver must be enabled.

When DTLS is used, following options are recommended to reduce DTLS handshake traffic when connection is re-established:

- `CONFIG_LWM2M_DTLS_CID` enables DTLS Connection Identifier support. When server supports it, this completely removes the handshake when device resumes operation after long idle period. Greatly helps when NAT mappings have timed out.
- `CONFIG_LWM2M_TLS_SESSION_CACHING` uses session cache when before falling back to full DTLS handshake. Reduces few packets from handshake, when session is still cached on server side. Most significant effect is to avoid full registration.

LwM2M stack provides callbacks in the *lwm2m_ctx* structure. They are used to feed keys from the LwM2M security object into the TLS credential subsystem. By default, these callbacks can be left as NULL pointers, in which case default callbacks are used. When an external TLS stack, or non-default socket options are required, you can overwrite the *lwm2m_ctx.load_credentials()* or *lwm2m_ctx.set_socketoptions()* callbacks.

An example of setting up the security object for PSK mode:

```
/* "000102030405060708090a0b0c0d0e0f" */
static unsigned char client_psk[] = {
        0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
        0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f
};

static const char client_identity[] = "Client_identity";

lwm2m_set_string(&LWM2M_OBJ(LWM2M_OBJECT_SECURITY_ID, 0, 0), "coaps://lwm2m.example.com");
lwm2m_set_u8(&LWM2M_OBJ(LWM2M_OBJECT_SECURITY_ID, 0, 2), LWM2M_SECURITY_PSK);
/* Set the client identity as a string, but this could be binary as well */
lwm2m_set_string(&LWM2M_OBJ(LWM2M_OBJECT_SECURITY_ID, 0, 3), client_identity);
/* Set the client pre-shared key (PSK) */
lwm2m_set_opaque(&LWM2M_OBJ(LWM2M_OBJECT_SECURITY_ID, 0, 5), client_psk, sizeof(client_
↪psk));
```

An example of setting up the security object for X509 certificate mode:

```
static const char certificate[] = "-----BEGIN CERTIFICATE-----\nMIIB6jCCAY+gAw...";
static const char key[] = "-----BEGIN EC PRIVATE KEY-----\nMHcCAQ...";
static const char root_ca[] = "-----BEGIN CERTIFICATE-----\nMIIBaz...";

lwm2m_set_string(&LWM2M_OBJ(LWM2M_OBJECT_SECURITY_ID, 0, 0), "coaps://lwm2m.example.com");
lwm2m_set_u8(&LWM2M_OBJ(LWM2M_OBJECT_SECURITY_ID, 0, 2), LWM2M_SECURITY_CERT);
lwm2m_set_string(&LWM2M_OBJ(LWM2M_OBJECT_SECURITY_ID, 0, 3), certificate);
lwm2m_set_string(&LWM2M_OBJ(LWM2M_OBJECT_SECURITY_ID, 0, 5), key);
lwm2m_set_string(&LWM2M_OBJ(LWM2M_OBJECT_SECURITY_ID, 0, 4), root_ca);
```

Before calling *lwm2m_rd_client_start()* assign the tls_tag # where the LwM2M library should store the DTLS information prior to connection (normally a value of 1 is ok here).

```
(void)memset(&client, 0x0, sizeof(client));
client.tls_tag = 1; /* <---- */
lwm2m_rd_client_start(&client, "endpoint-name", 0, rd_client_event);
```

For a more detailed LwM2M client sample see: lwm2m-client.

Multi-thread usage Writing a value to a resource can be done using functions like lwm2m_set_u8. When writing to multiple resources, the function lwm2m_registry_lock will ensure that the client halts until all writing operations are finished:

```
lwm2m_registry_lock();
lwm2m_set_u32(&LWM2M_OBJ(1, 0, 1), 60);
lwm2m_set_u8(&LWM2M_OBJ(5, 0, 3), 0);
lwm2m_set_f64(&LWM2M_OBJ(3303, 0, 5700), value);
lwm2m_registry_unlock();
```

This is especially useful if the server is composite-observing the resources being written to. Locking will then ensure that the client only updates and sends notifications to the server after all operations are done, resulting in fewer messages in general.

Support for time series data LwM2M version 1.1 adds support for SenML CBOR and SenML JSON data formats. These data formats add support for time series data. Time series formats can

be used for READ, NOTIFY and SEND operations. When data cache is enabled for a resource, each write will create a timestamped entry in a cache, and its content is then returned as a content in READ, NOTIFY or SEND operation for a given resource.

Data cache is only supported for resources with a fixed data size.

Supported resource types:

- Signed and unsigned 8-64-bit integers
- Float
- Boolean

Enabling and configuring Enable data cache by selecting `CONFIG_LWM2M_RESOURCE_DATA_CACHE_SUPPORT`. Application needs to allocate an array of *lwm2m_time_series_elem* structures and then enable the cache by calling `lwm2m_engine_enable_cache()` for a given resource. Each resource must be enabled separately and each resource needs their own storage.

```
/* Allocate data cache storage */
static struct lwm2m_time_series_elem temperature_cache[10];
/* Enable data cache */
lwm2m_engine_enable_cache(LWM2M_PATH(IPSO_OBJECT_TEMP_SENSOR_ID, 0, SENSOR_VALUE_RID),
        temperature_cache, ARRAY_SIZE(temperature_cache));
```

LwM2M engine have room for four resources that have cache enabled. Limit can be increased by changing `CONFIG_LWM2M_MAX_CACHED_RESOURCES`. This affects a static memory usage of engine.

Data caches depends on one of the SenML data formats `CONFIG_LWM2M_RW_SENML_CBOR_SUPPORT` or `CONFIG_LWM2M_RW_SENML_JSON_SUPPORT` and needs `CONFIG_POSIX_TIMERS` so it can request a timestamp from the system and `CONFIG_RING_BUFFER` for ring buffer.

Read and Write operations Full content of data cache is written into a payload when any READ, SEND or NOTIFY operation internally reads the content of a given resource. This has a side effect that any read callbacks registered for a that resource are ignored when cache is enabled. Data is written into a cache when any of the `lwm2m_set_*` functions are called. To filter the data entering the cache, application may register a validation callback using *lwm2m_register_validate_callback()*.

Limitations Cache size should be manually set so small that the content can fit normal packets sizes. When cache is full, new values are dropped.

LwM2M engine and application events The Zephyr LwM2M engine defines events that can be sent back to the application through callback functions. The engine state machine shows when the events are spawned. Events depicted in the diagram are listed in the table. The events are prefixed with `LWM2M_RD_CLIENT_EVENT_`.

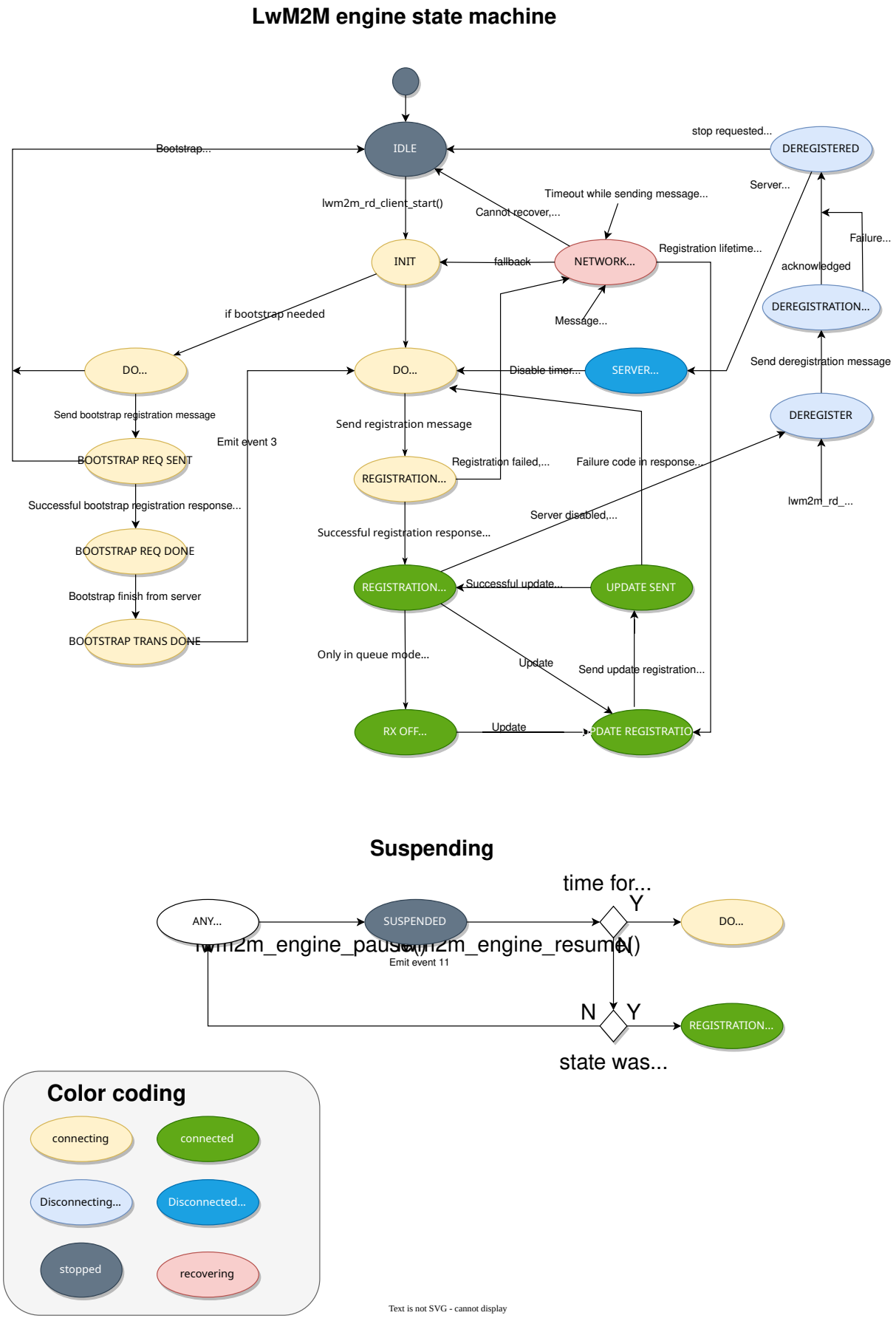


Fig. 15: State machine for the LwM2M engine

Table 30: LwM2M RD Client events

Ever ID	Event Name	Description
0	NONE	No event
1	BOOT-STRAP_REG_F	Bootstrap registration failed. Occurs if there is a timeout or failure in bootstrap registration.
2	BOOT-STRAP_REG_C	Bootstrap registration complete. Occurs after successful bootstrap registration.
3	BOOT-STRAP_TRANS	Bootstrap finish command received from the server.
4	REGISTRA-TION_FAILUR	Registration to LwM2M server failed. Occurs if there is a failure in the registration.
5	REGISTRA-TION_COMPL	Registration to LwM2M server successful. Occurs after a successful registration reply from the LwM2M server or when session resumption is used.
6	REG_TIMEOU	Registration or registration update timeout. Occurs if there is a timeout during registration. Client have lost connection to the server.
7	REG_UPDATE_	Registration update completed. Occurs after successful registration update reply from the LwM2M server.
8	DEREGIS-TER_FAILURE	Deregistration to LwM2M server failed. Occurs if there is a timeout or failure in the deregistration.
9	DISCON-NECT	LwM2M client have de-registered from server and is now stopped. Triggered only if the application have requested the client to stop.
10	QUEUE_MODI	Used only in queue mode, not actively listening for incoming packets. In queue mode the client is not required to actively listen for the incoming packets after a configured time period.
11	EN-GINE_SUSPEN	Indicate that client has now paused as a result of calling *lwm2m_engine_pause()*. State machine is no longer running and the handler thread is suspended. All timers are stopped so notifications are not triggered.
12	SERVER_DISA	Server have executed the disable command. Client will deregister and stay idle for the disable period.
13	NET-WORK_ERROI	Sending messages to the network failed too many times. Client cannot reach any servers or fallback to bootstrap. LwM2M engine cannot recover and have stopped.

The LwM2M client engine handles most of the state transitions automatically. The application needs to handle only the events that indicate that the client have stopped or is in a state where it cannot recover.

Table 31: How application should react to events

Event Name	How application should react
NONE	Ignore the event.
BOOT-STRAP_REG_FAIL	Try to recover network connection. Then restart the client by calling *lwm2m_rd_client_start()*. This might also indicate configuration issue.
BOOT-STRAP_REG_COM	No actions needed
BOOT-STRAP_TRANSFE	No actions needed
REGISTRA-TION_FAILURE	No actions needed
REGISTRA-TION_COMPLETI	No actions needed. Application can send or receive data.
REG_TIMEOUT	No actions needed. Client proceeds to re-registration automatically. Cannot send or receive data.
REG_UPDATE_CO	No actions needed Application can send or receive data.
DEREGIS-TER_FAILURE	No actions needed, client proceeds to idle state automatically. Cannot send or receive data.
DISCONNECT	Engine have stopped as a result of calling *lwm2m_rd_client_stop()*. If connection is required, the application should restart the client by calling *lwm2m_rd_client_start()*.
QUEUE_MODE_R	No actions needed. Application can send but cannot receive data. Any data transmission will trigger a registration update.
EN-GINE_SUSPENDE	Engine can be resumed by calling *lwm2m_engine_resume()*. Cannot send or receive data.
SERVER_DISABLI	No actions needed, client will re-register once the disable period is over. Cannot send or receive data.
NET-WORK_ERROR	Try to recover network connection. Then restart the client by calling *lwm2m_rd_client_start()*. This might also indicate configuration issue.

Sending of data in the table above refers to calling *lwm2m_send_cb()* or by writing into one of the observed resources where observation would trigger a notify message. Receiving of data refers to receiving read, write or execute operations from the server. Application can register callbacks for these operations.

Configuring lifetime and activity period In LwM2M engine, there are three Kconfig options and one runtime value that configures how often the client will send LwM2M Update message.

Table 32: Update period variables

Variable	Effect
LwM2M regis-tration lifetime	The lifetime parameter in LwM2M specifies how long a device's registration with an LwM2M server remains valid. Device is expected to send LwM2M Update message before the lifetime exprires.
CON-FIG_LWM2M_ENGIN	Default lifetime value, unless set by the bootstrap server. Also defines lower limit that client accepts as a lifetime.
CON-FIG_LWM2M_UPDAT	How long the client can stay idle before sending a next update.
CON-FIG_LWM2M_SECON	Minimum time margin to send the update message before the registration lifetime expires.

By default, the client uses `CONFIG_LWM2M_SECONDS_TO_UPDATE_EARLY` to calculate how many seconds before the expiration of lifetime it is going to send the registration update. The problem with default mode is when the server changes the lifetime of the registration. This is then affecting the period of updates the client is doing. If this is used with the QUEUE mode, which is

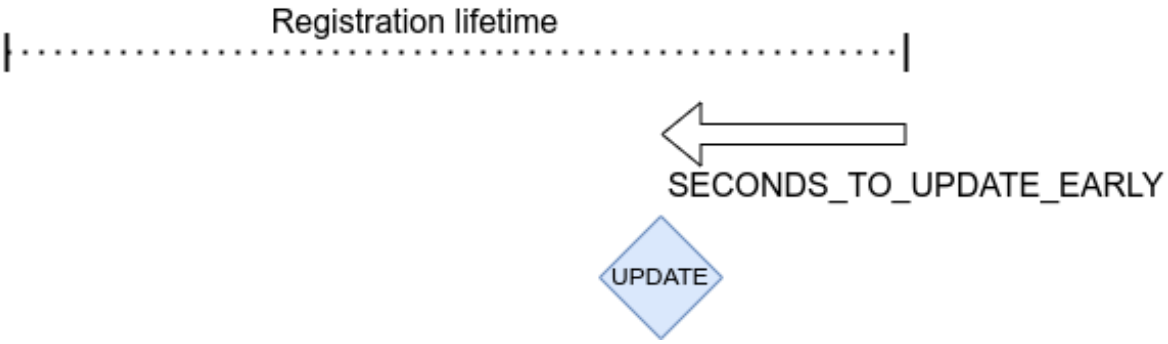


Fig. 16: Default way of calculating when to update registration.

typical in IPv4 networks, it is also affecting the period of when the device is reachable from the server.

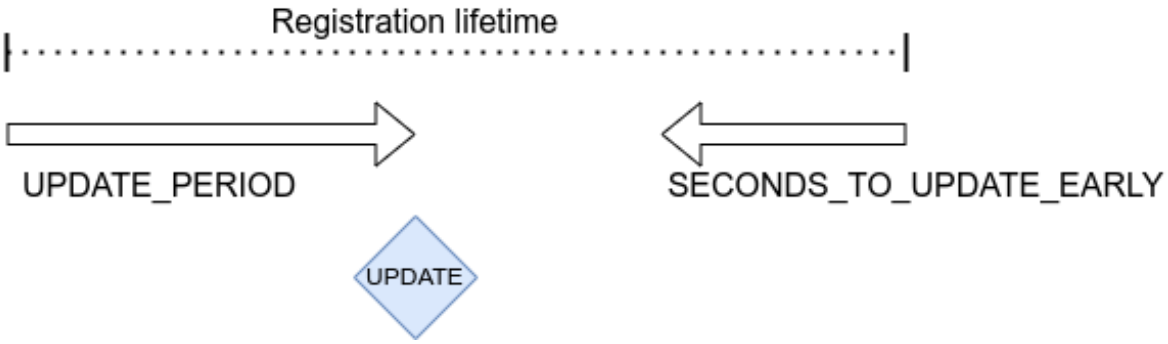


Fig. 17: Update time is controlled by UPDATE_PERIOD.

When also the `CONFIG_LWM2M_UPDATE_PERIOD` is set, time to send the update message is the earliest when any of these values expire. This allows setting long lifetime for the registration and configure the period accurately, even if server changes the lifetime parameter.

In runtime, the update frequency is limited to once in 15 seconds to avoid flooding.

LwM2M shell For testing the client it is possible to enable Zephyr's shell and LwM2M specific commands which support changing the state of the client. Operations supported are read, write and execute resources. Client start, stop, pause and resume are also available. The feature is enabled by selecting `CONFIG_LWM2M_SHELL`. The shell is meant for testing so productions systems should not enable it.

One imaginable scenario, where to use the shell, would be executing client side actions over UART when a server side tests would require those. It is assumed that not all tests are able to trigger required actions from the server side.

```
uart:~$ lwm2m
lwm2m - LwM2M commands
Subcommands:
  send    :send PATHS
          LwM2M SEND operation

  exec    :exec PATH [PARAM]
          Execute a resource

  read    :read PATH [OPTIONS]
          Read value from LwM2M resource
```

(continues on next page)

(continued from previous page)

```
          -x   Read value as hex stream (default)
          -s   Read value as string
          -b   Read value as bool (1/0)
          -uX  Read value as uintX_t
          -sX  Read value as intX_t
          -f   Read value as float
          -t   Read value as time_t

  write   :write PATH [OPTIONS] VALUE
          Write into LwM2M resource
          -s   Write value as string (default)
          -b   Write value as bool
          -uX  Write value as uintX_t
          -sX  Write value as intX_t
          -f   Write value as float
          -t   Write value as time_t

  create  :create PATH
          Create object or resource instance

  delete  :delete PATH
          Delete object or resource instance

  cache   :cache PATH NUM
          Enable data cache for resource
          PATH is LwM2M path
          NUM how many elements to cache

  start   :start EP_NAME [BOOTSTRAP FLAG]
          Start the LwM2M RD (Registration / Discovery) Client
          -b   Set the bootstrap flag (default 0)

  stop    :stop [OPTIONS]
          Stop the LwM2M RD (De-register) Client
          -f   Force close the connection

  update  :Trigger Registration Update of the LwM2M RD Client

  pause   :LwM2M engine thread pause
  resume  :LwM2M engine thread resume
  lock    :Lock the LwM2M registry
  unlock  :Unlock the LwM2M registry
```

Related code samples

LwM2M client
Implement a LwM2M client that connects to a LwM2M server.

API Reference

group `lwm2m_api`

Since
1.9

Version
0.8.0

LwM2M Objects managed by OMA for LwM2M tech specification.

Objects in this range have IDs from 0 to 1023.

`LWM2M_OBJECT_SECURITY_ID`
: Security object.

`LWM2M_OBJECT_SERVER_ID`
: Server object.

`LWM2M_OBJECT_ACCESS_CONTROL_ID`
: Access Control object.

`LWM2M_OBJECT_DEVICE_ID`
: Device object.

`LWM2M_OBJECT_CONNECTIVITY_MONITORING_ID`
: Connectivity Monitoring object.

`LWM2M_OBJECT_FIRMWARE_ID`
: Firmware object.

`LWM2M_OBJECT_LOCATION_ID`
: Location object.

`LWM2M_OBJECT_CONNECTIVITY_STATISTICS_ID`
: Connectivity Statistics object.

`LWM2M_OBJECT_SOFTWARE_MANAGEMENT_ID`
: Software Management object.

`LWM2M_OBJECT_PORTFOLIO_ID`
: Portfolio object.

`LWM2M_OBJECT_BINARYAPPDATACONTAINER_ID`
: Binary App Data Container object.

`LWM2M_OBJECT_EVENT_LOG_ID`
: Event Log object.

`LWM2M_OBJECT_OSCORE_ID`
: OSCORE object.

`LWM2M_OBJECT_GATEWAY_ID`
: Gateway object.

LwM2M Objects produced by 3rd party Standards Development

Organizations.

Refer to the OMA LightweightM2M (LwM2M) Object and Resource Registry: http://www.openmobilealliance.org/wp/OMNA/LwM2M/LwM2MRegistry.html

`IPSO_OBJECT_GENERIC_SENSOR_ID`
: IPSO Generic Sensor object.

`IPSO_OBJECT_TEMP_SENSOR_ID`
: IPSO Temperature Sensor object.

`IPSO_OBJECT_HUMIDITY_SENSOR_ID`
: IPSO Humidity Sensor object.

`IPSO_OBJECT_LIGHT_CONTROL_ID`
: IPSO Light Control object.

`IPSO_OBJECT_ACCELEROMETER_ID`
: IPSO Accelerometer object.

`IPSO_OBJECT_VOLTAGE_SENSOR_ID`
: IPSO Voltage Sensor object.

`IPSO_OBJECT_CURRENT_SENSOR_ID`
: IPSO Current Sensor object.

`IPSO_OBJECT_PRESSURE_ID`
: IPSO Pressure Sensor object.

`IPSO_OBJECT_BUZZER_ID`
: IPSO Buzzer object.

`IPSO_OBJECT_TIMER_ID`
: IPSO Timer object.

`IPSO_OBJECT_ONOFF_SWITCH_ID`
: IPSO On/Off Switch object.

`IPSO_OBJECT_PUSH_BUTTON_ID`
: IPSO Push Button object.

`UCIFI_OBJECT_BATTERY_ID`
: uCIFI Battery object

`IPSO_OBJECT_FILLING_LEVEL_SENSOR_ID`
: IPSO Filling Level Sensor object.

Power source types used for the “Available Power Sources” resource of

the LwM2M Device object (3/0/6).

`LWM2M_DEVICE_PWR_SRC_TYPE_DC_POWER`
: DC power.

`LWM2M_DEVICE_PWR_SRC_TYPE_BAT_INT`
: Internal battery.

`LWM2M_DEVICE_PWR_SRC_TYPE_BAT_EXT`
: External battery.

`LWM2M_DEVICE_PWR_SRC_TYPE_FUEL_CELL`
: Fuel cell.

`LWM2M_DEVICE_PWR_SRC_TYPE_PWR_OVER_ETH`
: Power over Ethernet.

`LWM2M_DEVICE_PWR_SRC_TYPE_USB`
: USB.

`LWM2M_DEVICE_PWR_SRC_TYPE_AC_POWER`
: AC (mains) power.

`LWM2M_DEVICE_PWR_SRC_TYPE_SOLAR`
: Solar.

`LWM2M_DEVICE_PWR_SRC_TYPE_MAX`
: Max value for Available Power Source type.

Error codes used for the “Error Code” resource of the LwM2M Device

object.

An LwM2M client can register one of the following error codes via the *lwm2m_device_add_err()* function.

`LWM2M_DEVICE_ERROR_NONE`
: No error.

`LWM2M_DEVICE_ERROR_LOW_POWER`
: Low battery power.

`LWM2M_DEVICE_ERROR_EXT_POWER_SUPPLY_OFF`
: External power supply off.

`LWM2M_DEVICE_ERROR_GPS_FAILURE`
: GPS module failure.

`LWM2M_DEVICE_ERROR_LOW_SIGNAL_STRENGTH`
: Low received signal strength.

`LWM2M_DEVICE_ERROR_OUT_OF_MEMORY`
: Out of memory.

`LWM2M_DEVICE_ERROR_SMS_FAILURE`
: SMS failure.

`LWM2M_DEVICE_ERROR_NETWORK_FAILURE`
: IP Connectivity failure.

`LWM2M_DEVICE_ERROR_PERIPHERAL_FAILURE`
: Peripheral malfunction.

Battery status codes used for the “Battery Status” resource (3/0/20)

of the LwM2M Device object.

As the battery status changes, an LwM2M client can set one of the following codes via: lwm2m_set_u8(“3/0/20”, [battery status])

`LWM2M_DEVICE_BATTERY_STATUS_NORMAL`
: The battery is operating normally and not on power.

`LWM2M_DEVICE_BATTERY_STATUS_CHARGING`
: The battery is currently charging.

`LWM2M_DEVICE_BATTERY_STATUS_CHARGE_COMP`
: The battery is fully charged and the charger is still connected.

`LWM2M_DEVICE_BATTERY_STATUS_DAMAGED`
: The battery has some problem.

`LWM2M_DEVICE_BATTERY_STATUS_LOW`
: The battery is low on charge.

`LWM2M_DEVICE_BATTERY_STATUS_NOT_INST`
: The battery is not installed.

`LWM2M_DEVICE_BATTERY_STATUS_UNKNOWN`
: The battery information is not available.

LWM2M Firmware Update object states

An LwM2M client or the LwM2M Firmware Update object use the following codes to represent the LwM2M Firmware Update state (5/0/3).

`STATE_IDLE`
: Idle.

  Before downloading or after successful updating.

STATE_DOWNLOADING
: Downloading.

  The data sequence is being downloaded.

STATE_DOWNLOADED
: Downloaded.

  The whole data sequence has been downloaded.

STATE_UPDATING
: Updating.

  The device is being updated.

LWM2M Firmware Update object result codes

After processing a firmware update, the client sets the result via one of the following codes via lwm2m_set_u8("5/0/5", [result code])

RESULT_DEFAULT
: Initial value.

RESULT_SUCCESS
: Firmware updated successfully.

RESULT_NO_STORAGE
: Not enough flash memory for the new firmware package.

RESULT_OUT_OF_MEM
: Out of RAM during downloading process.

RESULT_CONNECTION_LOST
: Connection lost during downloading process.

RESULT_INTEGRITY_FAILED
: Integrity check failure for new downloaded package.

RESULT_UNSUP_FW
: Unsupported package type.

RESULT_INVALID_URI
: Invalid URI.

RESULT_UPDATE_FAILED
: Firmware update failed.

RESULT_UNSUP_PROTO
: Unsupported protocol.

Defines

LWM2M_OBJLNK_MAX_ID
: Maximum value for Objlnk resource fields.

LWM2M_RES_DATA_READ_ONLY
: Resource read-only value bit.

LWM2M_RES_DATA_FLAG_RO
: Resource read-only flag.

LWM2M_HAS_RES_FLAG(res, f)
: Read resource flags helper macro.

LWM2M_RD_CLIENT_FLAG_BOOTSTRAP
: Run bootstrap procedure in current session.

LWM2M_MAX_PATH_STR_SIZE
: LwM2M path maximum length.

Typedefs

typedef void (*lwm2m_socket_fault_cb_t)(int error)
: Callback function called when a socket error is encountered.

  Param error
  : Error code

typedef void (*lwm2m_observe_cb_t)(enum *lwm2m_observe_event* event, struct *lwm2m_obj_path* *path, void *user_data)
: Observe callback indicating observer adds and deletes, and notification ACKs and timeouts.

  Param event
  : **[in]** Observer add/delete or notification ack/timeout

  Param path
  : **[in]** LwM2M path

  Param user_data
  : **[in]** Pointer to user_data buffer, as provided in send_traceable_notification(). Used to determine for which data the ACKed/timed out notification was.

typedef void (*lwm2m_ctx_event_cb_t)(struct *lwm2m_ctx* *ctx, enum *lwm2m_rd_client_event* event)
: Asynchronous RD client event callback.

  Param ctx
  : **[in]** LwM2M context generating the event

  Param event
  : **[in]** LwM2M RD client event code

typedef void *(*lwm2m_engine_get_data_cb_t)(uint16_t obj_inst_id, uint16_t res_id, uint16_t res_inst_id, size_t *data_len)

Asynchronous callback to get a resource buffer and length.

Prior to accessing the data buffer of a resource, the engine can use this callback to get the buffer pointer and length instead of using the resource's data buffer.

The client or LwM2M objects can register a function of this type via: *lwm2m_register_read_callback() lwm2m_register_pre_write_callback()*

Param obj_inst_id
[in] Object instance ID generating the callback.

Param res_id
[in] Resource ID generating the callback.

Param res_inst_id
[in] Resource instance ID generating the callback (typically 0 for non-multi instance resources).

Param data_len
[out] Length of the data buffer.

Return
Callback returns a pointer to the data buffer or NULL for failure.

typedef int (*lwm2m_engine_set_data_cb_t)(uint16_t obj_inst_id, uint16_t res_id, uint16_t res_inst_id, uint8_t *data, uint16_t data_len, bool last_block, size_t total_size, size_t offset)

Asynchronous callback when data has been set to a resource buffer.

After changing the data of a resource buffer, the LwM2M engine can make use of this callback to pass the data back to the client or LwM2M objects.

On a block-wise transfers the handler is called multiple times with the data blocks and increasing offset. The last block has the last_block flag set to true. Beginning of the block transfer has the offset set to 0.

A function of this type can be registered via: *lwm2m_register_validate_callback() lwm2m_register_post_write_callback()*

Param obj_inst_id
[in] Object instance ID generating the callback.

Param res_id
[in] Resource ID generating the callback.

Param res_inst_id
[in] Resource instance ID generating the callback (typically 0 for non-multi instance resources).

Param data
[in] Pointer to data.

Param data_len
[in] Length of the data.

Param last_block
[in] Flag used during block transfer to indicate the last block of data. For non-block transfers this is always false.

Param total_size
[in] Expected total size of data for a block transfer. For non-block transfers this is 0.

Param offset
[in] Offset of the data block. For non-block transfers this is always 0.

Return
Callback returns a negative error code (errno.h) indicating reason of failure or 0 for success.

typedef int (*`lwm2m_engine_user_cb_t`)(uint16_t obj_inst_id)

Asynchronous event notification callback.

Various object instance and resource-based events in the LwM2M engine can trigger a callback of this function type: object instance create, and object instance delete.

Register a function of this type via: *lwm2m_register_create_callback()* *lwm2m_register_delete_callback()*

Param obj_inst_id
[in] Object instance ID generating the callback.

Return
Callback returns a negative error code (errno.h) indicating reason of failure or 0 for success.

typedef int (*`lwm2m_engine_execute_cb_t`)(uint16_t obj_inst_id, uint8_t *args, uint16_t args_len)

Asynchronous execute notification callback.

Resource executes trigger a callback of this type.

Register a function of this type via: *lwm2m_register_exec_callback()*

Param obj_inst_id
[in] Object instance ID generating the callback.

Param args
[in] Pointer to execute arguments payload. (This can be NULL if no arguments are provided)

Param args_len
[in] Length of argument payload in bytes.

Return
Callback returns a negative error code (errno.h) indicating reason of failure or 0 for success.

typedef void (*`lwm2m_send_cb_t`)(enum *lwm2m_send_status* status)

Callback returning send status.

Enums

enum `lwm2m_observe_event`

Observe callback events.

Values:

enumerator `LWM2M_OBSERVE_EVENT_OBSERVER_ADDED`

Observer added.

enumerator `LWM2M_OBSERVE_EVENT_OBSERVER_REMOVED`

Observer removed.

enumerator LWM2M_OBSERVE_EVENT_NOTIFY_ACK

Notification ACKed.

enumerator LWM2M_OBSERVE_EVENT_NOTIFY_TIMEOUT

Notification timed out.

enum lwm2m_socket_states

Different traffic states of the LwM2M socket.

This information can be used to give hints for the network interface that can decide what kind of power management should be used.

These hints are given from CoAP layer messages, so usage of DTLS might affect the actual number of expected datagrams.

Values:

enumerator LWM2M_SOCKET_STATE_ONGOING

Ongoing traffic is expected.

enumerator LWM2M_SOCKET_STATE_ONE_RESPONSE

One response is expected for the next message.

enumerator LWM2M_SOCKET_STATE_LAST

Next message is the last one.

enumerator LWM2M_SOCKET_STATE_NO_DATA

No more data is expected.

enum lwm2m_rd_client_event

LwM2M RD client events.

LwM2M client events are passed back to the event_cb function in *lwm2m_rd_client_start()*

Values:

enumerator LWM2M_RD_CLIENT_EVENT_NONE

Invalid event.

enumerator LWM2M_RD_CLIENT_EVENT_BOOTSTRAP_REG_FAILURE

Bootstrap registration failure.

enumerator LWM2M_RD_CLIENT_EVENT_BOOTSTRAP_REG_COMPLETE

Bootstrap registration complete.

enumerator LWM2M_RD_CLIENT_EVENT_BOOTSTRAP_TRANSFER_COMPLETE

Bootstrap transfer complete.

enumerator LWM2M_RD_CLIENT_EVENT_REGISTRATION_FAILURE

Registration failure.

enumerator LWM2M_RD_CLIENT_EVENT_REGISTRATION_COMPLETE
Registration complete.

enumerator LWM2M_RD_CLIENT_EVENT_REG_TIMEOUT
Registration timeout.

enumerator LWM2M_RD_CLIENT_EVENT_REG_UPDATE_COMPLETE
Registration update complete.

enumerator LWM2M_RD_CLIENT_EVENT_DEREGISTER_FAILURE
De-registration failure.

enumerator LWM2M_RD_CLIENT_EVENT_DISCONNECT
Disconnected.

enumerator LWM2M_RD_CLIENT_EVENT_QUEUE_MODE_RX_OFF
Queue mode RX off.

enumerator LWM2M_RD_CLIENT_EVENT_ENGINE_SUSPENDED
Engine suspended.

enumerator LWM2M_RD_CLIENT_EVENT_NETWORK_ERROR
Network error.

enumerator LWM2M_RD_CLIENT_EVENT_REG_UPDATE
Registration update.

enumerator LWM2M_RD_CLIENT_EVENT_DEREGISTER
De-register.

enumerator LWM2M_RD_CLIENT_EVENT_SERVER_DISABLED
Server disabled.

enum lwm2m_send_status
LwM2M send status.

LwM2M send status are generated back to the lwm2m_send_cb_t function in *lwm2m_send_cb()*

Values:

enumerator LWM2M_SEND_STATUS_SUCCESS
Succeed.

enumerator LWM2M_SEND_STATUS_FAILURE
Failure.

enumerator LWM2M_SEND_STATUS_TIMEOUT
Timeout.

enum `lwm2m_security_mode_e`

Security modes as defined in LwM2M Security object.

Values:

enumerator `LWM2M_SECURITY_PSK` = 0

Pre-Shared Key mode.

enumerator `LWM2M_SECURITY_RAW_PK` = 1

Raw Public Key mode.

enumerator `LWM2M_SECURITY_CERT` = 2

Certificate mode.

enumerator `LWM2M_SECURITY_NOSEC` = 3

NoSec mode.

enumerator `LWM2M_SECURITY_CERT_EST` = 4

Certificate mode with EST.

Functions

int `lwm2m_device_add_err`(uint8_t error_code)

Register a new error code with LwM2M Device object.

Parameters

- `error_code` – **[in]** New error code.

Returns

0 for success or negative in case of error.

void `lwm2m_firmware_set_write_cb`(*lwm2m_engine_set_data_cb_t* cb)

Set data callback for firmware block transfer.

LwM2M clients use this function to register a callback for receiving the block transfer data when performing a firmware update.

Parameters

- `cb` – **[in]** A callback function to receive the block transfer data

lwm2m_engine_set_data_cb_t `lwm2m_firmware_get_write_cb`(void)

Get the data callback for firmware block transfer writes.

Returns

A registered callback function to receive the block transfer data

void `lwm2m_firmware_set_write_cb_inst`(uint16_t obj_inst_id, *lwm2m_engine_set_data_cb_t* cb)

Set data callback for firmware block transfer.

LwM2M clients use this function to register a callback for receiving the block transfer data when performing a firmware update.

Parameters

- `obj_inst_id` – **[in]** Object instance ID
- `cb` – **[in]** A callback function to receive the block transfer data

lwm2m_engine_set_data_cb_t **lwm2m_firmware_get_write_cb_inst**(uint16_t obj_inst_id)

Get the data callback for firmware block transfer writes.

Parameters

- **obj_inst_id** – **[in]** Object instance ID

Returns

A registered callback function to receive the block transfer data

void **lwm2m_firmware_set_cancel_cb**(*lwm2m_engine_user_cb_t* cb)

Set callback for firmware update cancel.

LwM2M clients use this function to register a callback to perform actions on firmware update cancel.

Parameters

- **cb** – **[in]** A callback function perform actions on firmware update cancel.

lwm2m_engine_user_cb_t **lwm2m_firmware_get_cancel_cb**(void)

Get a callback for firmware update cancel.

Returns

A registered callback function perform actions on firmware update cancel.

void **lwm2m_firmware_set_cancel_cb_inst**(uint16_t obj_inst_id, *lwm2m_engine_user_cb_t* cb)

Set data callback for firmware update cancel.

LwM2M clients use this function to register a callback to perform actions on firmware update cancel.

Parameters

- **obj_inst_id** – **[in]** Object instance ID
- **cb** – **[in]** A callback function perform actions on firmware update cancel.

lwm2m_engine_user_cb_t **lwm2m_firmware_get_cancel_cb_inst**(uint16_t obj_inst_id)

Get the callback for firmware update cancel.

Parameters

- **obj_inst_id** – **[in]** Object instance ID

Returns

A registered callback function perform actions on firmware update cancel.

void **lwm2m_firmware_set_update_cb**(*lwm2m_engine_execute_cb_t* cb)

Set data callback to handle firmware update execute events.

LwM2M clients use this function to register a callback for receiving the update resource “execute” operation on the LwM2M Firmware Update object.

Parameters

- **cb** – **[in]** A callback function to receive the execute event.

lwm2m_engine_execute_cb_t **lwm2m_firmware_get_update_cb**(void)

Get the event callback for firmware update execute events.

Returns

A registered callback function to receive the execute event.

void **lwm2m_firmware_set_update_cb_inst**(uint16_t obj_inst_id, *lwm2m_engine_execute_cb_t* cb)

Set data callback to handle firmware update execute events.

LwM2M clients use this function to register a callback for receiving the update resource "execute" operation on the LwM2M Firmware Update object.

Parameters

- **obj_inst_id** – **[in]** Object instance ID
- **cb** – **[in]** A callback function to receive the execute event.

lwm2m_engine_execute_cb_t **lwm2m_firmware_get_update_cb_inst**(uint16_t obj_inst_id)

Get the event callback for firmware update execute events.

Parameters

- **obj_inst_id** – **[in]** Object instance ID

Returns

A registered callback function to receive the execute event.

int **lwm2m_swmgmt_set_activate_cb**(uint16_t obj_inst_id, *lwm2m_engine_execute_cb_t* cb)

Set callback to handle software activation requests.

The callback will be executed when the LWM2M execute operation gets called on the corresponding object's Activate resource instance.

Parameters

- **obj_inst_id** – **[in]** The instance number to set the callback for.
- **cb** – **[in]** A callback function to receive the execute event.

Returns

0 on success, otherwise a negative integer.

int **lwm2m_swmgmt_set_deactivate_cb**(uint16_t obj_inst_id, *lwm2m_engine_execute_cb_t* cb)

Set callback to handle software deactivation requests.

The callback will be executed when the LWM2M execute operation gets called on the corresponding object's Deactivate resource instance.

Parameters

- **obj_inst_id** – **[in]** The instance number to set the callback for.
- **cb** – **[in]** A callback function to receive the execute event.

Returns

0 on success, otherwise a negative integer.

int **lwm2m_swmgmt_set_install_package_cb**(uint16_t obj_inst_id, *lwm2m_engine_execute_cb_t* cb)

Set callback to handle software install requests.

The callback will be executed when the LWM2M execute operation gets called on the corresponding object's Install resource instance.

Parameters

- **obj_inst_id** – **[in]** The instance number to set the callback for.
- **cb** – **[in]** A callback function to receive the execute event.

Returns

0 on success, otherwise a negative integer.

int `lwm2m_swmgmt_set_delete_package_cb`(uint16_t obj_inst_id, *lwm2m_engine_execute_cb_t* cb)

Set callback to handle software uninstall requests.

The callback will be executed when the LWM2M execute operation gets called on the corresponding object's Uninstall resource instance.

Parameters

- `obj_inst_id` – **[in]** The instance number to set the callback for.
- `cb` – **[in]** A callback function for handling the execute event.

Returns
0 on success, otherwise a negative integer.

int `lwm2m_swmgmt_set_read_package_version_cb`(uint16_t obj_inst_id, *lwm2m_engine_get_data_cb_t* cb)

Set callback to read software package.

The callback will be executed when the LWM2M read operation gets called on the corresponding object.

Parameters

- `obj_inst_id` – **[in]** The instance number to set the callback for.
- `cb` – **[in]** A callback function for handling the read event.

Returns
0 on success, otherwise a negative integer.

int `lwm2m_swmgmt_set_write_package_cb`(uint16_t obj_inst_id, *lwm2m_engine_set_data_cb_t* cb)

Set data callback for software management block transfer.

The callback will be executed when the LWM2M block write operation gets called on the corresponding object's resource instance.

Parameters

- `obj_inst_id` – **[in]** The instance number to set the callback for.
- `cb` – **[in]** A callback function for handling the block write event.

Returns
0 on success, otherwise a negative integer.

int `lwm2m_swmgmt_install_completed`(uint16_t obj_inst_id, int error_code)

Function to be called when a Software Management object instance completed the Install operation.

return 0 on success, otherwise a negative integer.

Parameters

- `obj_inst_id` – **[in]** The Software Management object instance
- `error_code` – **[in]** The result code of the operation. Zero on success otherwise it should be a negative integer.

void `lwm2m_event_log_set_read_log_data_cb`(*lwm2m_engine_get_data_cb_t* cb)

Set callback to read log data.

The callback will be executed when the LWM2M read operation gets called on the corresponding object.

Parameters

- **cb** – **[in]** A callback function for handling the read event.

int **lwm2m_update_observer_min_period**(struct *lwm2m_ctx* *client_ctx, const struct *lwm2m_obj_path* *path, uint32_t period_s)

Change an observer's pmin value.

LwM2M clients use this function to modify the pmin attribute for an observation being made. Example to update the pmin of a temperature sensor value being observed: lwm2m_update_observer_min_period(client_ctx, &LWM2M_OBJ(3303, 0, 5700), 5);

Parameters

- **client_ctx** – **[in]** LwM2M context
- **path** – **[in]** LwM2M path as a struct
- **period_s** – **[in]** Value of pmin to be given (in seconds).

Returns

0 for success or negative in case of error.

int **lwm2m_update_observer_max_period**(struct *lwm2m_ctx* *client_ctx, const struct *lwm2m_obj_path* *path, uint32_t period_s)

Change an observer's pmax value.

LwM2M clients use this function to modify the pmax attribute for an observation being made. Example to update the pmax of a temperature sensor value being observed: lwm2m__update_observer_max_period(client_ctx, &LWM2M_OBJ(3303, 0, 5700), 5);

Parameters

- **client_ctx** – **[in]** LwM2M context
- **path** – **[in]** LwM2M path as a struct
- **period_s** – **[in]** Value of pmax to be given (in seconds).

Returns

0 for success or negative in case of error.

int **lwm2m_create_object_inst**(const struct *lwm2m_obj_path* *path)

Create an LwM2M object instance.

LwM2M clients use this function to create non-default LwM2M objects: Example to create first temperature sensor object: lwm2m_create_obj_inst(&LWM2M_OBJ(3303, 0));

Parameters

- **path** – **[in]** LwM2M path as a struct

Returns

0 for success or negative in case of error.

int **lwm2m_delete_object_inst**(const struct *lwm2m_obj_path* *path)

Delete an LwM2M object instance.

LwM2M clients use this function to delete LwM2M objects.

Parameters

- **path** – **[in]** LwM2M path as a struct

Returns

0 for success or negative in case of error.

void `lwm2m_registry_lock`(void)

Locks the registry for this thread.

Use this function before writing to multiple resources. This halts the lwm2m main thread until all the write-operations are finished.

void `lwm2m_registry_unlock`(void)

Unlocks the registry previously locked by *lwm2m_registry_lock()*.

int `lwm2m_set_opaque`(const struct *lwm2m_obj_path* *path, const char *data_ptr, uint16_t data_len)

Set resource (instance) value (opaque buffer)

Parameters

- `path` – **[in]** LwM2M path as a struct
- `data_ptr` – **[in]** Data buffer
- `data_len` – **[in]** Length of buffer

Returns

0 for success or negative in case of error.

int `lwm2m_set_string`(const struct *lwm2m_obj_path* *path, const char *data_ptr)

Set resource (instance) value (string)

Parameters

- `path` – **[in]** LwM2M path as a struct
- `data_ptr` – **[in]** NULL terminated char buffer

Returns

0 for success or negative in case of error.

int `lwm2m_set_u8`(const struct *lwm2m_obj_path* *path, uint8_t value)

Set resource (instance) value (u8)

Parameters

- `path` – **[in]** LwM2M path as a struct
- `value` – **[in]** u8 value

Returns

0 for success or negative in case of error.

int `lwm2m_set_u16`(const struct *lwm2m_obj_path* *path, uint16_t value)

Set resource (instance) value (u16)

Parameters

- `path` – **[in]** LwM2M path as a struct
- `value` – **[in]** u16 value

Returns

0 for success or negative in case of error.

int `lwm2m_set_u32`(const struct *lwm2m_obj_path* *path, uint32_t value)

Set resource (instance) value (u32)

Parameters

- `path` – **[in]** LwM2M path as a struct
- `value` – **[in]** u32 value

Returns
0 for success or negative in case of error.

int **lwm2m_set_u64**(const struct *lwm2m_obj_path* *path, uint64_t value)
Set resource (instance) value (u64)

Deprecated:
Unsigned 64bit value type does not exits. This is internally handled as a int64_t. Use *lwm2m_set_s64()* instead.

Parameters
- **path** – **[in]** LwM2M path as a struct
- **value** – **[in]** u64 value

Returns
0 for success or negative in case of error.

int **lwm2m_set_s8**(const struct *lwm2m_obj_path* *path, int8_t value)
Set resource (instance) value (s8)

Parameters
- **path** – **[in]** LwM2M path as a struct
- **value** – **[in]** s8 value

Returns
0 for success or negative in case of error.

int **lwm2m_set_s16**(const struct *lwm2m_obj_path* *path, int16_t value)
Set resource (instance) value (s16)

Parameters
- **path** – **[in]** LwM2M path as a struct
- **value** – **[in]** s16 value

Returns
0 for success or negative in case of error.

int **lwm2m_set_s32**(const struct *lwm2m_obj_path* *path, int32_t value)
Set resource (instance) value (s32)

Parameters
- **path** – **[in]** LwM2M path as a struct
- **value** – **[in]** s32 value

Returns
0 for success or negative in case of error.

int **lwm2m_set_s64**(const struct *lwm2m_obj_path* *path, int64_t value)
Set resource (instance) value (s64)

Parameters
- **path** – **[in]** LwM2M path as a struct
- **value** – **[in]** s64 value

Returns
0 for success or negative in case of error.

int **lwm2m_set_bool**(const struct *lwm2m_obj_path* *path, bool value)

Set resource (instance) value (bool)

Parameters

- **path** – **[in]** LwM2M path as a struct
- **value** – **[in]** bool value

Returns

0 for success or negative in case of error.

int **lwm2m_set_f64**(const struct *lwm2m_obj_path* *path, const double value)

Set resource (instance) value (double)

Parameters

- **path** – **[in]** LwM2M path as a struct
- **value** – **[in]** double value

Returns

0 for success or negative in case of error.

int **lwm2m_set_objlnk**(const struct *lwm2m_obj_path* *path, const struct *lwm2m_objlnk* *value)

Set resource (instance) value (Objlnk)

Parameters

- **path** – **[in]** LwM2M path as a struct
- **value** – **[in]** pointer to the *lwm2m_objlnk* structure

Returns

0 for success or negative in case of error.

int **lwm2m_set_time**(const struct *lwm2m_obj_path* *path, time_t value)

Set resource (instance) value (Time)

Parameters

- **path** – **[in]** LwM2M path as a struct
- **value** – **[in]** Epoch timestamp

Returns

0 for success or negative in case of error.

int **lwm2m_set_bulk**(const struct *lwm2m_res_item* res_list[], size_t res_list_size)

Set multiple resource (instance) values.

NOTE: Value type must match the target resource as this function does not do any type conversion. See struct *lwm2m_res_item* for list of resource types.

Parameters

- **res_list** – **[in]** LwM2M resource item list
- **res_list_size** – **[in]** Length of resource list

Returns

0 for success or negative in case of error.

int **lwm2m_get_opaque**(const struct *lwm2m_obj_path* *path, void *buf, uint16_t buflen)

Get resource (instance) value (opaque buffer)

Parameters

- **path** – **[in]** LwM2M path as a struct

- `buf` – **[out]** Data buffer to copy data into
- `buflen` – **[in]** Length of buffer

Returns
0 for success or negative in case of error.

int `lwm2m_get_string`(const struct *lwm2m_obj_path* *path, void *str, uint16_t buflen)
Get resource (instance) value (string)

Parameters

- `path` – **[in]** LwM2M path as a struct
- `str` – **[out]** String buffer to copy data into
- `buflen` – **[in]** Length of buffer

Returns
0 for success or negative in case of error.

int `lwm2m_get_u8`(const struct *lwm2m_obj_path* *path, uint8_t *value)
Get resource (instance) value (u8)

Parameters

- `path` – **[in]** LwM2M path as a struct
- `value` – **[out]** u8 buffer to copy data into

Returns
0 for success or negative in case of error.

int `lwm2m_get_u16`(const struct *lwm2m_obj_path* *path, uint16_t *value)
Get resource (instance) value (u16)

Parameters

- `path` – **[in]** LwM2M path as a struct
- `value` – **[out]** u16 buffer to copy data into

Returns
0 for success or negative in case of error.

int `lwm2m_get_u32`(const struct *lwm2m_obj_path* *path, uint32_t *value)
Get resource (instance) value (u32)

Parameters

- `path` – **[in]** LwM2M path as a struct
- `value` – **[out]** u32 buffer to copy data into

Returns
0 for success or negative in case of error.

int `lwm2m_get_u64`(const struct *lwm2m_obj_path* *path, uint64_t *value)
Get resource (instance) value (u64)

Deprecated:
Unsigned 64bit value type does not exits. This is internally handled as a int64_t. Use *lwm2m_get_s64()* instead.

Parameters

- `path` – **[in]** LwM2M path as a struct
- `value` – **[out]** u64 buffer to copy data into

Returns
0 for success or negative in case of error.

int `lwm2m_get_s8`(const struct *lwm2m_obj_path* *path, int8_t *value)
Get resource (instance) value (s8)

Parameters
- `path` – **[in]** LwM2M path as a struct
- `value` – **[out]** s8 buffer to copy data into

Returns
0 for success or negative in case of error.

int `lwm2m_get_s16`(const struct *lwm2m_obj_path* *path, int16_t *value)
Get resource (instance) value (s16)

Parameters
- `path` – **[in]** LwM2M path as a struct
- `value` – **[out]** s16 buffer to copy data into

Returns
0 for success or negative in case of error.

int `lwm2m_get_s32`(const struct *lwm2m_obj_path* *path, int32_t *value)
Get resource (instance) value (s32)

Parameters
- `path` – **[in]** LwM2M path as a struct
- `value` – **[out]** s32 buffer to copy data into

Returns
0 for success or negative in case of error.

int `lwm2m_get_s64`(const struct *lwm2m_obj_path* *path, int64_t *value)
Get resource (instance) value (s64)

Parameters
- `path` – **[in]** LwM2M path as a struct
- `value` – **[out]** s64 buffer to copy data into

Returns
0 for success or negative in case of error.

int `lwm2m_get_bool`(const struct *lwm2m_obj_path* *path, bool *value)
Get resource (instance) value (bool)

Parameters
- `path` – **[in]** LwM2M path as a struct
- `value` – **[out]** bool buffer to copy data into

Returns
0 for success or negative in case of error.

int `lwm2m_get_f64`(const struct *lwm2m_obj_path* *path, double *value)
Get resource (instance) value (double)

Parameters
- `path` – **[in]** LwM2M path as a struct
- `value` – **[out]** double buffer to copy data into

Returns
0 for success or negative in case of error.

int `lwm2m_get_objlnk`(const struct *lwm2m_obj_path* *path, struct *lwm2m_objlnk* *buf)

Get resource (instance) value (Objlnk)

Parameters

- `path` – **[in]** LwM2M path as a struct
- `buf` – **[out]** *lwm2m_objlnk* buffer to copy data into

Returns
0 for success or negative in case of error.

int `lwm2m_get_time`(const struct *lwm2m_obj_path* *path, time_t *buf)

Get resource (instance) value (Time)

Parameters

- `path` – **[in]** LwM2M path as a struct
- `buf` – **[out]** time_t pointer to copy data

Returns
0 for success or negative in case of error.

int `lwm2m_register_read_callback`(const struct *lwm2m_obj_path* *path,
lwm2m_engine_get_data_cb_t cb)

Set resource (instance) read callback.

LwM2M clients can use this to set the callback function for resource reads when data handling in the LwM2M engine needs to be bypassed. For example reading back opaque binary data from external storage.

This callback should not generally be used for any data that might be observed as engine does not have any knowledge of data changes.

When separate buffer for data should be used, use *lwm2m_set_res_buf()* instead to set the storage.

Parameters

- `path` – **[in]** LwM2M path as a struct
- `cb` – **[in]** Read resource callback

Returns
0 for success or negative in case of error.

int `lwm2m_register_pre_write_callback`(const struct *lwm2m_obj_path* *path,
lwm2m_engine_get_data_cb_t cb)

Set resource (instance) pre-write callback.

This callback is triggered before setting the value of a resource. It can pass a special data buffer to the engine so that the actual resource value can be calculated later, etc.

Parameters

- `path` – **[in]** LwM2M path as a struct
- `cb` – **[in]** Pre-write resource callback

Returns
0 for success or negative in case of error.

int **lwm2m_register_validate_callback**(const struct *lwm2m_obj_path* *path, *lwm2m_engine_set_data_cb_t* cb)

Set resource (instance) validation callback.

This callback is triggered before setting the value of a resource to the resource data buffer.

The callback allows an LwM2M client or object to validate the data before writing and notify an error if the data should be discarded for any reason (by returning a negative error code).

> **Note**
>
> All resources that have a validation callback registered are initially decoded into a temporary validation buffer. Make sure that CONFIG_LWM2M_ENGINE_VALIDATION_BUFFER_SIZE is large enough to store each of the validated resources (individually).

Parameters

- **path** – **[in]** LwM2M path as a struct
- **cb** – **[in]** Validate resource data callback

Returns
0 for success or negative in case of error.

int **lwm2m_register_post_write_callback**(const struct *lwm2m_obj_path* *path, *lwm2m_engine_set_data_cb_t* cb)

Set resource (instance) post-write callback.

This callback is triggered after setting the value of a resource to the resource data buffer.

It allows an LwM2M client or object to post-process the value of a resource or trigger other related resource calculations.

Parameters

- **path** – **[in]** LwM2M path as a struct
- **cb** – **[in]** Post-write resource callback

Returns
0 for success or negative in case of error.

int **lwm2m_register_exec_callback**(const struct *lwm2m_obj_path* *path, *lwm2m_engine_execute_cb_t* cb)

Set resource execute event callback.

This event is triggered when the execute method of a resource is enabled.

Parameters

- **path** – **[in]** LwM2M path as a struct
- **cb** – **[in]** Execute resource callback

Returns
0 for success or negative in case of error.

int **lwm2m_register_create_callback**(uint16_t obj_id, *lwm2m_engine_user_cb_t* cb)

Set object instance create event callback.

This event is triggered when an object instance is created.

Parameters

- `obj_id` – **[in]** LwM2M object id
- `cb` – **[in]** Create object instance callback

Returns
0 for success or negative in case of error.

int `lwm2m_register_delete_callback`(uint16_t obj_id, *lwm2m_engine_user_cb_t* cb)

Set object instance delete event callback.

This event is triggered when an object instance is deleted.

Parameters

- `obj_id` – **[in]** LwM2M object id
- `cb` – **[in]** Delete object instance callback

Returns
0 for success or negative in case of error.

int `lwm2m_set_res_buf`(const struct *lwm2m_obj_path* *path, void *buffer_ptr, uint16_t buffer_len, uint16_t data_len, uint8_t data_flags)

Set data buffer for a resource.

Use this function to set the data buffer and flags for the specified LwM2M resource.

Parameters

- `path` – **[in]** LwM2M path as a struct
- `buffer_ptr` – **[in]** Data buffer pointer
- `buffer_len` – **[in]** Length of buffer
- `data_len` – **[in]** Length of existing data in the buffer
- `data_flags` – **[in]** Data buffer flags (such as read-only, etc)

Returns
0 for success or negative in case of error.

int `lwm2m_set_res_data_len`(const struct *lwm2m_obj_path* *path, uint16_t data_len)

Update data size for a resource.

Use this function to set the new size of data in the buffer if you write to a buffer received by *lwm2m_get_res_buf()*.

Parameters

- `path` – **[in]** LwM2M path as a struct
- `data_len` – **[in]** Length of data

Returns
0 for success or negative in case of error.

int `lwm2m_get_res_buf`(const struct *lwm2m_obj_path* *path, void **buffer_ptr, uint16_t *buffer_len, uint16_t *data_len, uint8_t *data_flags)

Get data buffer for a resource.

Use this function to get the data buffer information for the specified LwM2M resource.

If you directly write into the buffer, you must use *lwm2m_set_res_data_len()* function to update the new size of the written data.

All parameters, except for the pathstr, can be NULL if you don't want to read those values.

Parameters

- `path` – **[in]** LwM2M path as a struct
- `buffer_ptr` – **[out]** Data buffer pointer
- `buffer_len` – **[out]** Length of buffer
- `data_len` – **[out]** Length of existing data in the buffer
- `data_flags` – **[out]** Data buffer flags (such as read-only, etc)

Returns
0 for success or negative in case of error.

int `lwm2m_create_res_inst`(const struct *lwm2m_obj_path* *path)

Create a resource instance.

LwM2M clients use this function to create multi-resource instances: Example to create 0 instance of device available power sources: lwm2m_create_res_inst(&LWM2M_OBJ(3, 0, 6, 0));

Parameters

- `path` – **[in]** LwM2M path as a struct

Returns
0 for success or negative in case of error.

int `lwm2m_delete_res_inst`(const struct *lwm2m_obj_path* *path)

Delete a resource instance.

Use this function to remove an existing resource instance

Parameters

- `path` – **[in]** LwM2M path as a struct

Returns
0 for success or negative in case of error.

int `lwm2m_update_device_service_period`(uint32_t period_ms)

Update the period of the device service.

Change the duration of the periodic device service that notifies the current time.

Parameters

- `period_ms` – **[in]** New period for the device service (in milliseconds)

Returns
0 for success or negative in case of error.

bool `lwm2m_path_is_observed`(const struct *lwm2m_obj_path* *path)

Check whether a path is observed.

Parameters

- `path` – **[in]** LwM2M path as a struct to check

Returns
true when there exists an observation of the same level or lower as the given path, false if it doesn't or path is not a valid LwM2M-path. E.g. true if path refers to a resource and the parent object has an observation, false for the inverse.

int **lwm2m_engine_stop**(struct *lwm2m_ctx* *client_ctx)

Stop the LwM2M engine.

LwM2M clients normally do not need to call this function as it is called within lwm2m_rd_client. However, if the client does not use the RD client implementation, it will need to be called manually.

Parameters

- **client_ctx** – **[in]** LwM2M context

Returns
0 for success or negative in case of error.

int **lwm2m_engine_start**(struct *lwm2m_ctx* *client_ctx)

Start the LwM2M engine.

LwM2M clients normally do not need to call this function as it is called by *lwm2m_rd_client_start()*. However, if the client does not use the RD client implementation, it will need to be called manually.

Parameters

- **client_ctx** – **[in]** LwM2M context

Returns
0 for success or negative in case of error.

void **lwm2m_acknowledge**(struct *lwm2m_ctx* *client_ctx)

Acknowledge the currently processed request with an empty ACK.

LwM2M engine by default sends piggybacked responses for requests. This function allows to send an empty ACK for a request earlier (from the application callback). The LwM2M engine will then send the actual response as a separate CON message after all callbacks are executed.

Parameters

- **client_ctx** – **[in]** LwM2M context

int **lwm2m_rd_client_start**(struct *lwm2m_ctx* *client_ctx, const char *ep_name, uint32_t flags, *lwm2m_ctx_event_cb_t* event_cb, *lwm2m_observe_cb_t* observe_cb)

Start the LwM2M RD (Registration / Discovery) Client.

The RD client sits just above the LwM2M engine and performs the necessary actions to implement the “Registration interface”. For more information see Section “Client Registration Interface” of LwM2M Technical Specification.

NOTE: *lwm2m_engine_start()* is called automatically by this function.

Parameters

- **client_ctx** – **[in]** LwM2M context
- **ep_name** – **[in]** Registered endpoint name
- **flags** – **[in]** Flags used to configure current LwM2M session.
- **event_cb** – **[in]** Client event callback function
- **observe_cb** – **[in]** Observe callback function called when an observer was added or deleted, and when a notification was acked or has timed out

Returns
0 for success, -EINPROGRESS when client is already running or negative error codes in case of failure.

int **lwm2m_rd_client_stop**(struct *lwm2m_ctx* *client_ctx, *lwm2m_ctx_event_cb_t* event_cb, bool deregister)

Stop the LwM2M RD (De-register) Client.

The RD client sits just above the LwM2M engine and performs the necessary actions to implement the "Registration interface". For more information see Section "Client Registration Interface" of the LwM2M Technical Specification.

Parameters

- **client_ctx** – **[in]** LwM2M context
- **event_cb** – **[in]** Client event callback function
- **deregister** – **[in]** True to deregister the client if registered. False to force close the connection.

Returns

0 for success or negative in case of error.

int **lwm2m_engine_pause**(void)

Suspend the LwM2M engine Thread.

Suspend LwM2M engine. Use case could be when network connection is down. LwM2M Engine indicate before it suspend by LWM2M_RD_CLIENT_EVENT_ENGINE_SUSPENDED event.

Returns

0 for success or negative in case of error.

int **lwm2m_engine_resume**(void)

Resume the LwM2M engine thread.

Resume suspended LwM2M engine. After successful resume call engine will do full registration or registration update based on suspended time. Event's LWM2M_RD_CLIENT_EVENT_REGISTRATION_COMPLETE or LWM2M_RD_CLIENT_EVENT_REG_UPDATE_COMPLETE indicate that client is connected to server.

Returns

0 for success or negative in case of error.

void **lwm2m_rd_client_update**(void)

Trigger a Registration Update of the LwM2M RD Client.

char ***lwm2m_path_log_buf**(char *buf, struct *lwm2m_obj_path* *path)

Helper function to print path objects' contents to log.

Parameters

- **buf** – **[in]** The buffer to use for formatting the string
- **path** – **[in]** The path to stringify

Returns

Resulting formatted path string

int **lwm2m_send_cb**(struct *lwm2m_ctx* *ctx, const struct *lwm2m_obj_path* path_list[], uint8_t path_list_size, *lwm2m_send_cb_t* reply_cb)

LwM2M SEND operation to given path list asynchronously with confirmation callback

Parameters

- **ctx** – LwM2M context
- **path_list** – LwM2M path struct list

- **path_list_size** – Length of path list. Max size is CONFIG_LWM2M_COMPOSITE_PATH_LIST_SIZE
- **reply_cb** – Callback triggered with confirmation state or NULL if not used

Returns
0 for success or negative in case of error.

struct *lwm2m_ctx* ***lwm2m_rd_client_ctx**(void)

Returns LwM2Mclient context

Returns
ctx LwM2M context

int **lwm2m_enable_cache**(const struct *lwm2m_obj_path* *path, struct *lwm2m_time_series_elem* *data_cache, size_t cache_len)

Enable data cache for a resource.

Application may enable caching of resource data by allocating buffer for LwM2M engine to use. Buffer must be size of struct *lwm2m_time_series_elem* times cache_len

Parameters

- **path** – LwM2M path to resource as a struct
- **data_cache** – Pointer to Data cache array
- **cache_len** – number of cached entries

Returns
0 for success or negative in case of error.

int **lwm2m_security_mode**(struct *lwm2m_ctx* *ctx)

Read security mode from selected security object instance.

This data is valid only if RD client is running.

Parameters

- **ctx** – Pointer to client context.

Returns
int Positive values are *lwm2m_security_mode_e*, negative error codes otherwise.

int **lwm2m_set_default_sockopt**(struct *lwm2m_ctx* *ctx)

Set default socket options for DTLS connections.

The engine calls this when *lwm2m_ctx::set_socketoptions* is not overwritten. You can call this from the overwritten callback to set extra options after or before defaults.

Parameters

- **ctx** – Client context

Returns
0 for success or negative in case of error.

struct **lwm2m_obj_path**

#include <*lwm2m.h*> LwM2M object path structure.

Public Members

uint16_t **obj_id**

Object ID.

uint16_t **obj_inst_id**

Object instance ID.

uint16_t **res_id**

Resource ID.

uint16_t **res_inst_id**

Resource instance ID.

uint8_t **level**

Path level (0-4).

Ex. 4 = resource instance.

struct **lwm2m_ctx**

#include <lwm2m.h> LwM2M context structure to maintain information for a single LwM2M connection.

DTLS related information

Available only when `CONFIG_LWM2M_DTLS_SUPPORT` is enabled and *lwm2m_ctx::use_dtls* is set to true.

int **tls_tag**

TLS tag is set by client as a reference used when the LwM2M engine calls tls_credential_(add|delete)

char ***desthostname**

Destination hostname.

When MBEDTLS SNI is enabled socket must be set with destination server hostname.

uint16_t **desthostnamelen**

Destination hostname length.

bool **hostname_verify**

Flag to indicate if hostname verification is enabled.

int (***load_credentials**)(struct *lwm2m_ctx* *client_ctx)

Custom load_credentials function.

Client can set load_credentials function as a way of overriding the default behavior of load_tls_credential() in lwm2m_engine.c

Public Members

struct *sockaddr* **remote_addr**

Destination address storage.

void ***processed_req**

A pointer to currently processed request, for internal LwM2M engine use.

The underlying type is `struct lwm2m_message`, but since it's declared in a private header and not exposed to the application, it's stored as a void pointer.

int (***set_socketoptions**)(struct *lwm2m_ctx* *client_ctx)

Custom socket options.

Client can override default socket options by providing a callback that is called after a socket is created and before connect.

bool **use_dtls**

Flag to indicate if context should use DTLS.

Enabled via the use of coaps:// protocol prefix in connection information. NOTE: requires `CONFIG_LWM2M_DTLS_SUPPORT`

bool **connection_suspended**

Flag to indicate that the socket connection is suspended.

With queue mode, this will tell if there is a need to reconnect.

bool **buffer_client_messages**

Flag to indicate that the client is buffering Notifications and Send messages.

True value buffer Notifications and Send messages.

int **sec_obj_inst**

Current index of Security Object used for server credentials.

int **srv_obj_inst**

Current index of Server Object used in this context.

bool **bootstrap_mode**

Flag to enable BOOTSTRAP interface.

See Section "Bootstrap Interface" of LwM2M Technical Specification for more information.

int **sock_fd**

Socket File Descriptor.

lwm2m_socket_fault_cb_t **fault_cb**

Socket fault callback.

LwM2M processing thread will call this callback in case of socket errors on receive.

lwm2m_observe_cb_t **observe_cb**

Callback for new or cancelled observations, and acknowledged or timed out notifications.

lwm2m_ctx_event_cb_t **event_cb**

Callback for client events.

uint8_t **validate_buf**[CONFIG_LWM2M_ENGINE_VALIDATION_BUFFER_SIZE]

Validation buffer.

Used as a temporary buffer to decode the resource value before validation. On successful validation, its content is copied into the actual resource buffer.

void (***set_socket_state**)(int fd, enum *lwm2m_socket_states* state)

Callback to indicate transmission states.

Client application may request LwM2M engine to indicate hints about transmission states and use that information to control various power saving modes.

struct **lwm2m_time_series_elem**

#include <*lwm2m.h*> LwM2M Time series data structure.

Public Members

time_t **t**

Cached data Unix timestamp.

union lwm2m_time_series_elem

Element value.

struct **lwm2m_objlnk**

#include <*lwm2m.h*> LWM2M Objlnk resource type structure.

Public Members

uint16_t **obj_id**

Object ID.

uint16_t **obj_inst**

Object instance ID.

struct **lwm2m_res_item**

#include <*lwm2m.h*> LwM2M resource item structure.

Value type must match the target resource as no type conversion are done and the value is just memcopied.

Following C types are used for resource types:

- BOOL is uint8_t
- U8 is uint8_t
- S8 is int8_t
- U16 is uint16_t
- S16 is int16_t

- U32 is uint32_t
- S32 is int32_t
- S64 is int64_t
- TIME is time_t
- FLOAT is double
- OBJLNK is struct *lwm2m_objlnk*
- STRING is char * and the null-terminator should be included in the size.
- OPAQUE is any binary data. When null-terminated string is written in OPAQUE resource, the terminator should not be included in size.

Public Members

struct *lwm2m_obj_path* ***path**

Pointer to LwM2M path as a struct.

void ***value**

Pointer to resource value.

uint16_t **size**

Size of the value.

For string resources, it should contain the null-terminator.

MQTT

- *Overview*
- *Sample usage*
- *Using MQTT with TLS*
- *API Reference*

Overview MQTT (Message Queuing Telemetry Transport) is an application layer protocol which works on top of the TCP/IP stack. It is a lightweight publish/subscribe messaging transport for machine-to-machine communication. For more information about the protocol itself, see http://mqtt.org/.

Zephyr provides an MQTT client library built on top of BSD sockets API. The library can be enabled with `CONFIG_MQTT_LIB` Kconfig option and is configurable at a per-client basis, with support for MQTT versions 3.1.0 and 3.1.1. The Zephyr MQTT implementation can be used with either plain sockets communicating over TCP, or with secure sockets communicating over TLS. See *BSD Sockets* for more information about Zephyr sockets.

MQTT clients require an MQTT server to connect to. Such a server, called an MQTT Broker, is responsible for managing client subscriptions and distributing messages published by clients. There are many implementations of MQTT brokers, one of them being Eclipse Mosquitto. See https://mosquitto.org/ for more information about the Eclipse Mosquitto project.

Sample usage To create an MQTT client, a client context structure and buffers need to be defined:

```
/* Buffers for MQTT client. */
static uint8_t rx_buffer[256];
static uint8_t tx_buffer[256];

/* MQTT client context */
static struct mqtt_client client_ctx;
```

Multiple MQTT client instances can be created in the application and managed independently. Additionally, a structure for MQTT Broker address information is needed. This structure must be accessible throughout the lifespan of the MQTT client and can be shared among MQTT clients:

```
/* MQTT Broker address information. */
static struct sockaddr_storage broker;
```

An MQTT client library will notify MQTT events to the application through a callback function created to handle respective events:

```
void mqtt_evt_handler(struct mqtt_client *client,
                      const struct mqtt_evt *evt)
{
   switch (evt->type) {
      /* Handle events here. */
   }
}
```

For a list of possible events, see *API Reference*.

The client context structure needs to be initialized and set up before it can be used. An example configuration for TCP transport is shown below:

```
mqtt_client_init(&client_ctx);

/* MQTT client configuration */
client_ctx.broker = &broker;
client_ctx.evt_cb = mqtt_evt_handler;
client_ctx.client_id.utf8 = (uint8_t *)"zephyr_mqtt_client";
client_ctx.client_id.size = sizeof("zephyr_mqtt_client") - 1;
client_ctx.password = NULL;
client_ctx.user_name = NULL;
client_ctx.protocol_version = MQTT_VERSION_3_1_1;
client_ctx.transport.type = MQTT_TRANSPORT_NON_SECURE;

/* MQTT buffers configuration */
client_ctx.rx_buf = rx_buffer;
client_ctx.rx_buf_size = sizeof(rx_buffer);
client_ctx.tx_buf = tx_buffer;
client_ctx.tx_buf_size = sizeof(tx_buffer);
```

After the configuration is set up, the MQTT client can connect to the MQTT broker. Call the `mqtt_connect` function, which will create the appropriate socket, establish a TCP/TLS connection, and send an `MQTT CONNECT` message. When notified, the application should call the `mqtt_input` function to process the response received. Note, that `mqtt_input` is a non-blocking function, therefore the application should use socket `poll` to wait for the response. If the connection was successful, `MQTT_EVT_CONNACK` will be notified to the application through the callback function.

```
rc = mqtt_connect(&client_ctx);
if (rc != 0) {
   return rc;
}
```

(continues on next page)

(continued from previous page)

```
fds[0].fd = client_ctx.transport.tcp.sock;
fds[0].events = ZSOCK_POLLIN;
poll(fds, 1, 5000);

mqtt_input(&client_ctx);

if (!connected) {
   mqtt_abort(&client_ctx);
}
```

In the above code snippet, the MQTT callback function should set the `connected` flag upon a successful connection. If the connection fails at the MQTT level or a timeout occurs, the connection will be aborted, and the underlying socket closed.

After the connection is established, an application needs to call `mqtt_input` and `mqtt_live` functions periodically to process incoming data and upkeep the connection. If an MQTT message is received, an MQTT callback function will be called and an appropriate event notified.

The connection can be closed by calling the `mqtt_disconnect` function.

Zephyr provides sample code utilizing the MQTT client API. See mqtt-publisher for more information.

Using MQTT with TLS The Zephyr MQTT library can be used with TLS transport for secure communication by selecting a secure transport type (`MQTT_TRANSPORT_SECURE`) and some additional configuration information:

```
client_ctx.transport.type = MQTT_TRANSPORT_SECURE;

struct mqtt_sec_config *tls_config = &client_ctx.transport.tls.config;

tls_config->peer_verify = TLS_PEER_VERIFY_REQUIRED;
tls_config->cipher_list = NULL;
tls_config->sec_tag_list = m_sec_tags;
tls_config->sec_tag_count = ARRAY_SIZE(m_sec_tags);
tls_config->hostname = MQTT_BROKER_HOSTNAME;
```

In this sample code, the `m_sec_tags` array holds a list of tags, referencing TLS credentials that the MQTT library should use for authentication. We do not specify `cipher_list`, to allow the use of all cipher suites available in the system. We set `hostname` field to broker hostname, which is required for server authentication. Finally, we enforce peer certificate verification by setting the `peer_verify` field.

Note, that TLS credentials referenced by the `m_sec_tags` array must be registered in the system first. For more information on how to do that, refer to *secure sockets documentation*.

An example of how to use TLS with MQTT is also present in mqtt-publisher sample application.

Related code samples

AWS IoT Core MQTT
Connect to AWS IoT Core and publish messages using MQTT.

MQTT publisher
Send MQTT PUBLISH messages to an MQTT server.

Microsoft Azure IoT Hub MQTT
Connect to Azure IoT Hub and publish messages using MQTT.

Secure MQTT Sensor/Actuator
Implement an MQTT-based IoT sensor/actuator device

API Reference

group `mqtt_socket`

Since
1.14

Version
0.8.0

Defines

`MQTT_UTF8_LITERAL`(literal)
Initialize UTF-8 encoded string from C literal string.

Use it as follows:

struct *mqtt_utf8* password = MQTT_UTF8_LITERAL("my_pass");

Parameters

- `literal` – **[in]** Literal string from which to generate *mqtt_utf8* object.

Typedefs

typedef void (*`mqtt_evt_cb_t`)(struct *mqtt_client* *client, const struct *mqtt_evt* *evt)
Asynchronous event notification callback registered by the application.

Param client
[in] Identifies the client for which the event is notified.

Param evt
[in] Event description along with result and associated parameters (if any).

Enums

enum `mqtt_evt_type`
MQTT Asynchronous Events notified to the application from the module through the callback registered by the application.

Values:

enumerator `MQTT_EVT_CONNACK`
Acknowledgment of connection request.

Event result accompanying the event indicates whether the connection failed or succeeded.

enumerator MQTT_EVT_DISCONNECT

Disconnection Event.

MQTT Client Reference is no longer valid once this event is received for the client.

enumerator MQTT_EVT_PUBLISH

Publish event received when message is published on a topic client is subscribed to.

> **Note**
>
> PUBLISH event structure only contains payload size, the payload data parameter should be ignored. Payload content has to be read manually with *mqtt_read_publish_payload* function.

enumerator MQTT_EVT_PUBACK

Acknowledgment for published message with QoS 1.

enumerator MQTT_EVT_PUBREC

Reception confirmation for published message with QoS 2.

enumerator MQTT_EVT_PUBREL

Release of published message with QoS 2.

enumerator MQTT_EVT_PUBCOMP

Confirmation to a publish release message with QoS 2.

enumerator MQTT_EVT_SUBACK

Acknowledgment to a subscribe request.

enumerator MQTT_EVT_UNSUBACK

Acknowledgment to a unsubscribe request.

enumerator MQTT_EVT_PINGRESP

Ping Response from server.

enum mqtt_version

MQTT version protocol level.

Values:

enumerator MQTT_VERSION_3_1_0 = 3

Protocol level for 3.1.0.

enumerator MQTT_VERSION_3_1_1 = 4

Protocol level for 3.1.1.

enum mqtt_qos

MQTT Quality of Service types.

Values:

enumerator MQTT_QOS_0_AT_MOST_ONCE = 0x00

Lowest Quality of Service, no acknowledgment needed for published message.

enumerator MQTT_QOS_1_AT_LEAST_ONCE = 0x01

Medium Quality of Service, if acknowledgment expected for published message, duplicate messages permitted.

enumerator MQTT_QOS_2_EXACTLY_ONCE = 0x02

Highest Quality of Service, acknowledgment expected and message shall be published only once.

Message not published to interested parties unless client issues a PUBREL.

enum mqtt_conn_return_code

MQTT CONNACK return codes.

Values:

enumerator MQTT_CONNECTION_ACCEPTED = 0x00

Connection accepted.

enumerator MQTT_UNACCEPTABLE_PROTOCOL_VERSION = 0x01

The Server does not support the level of the MQTT protocol requested by the Client.

enumerator MQTT_IDENTIFIER_REJECTED = 0x02

The Client identifier is correct UTF-8 but not allowed by the Server.

enumerator MQTT_SERVER_UNAVAILABLE = 0x03

The Network Connection has been made but the MQTT service is unavailable.

enumerator MQTT_BAD_USER_NAME_OR_PASSWORD = 0x04

The data in the user name or password is malformed.

enumerator MQTT_NOT_AUTHORIZED = 0x05

The Client is not authorized to connect.

enum mqtt_suback_return_code

MQTT SUBACK return codes.

Values:

enumerator MQTT_SUBACK_SUCCESS_QoS_0 = 0x00

Subscription with QoS 0 succeeded.

enumerator MQTT_SUBACK_SUCCESS_QoS_1 = 0x01

Subscription with QoS 1 succeeded.

enumerator MQTT_SUBACK_SUCCESS_QoS_2 = 0x02

Subscription with QoS 2 succeeded.

enumerator `MQTT_SUBACK_FAILURE` = 0x80

Subscription for a topic failed.

enum `mqtt_transport_type`

MQTT transport type.

Values:

enumerator `MQTT_TRANSPORT_NON_SECURE`

Use non secure TCP transport for MQTT connection.

enumerator `MQTT_TRANSPORT_NUM`

Shall not be used as a transport type.

Indicator of maximum transport types possible.

Functions

void `mqtt_client_init`(struct *mqtt_client* *client)

Initializes the client instance.

> **Note**
>
> Shall be called to initialize client structure, before setting any client parameters and before connecting to broker.

Parameters

- `client` – **[in]** Client instance for which the procedure is requested. Shall not be NULL.

int `mqtt_connect`(struct *mqtt_client* *client)

API to request new MQTT client connection.

> **Note**
>
> This memory is assumed to be resident until mqtt_disconnect is called.

> **Note**
>
> Any subsequent changes to parameters like broker address, user name, device id, etc. have no effect once MQTT connection is established.

> **Note**
>
> Default protocol revision used for connection request is 3.1.1. Please set client.protocol_version = MQTT_VERSION_3_1_0 to use protocol 3.1.0.

> **Note**
>
> Please modify `CONFIG_MQTT_KEEPALIVE` time to override default of 1 minute.

Parameters

- **client** – **[in]** Client instance for which the procedure is requested. Shall not be NULL.

Returns

0 or a negative error code (errno.h) indicating reason of failure.

int **mqtt_publish**(struct *mqtt_client* *client, const struct *mqtt_publish_param* *param)

API to publish messages on topics.

Parameters

- **client** – **[in]** Client instance for which the procedure is requested. Shall not be NULL.
- **param** – **[in]** Parameters to be used for the publish message. Shall not be NULL.

Returns

0 or a negative error code (errno.h) indicating reason of failure.

int **mqtt_publish_qos1_ack**(struct *mqtt_client* *client, const struct *mqtt_puback_param* *param)

API used by client to send acknowledgment on receiving QoS1 publish message.

Should be called on reception of *MQTT_EVT_PUBLISH* with QoS level *MQTT_QOS_1_AT_LEAST_ONCE*.

Parameters

- **client** – **[in]** Client instance for which the procedure is requested. Shall not be NULL.
- **param** – **[in]** Identifies message being acknowledged.

Returns

0 or a negative error code (errno.h) indicating reason of failure.

int **mqtt_publish_qos2_receive**(struct *mqtt_client* *client, const struct *mqtt_pubrec_param* *param)

API used by client to send acknowledgment on receiving QoS2 publish message.

Should be called on reception of *MQTT_EVT_PUBLISH* with QoS level *MQTT_QOS_2_EXACTLY_ONCE*.

Parameters

- **client** – **[in]** Identifies client instance for which the procedure is requested. Shall not be NULL.
- **param** – **[in]** Identifies message being acknowledged.

Returns

0 or a negative error code (errno.h) indicating reason of failure.

int **mqtt_publish_qos2_release**(struct *mqtt_client* *client, const struct *mqtt_pubrel_param* *param)

API used by client to request release of QoS2 publish message.

Should be called on reception of *MQTT_EVT_PUBREC*.

Parameters

- `client` – **[in]** Client instance for which the procedure is requested. Shall not be NULL.
- `param` – **[in]** Identifies message being released.

Returns
0 or a negative error code (errno.h) indicating reason of failure.

int `mqtt_publish_qos2_complete`(struct *mqtt_client* *client, const struct *mqtt_pubcomp_param* *param)

API used by client to send acknowledgment on receiving QoS2 publish release message.

Should be called on reception of *MQTT_EVT_PUBREL*.

Parameters

- `client` – **[in]** Identifies client instance for which the procedure is requested. Shall not be NULL.
- `param` – **[in]** Identifies message being completed.

Returns
0 or a negative error code (errno.h) indicating reason of failure.

int `mqtt_subscribe`(struct *mqtt_client* *client, const struct *mqtt_subscription_list* *param)

API to request subscription of one or more topics on the connection.

Parameters

- `client` – **[in]** Identifies client instance for which the procedure is requested. Shall not be NULL.
- `param` – **[in]** Subscription parameters. Shall not be NULL.

Returns
0 or a negative error code (errno.h) indicating reason of failure.

int `mqtt_unsubscribe`(struct *mqtt_client* *client, const struct *mqtt_subscription_list* *param)

API to request unsubscription of one or more topics on the connection.

> **Note**
>
> QoS included in topic description is unused in this API.

Parameters

- `client` – **[in]** Identifies client instance for which the procedure is requested. Shall not be NULL.
- `param` – **[in]** Parameters describing topics being unsubscribed from. Shall not be NULL.

Returns
0 or a negative error code (errno.h) indicating reason of failure.

int `mqtt_ping`(struct *mqtt_client* *client)

API to send MQTT ping.

The use of this API is optional, as the library handles the connection keep-alive on it's own, see *mqtt_live*.

Parameters

- **client** – **[in]** Identifies client instance for which procedure is requested.

Returns
0 or a negative error code (errno.h) indicating reason of failure.

int **mqtt_disconnect**(struct *mqtt_client* *client)

API to disconnect MQTT connection.

Parameters

- **client** – **[in]** Identifies client instance for which procedure is requested.

Returns
0 or a negative error code (errno.h) indicating reason of failure.

int **mqtt_abort**(struct *mqtt_client* *client)

API to abort MQTT connection.

This will close the corresponding transport without closing the connection gracefully at the MQTT level (with disconnect message).

Parameters

- **client** – **[in]** Identifies client instance for which procedure is requested.

Returns
0 or a negative error code (errno.h) indicating reason of failure.

int **mqtt_live**(struct *mqtt_client* *client)

This API should be called periodically for the client to be able to keep the connection alive by sending Ping Requests if need be.

Note

Application shall ensure that the periodicity of calling this function makes it possible to respect the Keep Alive time agreed with the broker on connection. *mqtt_connect* for details on Keep Alive time.

Parameters

- **client** – **[in]** Client instance for which the procedure is requested. Shall not be NULL.

Returns
0 or a negative error code (errno.h) indicating reason of failure.

int **mqtt_keepalive_time_left**(const struct *mqtt_client* *client)

Helper function to determine when next keep alive message should be sent.

Can be used for instance as a source for `poll` timeout.

Parameters

- **client** – **[in]** Client instance for which the procedure is requested.

Returns
Time in milliseconds until next keep alive message is expected to be sent. Function will return -1 if keep alive messages are not enabled.

int **mqtt_input**(struct *mqtt_client* *client)

Receive an incoming MQTT packet.

The registered callback will be called with the packet content.

Note

In case of PUBLISH message, the payload has to be read separately with *mqtt_read_publish_payload* function. The size of the payload to read is provided in the publish event structure.

Note

This is a non-blocking call.

Parameters

- **client** – **[in]** Client instance for which the procedure is requested. Shall not be NULL.

Returns

0 or a negative error code (errno.h) indicating reason of failure.

int **mqtt_read_publish_payload**(struct *mqtt_client* *client, void *buffer, size_t length)

Read the payload of the received PUBLISH message.

This function should be called within the MQTT event handler, when MQTT PUBLISH message is notified.

Note

This is a non-blocking call.

Parameters

- **client** – **[in]** Client instance for which the procedure is requested. Shall not be NULL.
- **buffer** – **[out]** Buffer where payload should be stored.
- **length** – **[in]** Length of the buffer, in bytes.

Returns

Number of bytes read or a negative error code (errno.h) indicating reason of failure.

int **mqtt_read_publish_payload_blocking**(struct *mqtt_client* *client, void *buffer, size_t length)

Blocking version of *mqtt_read_publish_payload* function.

Parameters

- **client** – **[in]** Client instance for which the procedure is requested. Shall not be NULL.
- **buffer** – **[out]** Buffer where payload should be stored.
- **length** – **[in]** Length of the buffer, in bytes.

Returns

Number of bytes read or a negative error code (errno.h) indicating reason of failure.

int **mqtt_readall_publish_payload**(struct *mqtt_client* *client, uint8_t *buffer, size_t length)

Blocking version of *mqtt_read_publish_payload* function which runs until the required number of bytes are read.

Parameters

- **client** – **[in]** Client instance for which the procedure is requested. Shall not be NULL.
- **buffer** – **[out]** Buffer where payload should be stored.
- **length** – **[in]** Number of bytes to read.

Returns

0 if success, otherwise a negative error code (errno.h) indicating reason of failure.

struct **mqtt_utf8**

#include <mqtt.h> Abstracts UTF-8 encoded strings.

Public Members

const uint8_t ***utf8**

Pointer to UTF-8 string.

uint32_t **size**

Size of UTF string, in bytes.

struct **mqtt_binstr**

#include <mqtt.h> Abstracts binary strings.

Public Members

uint8_t ***data**

Pointer to binary stream.

uint32_t **len**

Length of binary stream.

struct **mqtt_topic**

#include <mqtt.h> Abstracts MQTT UTF-8 encoded topic that can be subscribed to or published.

Public Members

struct *mqtt_utf8* **topic**

Topic on to be published or subscribed to.

uint8_t **qos**

Quality of service requested for the subscription.

mqtt_qos for details.

struct **mqtt_publish_message**

#include <*mqtt.h*> Parameters for a publish message.

Public Members

struct *mqtt_topic* **topic**

Topic on which data was published.

struct *mqtt_binstr* **payload**

Payload on the topic published.

struct **mqtt_connack_param**

#include <*mqtt.h*> Parameters for a connection acknowledgment (CONNACK).

Public Members

uint8_t **session_present_flag**

The Session Present flag enables a Client to establish whether the Client and Server have a consistent view about whether there is already stored Session state.

enum *mqtt_conn_return_code* **return_code**

The appropriate non-zero Connect return code indicates if the Server is unable to process a connection request for some reason.

struct **mqtt_puback_param**

#include <*mqtt.h*> Parameters for MQTT publish acknowledgment (PUBACK).

Public Members

uint16_t **message_id**

Message id of the PUBLISH message being acknowledged.

struct **mqtt_pubrec_param**

#include <*mqtt.h*> Parameters for MQTT publish receive (PUBREC).

Public Members

uint16_t **message_id**

Message id of the PUBLISH message being acknowledged.

struct **mqtt_pubrel_param**

#include <*mqtt.h*> Parameters for MQTT publish release (PUBREL).

Public Members

uint16_t **message_id**

Message id of the PUBREC message being acknowledged.

struct **mqtt_pubcomp_param**

#include <*mqtt.h*> Parameters for MQTT publish complete (PUBCOMP).

Public Members

uint16_t **message_id**

Message id of the PUBREL message being acknowledged.

struct **mqtt_suback_param**

#include <*mqtt.h*> Parameters for MQTT subscription acknowledgment (SUBACK).

Public Members

uint16_t **message_id**

Message id of the SUBSCRIBE message being acknowledged.

struct *mqtt_binstr* **return_codes**

Return codes indicating maximum QoS level granted for each topic in the subscription list.

struct **mqtt_unsuback_param**

#include <*mqtt.h*> Parameters for MQTT unsubscribe acknowledgment (UNSUBACK).

Public Members

uint16_t **message_id**

Message id of the UNSUBSCRIBE message being acknowledged.

struct **mqtt_publish_param**

#include <*mqtt.h*> Parameters for a publish message (PUBLISH).

Public Members

struct *mqtt_publish_message* **message**

Messages including topic, QoS and its payload (if any) to be published.

uint16_t **message_id**

Message id used for the publish message.

Redundant for QoS 0.

uint8_t **dup_flag**

Duplicate flag.

If 1, it indicates the message is being retransmitted. Has no meaning with QoS 0.

uint8_t **retain_flag**

Retain flag.

If 1, the message shall be stored persistently by the broker.

struct **mqtt_subscription_list**

#include <mqtt.h> List of topics in a subscription request.

Public Members

struct *mqtt_topic* ***list**

Array containing topics along with QoS for each.

uint16_t **list_count**

Number of topics in the subscription list.

uint16_t **message_id**

Message id used to identify subscription request.

union **mqtt_evt_param**

#include <mqtt.h> Defines event parameters notified along with asynchronous events to the application.

Public Members

struct *mqtt_connack_param* **connack**

Parameters accompanying MQTT_EVT_CONNACK event.

struct *mqtt_publish_param* **publish**

Parameters accompanying MQTT_EVT_PUBLISH event.

> **Note**
>
> PUBLISH event structure only contains payload size, the payload data parameter should be ignored. Payload content has to be read manually with *mqtt_read_publish_payload* function.

struct *mqtt_puback_param* **puback**

Parameters accompanying MQTT_EVT_PUBACK event.

struct *mqtt_pubrec_param* **pubrec**

Parameters accompanying MQTT_EVT_PUBREC event.

struct *mqtt_pubrel_param* **pubrel**
Parameters accompanying MQTT_EVT_PUBREL event.

struct *mqtt_pubcomp_param* **pubcomp**
Parameters accompanying MQTT_EVT_PUBCOMP event.

struct *mqtt_suback_param* **suback**
Parameters accompanying MQTT_EVT_SUBACK event.

struct *mqtt_unsuback_param* **unsuback**
Parameters accompanying MQTT_EVT_UNSUBACK event.

struct **mqtt_evt**
#include <mqtt.h> Defines MQTT asynchronous event notified to the application.

Public Members

enum *mqtt_evt_type* **type**
Identifies the event.

union *mqtt_evt_param* **param**
Contains parameters (if any) accompanying the event.

int **result**
Event result.
0 or a negative error code (errno.h) indicating reason of failure.

struct **mqtt_sec_config**
#include <mqtt.h> TLS configuration for secure MQTT transports.

Public Members

int **peer_verify**
Indicates the preference for peer verification.

uint32_t **cipher_count**
Indicates the number of entries in the cipher list.

const int ***cipher_list**
Indicates the list of ciphers to be used for the session.
May be NULL to use the default ciphers.

uint32_t **sec_tag_count**
Indicates the number of entries in the sec tag list.

const *sec_tag_t* ***sec_tag_list**

Indicates the list of security tags to be used for the session.

const char ***hostname**

Peer hostname for ceritificate verification.

May be NULL to skip hostname verification.

int **cert_nocopy**

Indicates the preference for copying certificates to the heap.

struct **mqtt_transport**

#include <*mqtt.h*> MQTT transport specific data.

Public Members

enum *mqtt_transport_type* **type**

Transport type selection for client instance.

mqtt_transport_type for possible values. MQTT_TRANSPORT_MAX is not a valid type.

int **sock**

Socket descriptor.

struct *mqtt_transport* **tcp**

TCP socket transport for MQTT.

union mqtt_transport

Use either unsecured TCP or secured TLS transport.

struct **mqtt_internal**

#include <*mqtt.h*> MQTT internal state.

Public Members

struct sys_mutex **mutex**

Internal.

Mutex to protect access to the client instance.

uint32_t **last_activity**

Internal.

Wall clock value (in milliseconds) of the last activity that occurred. Needed for periodic PING.

uint32_t **state**

Internal.

Client's state in the connection.

uint32_t **rx_buf_datalen**

Internal.

Packet length read so far.

uint32_t **remaining_payload**

Internal.

Remaining payload length to read.

struct **mqtt_client**

#include <*mqtt.h*> MQTT Client definition to maintain information relevant to the client.

Public Members

struct *mqtt_internal* **internal**

MQTT client internal state.

struct *mqtt_transport* **transport**

MQTT transport configuration and data.

struct *mqtt_utf8* **client_id**

Unique client identification to be used for the connection.

const void ***broker**

Broker details, for example, address, port.

Address type should be compatible with transport used.

struct *mqtt_utf8* ***user_name**

User name (if any) to be used for the connection.

NULL indicates no user name.

struct *mqtt_utf8* ***password**

Password (if any) to be used for the connection.

Note that if password is provided, user name shall also be provided. NULL indicates no password.

struct *mqtt_topic* ***will_topic**

Will topic and QoS.

Can be NULL.

struct *mqtt_utf8* ***will_message**

Will message.

Can be NULL. Non NULL value valid only if will topic is not NULL.

mqtt_evt_cb_t **evt_cb**

Application callback registered with the module to get MQTT events.

uint8_t *rx_buf
: Receive buffer used for MQTT packet reception in RX path.

uint32_t rx_buf_size
: Size of receive buffer.

uint8_t *tx_buf
: Transmit buffer used for creating MQTT packet in TX path.

uint32_t tx_buf_size
: Size of transmit buffer.

uint16_t keepalive
: Keepalive interval for this client in seconds.

  Default is CONFIG_MQTT_KEEPALIVE.

uint8_t protocol_version
: MQTT protocol version.

int8_t unacked_ping
: Unanswered PINGREQ count on this connection.

uint8_t will_retain
: Will retain flag, 1 if will message shall be retained persistently.

uint8_t clean_session
: Clean session flag indicating a fresh (1) or a retained session (0).

  Default is CONFIG_MQTT_CLEAN_SESSION.

void *user_data
: User specific opaque data.

MQTT-SN

- *Overview*
- *Sample usage*
- *Deviations from the standard*
- *API Reference*

Overview MQTT-SN is a variant of the well-known MQTT protocol - see *MQTT*.

In contrast to MQTT, MQTT-SN does not require a TCP transport, but is designed to be used over any message-based transport. Originally, it was mainly created with ZigBee in mind, but others like Bluetooth, UDP or even a UART can be used just as well.

Zephyr provides an MQTT-SN client library built on top of BSD sockets API. The library can be enabled with `CONFIG_MQTT_SN_LIB` Kconfig option and is configurable at a per-client basis, with

support for MQTT-SN version 1.2. The Zephyr MQTT-SN implementation can be used with any message-based transport, but support for UDP is already built-in.

MQTT-SN clients require an MQTT-SN gateway to connect to. These gateways translate between MQTT-SN and MQTT. The Eclipse Paho project offers an implementation of a MQTT-SN gateway, but others are available too. https://www.eclipse.org/paho/index.php?page=components/mqtt-sn-transparent-gateway/index.php

The MQTT-SN spec v1.2 can be found here: https://www.oasis-open.org/committees/download.php/66091/MQTT-SN_spec_v1.2.pdf

Sample usage To create an MQTT-SN client, a client context structure and buffers need to be defined:

```
/* Buffers for MQTT client. */
static uint8_t rx_buffer[256];
static uint8_t tx_buffer[256];

/* MQTT-SN client context */
static struct mqtt_sn_client client;
```

Multiple MQTT-SN client instances can be created in the application and managed independently. Additionally, a structure for the transport is needed as well. The library already comes with an example implementation for UDP.

```
/* MQTT Broker address information. */
static struct mqtt_sn_transport tp;
```

The MQTT-SN library will inform clients about certain events using a callback.

```
static void evt_cb(struct mqtt_sn_client *client,
                   const struct mqtt_sn_evt *evt)
{
   switch(evt->type) {
   {
      /* Handle events here. */
   }
}
```

For a list of possible events, see *API Reference*.

The client context structure needs to be initialized and set up before it can be used. An example configuration for UDP transport is shown below:

```
struct mqtt_sn_data client_id = MQTT_SN_DATA_STRING_LITERAL("ZEPHYR");
struct sockaddr_in gateway = {0};

uint8_t tx_buf[256];
uint8_t rx_buf[256];

mqtt_sn_transport_udp_init(&tp, (struct sockaddr*)&gateway, sizeof((gateway)));

mqtt_sn_client_init(&client, &client_id, &tp.tp, evt_cb, tx_buf, sizeof(tx_buf), rx_buf,␣
↪sizeof(rx_buf));
```

After the configuration is set up, the MQTT-SN client can connect to the gateway. While the MQTT-SN protocol offers functionality to discover gateways through an advertisement mechanism, this is not implemented yet in the library.

Call the `mqtt_sn_connect` function, which will send a `CONNECT` message. The application should periodically call the `mqtt_sn_input` function to process the response received. The application does not have to call `mqtt_sn_input` if it knows that no data has been received (e.g. when using

Bluetooth). Note that `mqtt_sn_input` is a non-blocking function, if the transport struct contains a `poll` compatible function pointer. If the connection was successful, `MQTT_SN_EVT_CONNECTED` will be notified to the application through the callback function.

```
err = mqtt_sn_connect(&client, false, true);
__ASSERT(err == 0, "mqtt_sn_connect() failed %d", err);

while (1) {
        mqtt_sn_input(&client);
        if (connected) {
                mqtt_sn_publish(&client, MQTT_SN_QOS_0, &topic_p, false, &pubdata);
        }
        k_sleep(K_MSEC(500));
}
```

In the above code snippet, the event handler function should set the `connected` flag upon a successful connection. If the connection fails at the MQTT level or a timeout occurs, the connection will be aborted.

After the connection is established, an application needs to call `mqtt_input` function periodically to process incoming data. Connection upkeep, on the other hand, is done automatically using a k_work item. If a MQTT message is received, an MQTT callback function will be called and an appropriate event notified.

The connection can be closed by calling the `mqtt_sn_disconnect` function. This has no effect on the transport, however. If you want to close the transport (e.g. the socket), call `mqtt_sn_client_deinit`, which will deinit the transport as well.

Zephyr provides sample code utilizing the MQTT-SN client API. See mqtt-sn-publisher for more information.

Deviations from the standard Certain parts of the protocol are not yet supported in the library.

- Pre-defined topic IDs
- QoS -1 - it's most useful with predefined topics
- Gateway discovery using ADVERTISE, SEARCHGW and GWINFO messages.
- Setting the will topic and message after the initial connect
- Forwarder Encapsulation

Related code samples

MQTT-SN publisher
Send MQTT-SN PUBLISH messages to an MQTT-SN gateway.

API Reference

group `mqtt_sn_socket`

Defines

MQTT_SN_DATA_STRING_LITERAL(literal)

Initialize memory buffer from C literal string.

Use it as follows:

struct *mqtt_sn_data* topic = MQTT_SN_DATA_STRING_LITERAL("/zephyr");

Parameters

- `literal` – **[in]** Literal string from which to generate *mqtt_sn_data* object.

MQTT_SN_DATA_BYTES(...)

Initialize memory buffer from single bytes.

Use it as follows:

struct *mqtt_sn_data* data = *MQTT_SN_DATA_BYTES(0x13, 0x37)*;

Typedefs

typedef void (*`mqtt_sn_evt_cb_t`)(struct *mqtt_sn_client* *client, const struct *mqtt_sn_evt* *evt)

Asynchronous event notification callback registered by the application.

Param client
[in] Identifies the client for which the event is notified.

Param evt
[in] Event description along with result and associated parameters (if any).

Enums

enum `mqtt_sn_qos`

Quality of Service.

QoS 0-2 work the same as basic MQTT, QoS -1 is an MQTT-SN addition. QOS -1 is not supported yet.

Values:

enumerator `MQTT_SN_QOS_0`

QOS 0.

enumerator `MQTT_SN_QOS_1`

QOS 1.

enumerator `MQTT_SN_QOS_2`

QOS 2.

enumerator `MQTT_SN_QOS_M1`

QOS -1.

enum `mqtt_sn_topic_type`

MQTT-SN topic types.

Values:

enumerator MQTT_SN_TOPIC_TYPE_NORMAL

Normal topic.

It allows usage of any valid UTF-8 string as a topic name.

enumerator MQTT_SN_TOPIC_TYPE_PREDEF

Pre-defined topic.

It allows usage of a two-byte identifier representing a topic name for which the corresponding topic name is known in advance by both the client and the gateway/server.

enumerator MQTT_SN_TOPIC_TYPE_SHORT

Short topic.

It allows usage of a two-byte string as a topic name.

enum mqtt_sn_return_code

MQTT-SN return codes.

Values:

enumerator MQTT_SN_CODE_ACCEPTED = 0x00

Accepted.

enumerator MQTT_SN_CODE_REJECTED_CONGESTION = 0x01

Rejected: congestion.

enumerator MQTT_SN_CODE_REJECTED_TOPIC_ID = 0x02

Rejected: Invalid Topic ID.

enumerator MQTT_SN_CODE_REJECTED_NOTSUP = 0x03

Rejected: Not Supported.

enum mqtt_sn_evt_type

Event types that can be emitted by the library.

Values:

enumerator MQTT_SN_EVT_CONNECTED

Connected to a gateway.

enumerator MQTT_SN_EVT_DISCONNECTED

Disconnected.

enumerator MQTT_SN_EVT_ASLEEP

Entered ASLEEP state.

enumerator MQTT_SN_EVT_AWAKE

Entered AWAKE state.

enumerator MQTT_SN_EVT_PUBLISH

Received a PUBLISH message.

enumerator MQTT_SN_EVT_PINGRESP

Received a PINGRESP.

Functions

int **mqtt_sn_client_init**(struct *mqtt_sn_client* *client, const struct *mqtt_sn_data* *client_id, struct *mqtt_sn_transport* *transport, *mqtt_sn_evt_cb_t* evt_cb, void *tx, size_t txsz, void *rx, size_t rxsz)

Initialize a client.

Parameters

- **client** – The MQTT-SN client to initialize.
- **client_id** – The ID to be used by the client.
- **transport** – The transport to be used by the client.
- **evt_cb** – The event callback function for the client.
- **tx** – Pointer to the transmit buffer.
- **txsz** – Size of the transmit buffer.
- **rx** – Pointer to the receive buffer.
- **rxsz** – Size of the receive buffer.

Returns

0 or a negative error code (errno.h) indicating reason of failure.

void **mqtt_sn_client_deinit**(struct *mqtt_sn_client* *client)

Deinitialize the client.

This removes all topics and publishes, and also de-inits the transport.

Parameters

- **client** – The MQTT-SN client to deinitialize.

int **mqtt_sn_connect**(struct *mqtt_sn_client* *client, bool will, bool clean_session)

Connect the client.

Parameters

- **client** – The MQTT-SN client to connect.
- **will** – Flag indicating if a Will message should be sent.
- **clean_session** – Flag indicating if a clean session should be started.

Returns

0 or a negative error code (errno.h) indicating reason of failure.

int **mqtt_sn_disconnect**(struct *mqtt_sn_client* *client)

Disconnect the client.

Parameters

- **client** – The MQTT-SN client to disconnect.

Returns

0 or a negative error code (errno.h) indicating reason of failure.

int **mqtt_sn_sleep**(struct *mqtt_sn_client* *client, uint16_t duration)

Set the client into sleep state.

Parameters

- **client** – The MQTT-SN client to be put to sleep.
- **duration** – Sleep duration (in seconds).

Returns

0 on success, negative errno code on failure.

int **mqtt_sn_subscribe**(struct *mqtt_sn_client* *client, enum *mqtt_sn_qos* qos, struct *mqtt_sn_data* *topic_name)

Subscribe to a given topic.

Parameters

- **client** – The MQTT-SN client that should subscribe.
- **qos** – The desired quality of service for the subscription.
- **topic_name** – The name of the topic to subscribe to.

Returns

0 or a negative error code (errno.h) indicating reason of failure.

int **mqtt_sn_unsubscribe**(struct *mqtt_sn_client* *client, enum *mqtt_sn_qos* qos, struct *mqtt_sn_data* *topic_name)

Unsubscribe from a topic.

Parameters

- **client** – The MQTT-SN client that should unsubscribe.
- **qos** – The quality of service used when subscribing.
- **topic_name** – The name of the topic to unsubscribe from.

Returns

0 or a negative error code (errno.h) indicating reason of failure.

int **mqtt_sn_publish**(struct *mqtt_sn_client* *client, enum *mqtt_sn_qos* qos, struct *mqtt_sn_data* *topic_name, bool retain, struct *mqtt_sn_data* *data)

Publish a value.

If the topic is not yet registered with the gateway, the library takes care of it.

Parameters

- **client** – The MQTT-SN client that should publish.
- **qos** – The desired quality of service for the publish.
- **topic_name** – The name of the topic to publish to.
- **retain** – Flag indicating if the message should be retained by the broker.
- **data** – The data to be published.

Returns

0 or a negative error code (errno.h) indicating reason of failure.

int **mqtt_sn_input**(struct *mqtt_sn_client* *client)

Check the transport for new incoming data.

Call this function periodically, or if you have good reason to believe there is any data. If the client's transport struct contains a poll-function, this function is non-blocking.

Parameters

- **client** – The MQTT-SN client to check for incoming data.

Returns
0 or a negative error code (errno.h) indicating reason of failure.

int **mqtt_sn_get_topic_name**(struct *mqtt_sn_client* *client, uint16_t id, struct *mqtt_sn_data* *topic_name)

Get topic name by topic ID.

Parameters

- **client** – **[in]** The MQTT-SN client that uses this topic.
- **id** – **[in]** Topic identifier.
- **topic_name** – **[out]** Will be assigned to topic name.

Returns
0 on success, -ENOENT if topic ID doesn't exist, or -EINVAL on invalid arguments.

struct **mqtt_sn_data**

#include <mqtt_sn.h> Abstracts memory buffers.

Public Members

const uint8_t ***data**

Pointer to data.

uint16_t **size**

Size of data, in bytes.

union **mqtt_sn_evt_param**

#include <mqtt_sn.h> Event metadata.

Public Members

struct *mqtt_sn_data* **data**

The payload data associated with the event.

enum *mqtt_sn_topic_type* **topic_type**

The type of topic for the event.

uint16_t **topic_id**

The identifier for the topic of the event.

struct *mqtt_sn_evt_param* **publish**

Structure holding publish event details.

struct **mqtt_sn_evt**

#include <mqtt_sn.h> MQTT-SN event structure to be handled by the event callback.

Public Members

enum *mqtt_sn_evt_type* **type**

Event type.

union *mqtt_sn_evt_param* **param**

Event parameters.

struct **mqtt_sn_transport**

#include <*mqtt_sn.h*> Structure to describe an MQTT-SN transport.

MQTT-SN does not require transports to be reliable or to hold a connection. Transports just need to be frame-based, so you can use UDP, ZigBee, or even a simple UART, given some kind of framing protocol is used.

Public Members

int (***init**)(struct *mqtt_sn_transport* *transport)

Will be called once on client init to initialize the transport.

Use this to open sockets or similar. May be NULL.

void (***deinit**)(struct *mqtt_sn_transport* *transport)

Will be called on client deinit.

Use this to close sockets or similar. May be NULL.

int (***msg_send**)(struct *mqtt_sn_client* *client, void *buf, size_t sz)

Will be called by the library when it wants to send a message.

ssize_t (***recv**)(struct *mqtt_sn_client* *client, void *buffer, size_t length)

Will be called by the library when it wants to receive a message.

Implementations should follow recv conventions.

int (***poll**)(struct *mqtt_sn_client* *client)

Check if incoming data is available.

If *poll()* returns a positive number, recv must not block.

May be NULL, but recv should not block then either.

Return

Positive number if data is available, or zero if there is none. Negative values signal errors.

struct **mqtt_sn_client**

#include <*mqtt_sn.h*> Structure describing an MQTT-SN client.

Public Members

struct *mqtt_sn_data* **client_id**

1-23 character unique client ID

struct *mqtt_sn_data* **will_topic**
Topic for Will message.
Must be initialized before connecting with will=true

struct *mqtt_sn_data* **will_msg**
Will message.
Must be initialized before connecting with will=true

enum *mqtt_sn_qos* **will_qos**
Quality of Service for the Will message.

bool **will_retain**
Flag indicating if the will message should be retained by the broker.

struct *mqtt_sn_transport* ***transport**
Underlying transport to be used by the client.

struct *net_buf_simple* **tx**
Buffer for outgoing data.

struct *net_buf_simple* **rx**
Buffer for incoming data.

mqtt_sn_evt_cb_t **evt_cb**
Event callback.

uint16_t **next_msg_id**
Message ID for the next message to be sent.

sys_slist_t **publish**
List of pending publish messages.

sys_slist_t **topic**
List of registered topics.

int **state**
Current state of the MQTT-SN client.

int64_t **last_ping**
Timestamp of the last ping request.

uint8_t **ping_retries**
Number of retries for failed ping attempts.

struct *k_work_delayable* **process_work**
Delayable work structure for processing MQTT-SN events.

Precision Time Protocol (PTP)

- *Overview*
- *Supported features*
- *Supported Management messages*
- *Enabling the stack*
- *Testing*
- *API Reference*

Overview PTP is a network protocol implemented in the application layer, used to synchronize clocks in a computer network. It's accurate up to less than a microsecond. The stack supports the protocol and procedures as defined in the IEEE 1588-2019 standard (IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems). It has multiple profiles, and can be implemented on top of L2 (Ethernet) or L3 (UDP/IPv4 or UDP/IPv6). Its accuracy is achieved by using hardware timestamping of the protocol packets.

Zephyr's implementation of PTP stack consist following items:

- PTP stack thread that handles incoming messages and events
- Integration with ptp_clock driver
- PTP stack initialization executed during system init

The implementation automatically creates PTP Ports (each PTP Port corresponds to unique interface).

Supported features Implementation of the stack doesn't support all features specified in the standard. In the table below all supported features are listed.

Table 33: Supported features

Feature	Supported
Ordinary Clock	yes
Boundary Clock	yes
Transparent Clock	
Management Node	
End to end delay mechanism	yes
Peer to peer delay mechanism	
Multicast operation mode	
Hybrid operation mode	
Unicast operation mode	
Non-volatile storage	
UDP IPv4 transport protocol	yes
UDP IPv6 transport protocol	yes
IEEE 802.3 (Ethernet) transport protocol	
Hardware timestamping	yes
Software timestamping	
TIME_RECEIVER_ONLY PTP Instance	yes
TIME_TRANSMITTER_ONLY PTP Instance	

Supported Management messages Based on Table 59 from section 15.5.2.3 of the IEEE 1588-2019 following management TLVs are supported:

Table 34: Supported management message's IDs

Management_ID	Management_ID name	Allowed actions
0x0000	NULL_PTP_MANAGEMENT	GET SET COMMAND
0x0001	CLOCK_DESCRIPTION	GET
0x0002	USER_DESCRIPTION	GET
0x0003	SAVE_IN_NON_VOLATILE_STORAGE	•
0x0004	RESET_NON_VOLATILE_STORAGE	•
0x0005	INITIALIZE	•
0x0006	FAULT_LOG	•
0x0007	FAULT_LOG_RESET	•
0x2000	DEFAULT_DATA_SET	GET
0x2001	CURRENT_DATA_SET	GET
0x2002	PARENT_DATA_SET	GET
0x2003	TIME_PROPERTIES_DATA_SET	GET
0x2004	PORT_DATA_SET	GET
0x2005	PRIORITY1	GET SET
0x2006	PRIORITY2	GET SET
0x2007	DOMAIN	GET SET
0x2008	TIME_RECEIVER_ONLY	GET SET
0x2009	LOG_ANNOUNCE_INTERVAL	GET SET
0x200A	ANNOUNCE_RECEIPT_TIMEOUT	GET SET
0x200B	LOG_SYNC_INTERVAL	GET SET
0x200C	VERSION_NUMBER	GET SET
0x200D	ENABLE_PORT	COMMAND
0x200E	DISABLE_PORT	COMMAND
0x200F	TIME	GET SET
0x2010	CLOCK_ACCURACY	GET SET
0x2011	UTC_PROPERTIES	GET SET
0x2012	TRACEBILITY_PROPERTIES	GET SET
0x2013	TIMESCALE_PROPERTIES	GET SET
0x2014	UNICAST_NEGOTIATION_ENABLE	•
0x2015	PATH_TRACE_LIST	•
0x2016	PATH_TRACE_ENABLE	•
0x2017	GRANDMASTER_CLUSTER_TABLE	•
0x2018	UNICAST_TIME_TRANSMITTER_TABLE	•
0x2019	UNICAST_TIME_TRANSMITTER_MAX_TABLE_SIZI	•

continues on next page

Table 34 – continued from previous page

Manage-ment_ID	Management_ID name	Allowed actions
0x201A	ACCEPTABLE_TIME_TRANSMITTER_TABLE	•
0x201B	ACCEPTABLE_TIME_TRANSMITTER_TABLE_ENAB	•
0x201C	ACCEPTABLE_TIME_TRANSMITTER_MAX_TABLE_	•
0x201D	ALTERNATE_TIME_TRANSMITTER	•
0x201E	ALTERNATE_TIME_OFFSET_ENABLE	•
0x201F	ALTERNATE_TIME_OFFSET_NAME	•
0x2020	ALTERNATE_TIME_OFFSET_MAX_KEY	•
0x2021	ALTERNATE_TIME_OFFSET_PROPERTIES	•
0x3000	EXTERNAL_PORT_CONFIGURATION_ENABLED	
0x3001	TIME_TRANSMITTER_ONLY	•
0x3002	HOLDOVER_UPGRADE_ENABLE	•
0x3003	EXT_PORT_CONFIG_PORT_DATA_SET	•
0x4000	TRANSPARENT_CLOCK_DEFAULT_DATA_SET	•
0x4001	TRANSPARENT_CLOCK_PORT_DATA_SET	•
0x4002	PRIMARY_DOMAIN	•
0x6000	DELAY_MECHANISM	GET
0x6001	LOG_MIN_PDELAY_REQ_INTERVAL	GET SET

Enabling the stack The following configuration option must me enabled in `prj.conf` file.

- `CONFIG_PTP`

Testing The stack has been informally tested using the Linux ptp4l daemons. The PTP sample application from the Zephyr source distribution can be used for testing.

Related code samples

PTP
Enable PTP support and monitor messages and events via logging.

API Reference

group `ptp`

Precision Time Protocol (PTP) support.

Defines

`PTP_MAJOR_VERSION`

Major PTP Version.

`PTP_MINOR_VERSION`

Minor PTP Version.

`PTP_VERSION`

PTP version IEEE-1588:2019.

TFTP Zephyr provides a simple TFTP client library that can enabled with `CONFIG_MQTT_SN_LIB` Kconfig option.

See TFTP client sample application for more information about the library usage.

Related code samples

TFTP client

Use the TFTP client library to get/put files from/to a TFTP server.

API Reference

group `tftp_client`

TFTP client error codes.

`TFTPC_SUCCESS`

Success.

`TFTPC_DUPLICATE_DATA`

Duplicate data received.

`TFTPC_BUFFER_OVERFLOW`

User buffer is too small.

`TFTPC_UNKNOWN_FAILURE`

Unknown failure.

`TFTPC_REMOTE_ERROR`

Remote server error.

TFTPC_RETRIES_EXHAUSTED
: Retries exhausted.

Defines

TFTP_BLOCK_SIZE
: RFC1350: the file is sent in fixed length blocks of 512 bytes.

  Each data packet contains one block of data, and must be acknowledged by an acknowledgment packet before the next packet can be sent. A data packet of less than 512 bytes signals termination of a transfer.

TFTP_HEADER_SIZE
: RFC1350: For non-request TFTP message, the header contains 2-byte operation code plus 2-byte block number or error code.

TFTPC_MAX_BUF_SIZE
: Maximum amount of data that can be sent or received.

Typedefs

typedef void (*tftp_callback_t)(const struct *tftp_evt* *evt)
: TFTP event notification callback registered by the application.

  Param evt
  [in] Event description along with result and associated parameters (if any).

Enums

enum tftp_evt_type
: TFTP Asynchronous Events notified to the application from the module through the callback registered by the application.

  Values:

  enumerator TFTP_EVT_DATA
  : DATA event when data is received from remote server.

    Note

    DATA event structure contains payload data and size.

  enumerator TFTP_EVT_ERROR
  : ERROR event when error is received from remote server.

    Note

    ERROR event structure contains error code and message.

Functions

int **tftp_get**(struct *tftpc* *client, const char *remote_file, const char *mode)

This function gets data from a "file" on the remote server.

> **Note**
>
> This function blocks until the transfer is completed or network error happens. The integrity of the `client` structure must be ensured until the function returns.

Parameters

- **client** – Client information of type *tftpc*.
- **remote_file** – Name of the remote file to get.
- **mode** – TFTP Client "mode" setting.

Return values

- **The** – size of data being received if the operation completed successfully.
- **TFTPC_BUFFER_OVERFLOW** – if the file is larger than the user buffer.
- **TFTPC_REMOTE_ERROR** – if the server failed to process our request.
- **TFTPC_RETRIES_EXHAUSTED** – if the client timed out waiting for server.
- **-EINVAL** – if `client` is NULL.

int **tftp_put**(struct *tftpc* *client, const char *remote_file, const char *mode, const uint8_t *user_buf, uint32_t user_buf_size)

This function puts data to a "file" on the remote server.

> **Note**
>
> This function blocks until the transfer is completed or network error happens. The integrity of the `client` structure must be ensured until the function returns.

Parameters

- **client** – Client information of type *tftpc*.
- **remote_file** – Name of the remote file to put.
- **mode** – TFTP Client "mode" setting.
- **user_buf** – Data buffer containing the data to put.
- **user_buf_size** – Length of the data to put.

Return values

- **The** – size of data being sent if the operation completed successfully.
- **TFTPC_REMOTE_ERROR** – if the server failed to process our request.
- **TFTPC_RETRIES_EXHAUSTED** – if the client timed out waiting for server.
- **-EINVAL** – if `client` or `user_buf` is NULL or if `user_buf_size` is zero.

struct **tftp_data_param**

#include <tftp.h> Parameters for data event.

Public Members

uint8_t *data_ptr
Pointer to binary data.

uint32_t len
Length of binary data.

struct tftp_error_param
#include <*tftp.h*> Parameters for error event.

Public Members

char *msg
Error message.

int code
Error code.

union tftp_evt_param
#include <*tftp.h*> Defines event parameters notified along with asynchronous events to the application.

Public Members

struct *tftp_data_param* data
Parameters accompanying TFTP_EVT_DATA event.

struct *tftp_error_param* error
Parameters accompanying TFTP_EVT_ERROR event.

struct tftp_evt
#include <*tftp.h*> Defines TFTP asynchronous event notified to the application.

Public Members

enum *tftp_evt_type* type
Identifies the event.

union *tftp_evt_param* param
Contains parameters (if any) accompanying the event.

struct tftpc
#include <*tftp.h*> TFTP client definition to maintain information relevant to the client.

> **Note**
>
> Application must initialize `server` and `callback` before calling GET or PUT API with the `tftpc` structure.

Public Members

struct *sockaddr* **server**

Socket address pointing to the remote TFTP server.

tftp_callback_t **callback**

Event notification callback.

No notification if NULL

uint8_t **tftp_buf**[(512 + 4)]

Buffer for internal usage.

Network System Management

Network Configuration Library

- *Overview*
- *Sample usage*
- *API Reference*

Overview The network configuration library sets up networking devices in a semi-automatic way during the system boot, based on user-supplied Kconfig options.

The following Kconfig options affect how configuration library will setup the system:

Table 35: Kconfig options for network configuration library

Option name	Description
CONFIG_NET_CONFIG_SETTINGS	This option controls whether the network system is configured or initialized at all. If not set, then the config library is not used for initialization and the application needs to do all the network related configuration itself. If this option is set, then the user can optionally configure static IP addresses to be set to the first network interface in the system. Typically setting static IP addresses is only usable in testing and should not be used in production code. See the config library Kconfig file subsys/net/lib/config/Kconfig for specific options to set the static IP addresses.
CONFIG_NET_CONFIG_AUTO_INIT	The networking system is automatically configured when the device is started.
CONFIG_NET_CONFIG_INIT_TIMEOUT	This tells how long to wait for the networking to be ready and available. If for example IPv4 address from DHCPv4 is not received within this limit, then a call to net_config_init() will return error during the device startup.
CONFIG_NET_CONFIG_NEED_IPV4	The network application needs IPv4 support to function properly. This option makes sure the network application is initialized properly in order to use IPv4. If CONFIG_NET_IPV4 is not enabled, then setting this option will automatically enable IPv4.
CONFIG_NET_CONFIG_NEED_IPV6	The network application needs IPv6 support to function properly. This option makes sure the network application is initialized properly in order to use IPv6. If CONFIG_NET_IPV6 is not enabled, then setting this option will automatically enable IPv6.
CONFIG_NET_CONFIG_NEED_IPV6_ROUTER	If IPv6 is enabled, then this option tells that the network application needs IPv6 router to exists before continuing. This means in practice that the application wants to wait until it receives IPv6 router advertisement message before continuing.
CONFIG_NET_CONFIG_MY_IPV6_ADDR	Local static IPv6 address assigned to the default network interface.
CONFIG_NET_CONFIG_PEER_IPV6_ADDR	Peer static IPv6 address. This is mainly useful in testing setups where the application can connect to a pre-defined host.
CONFIG_NET_CONFIG_MY_IPV4_ADDR	Local static IPv4 address assigned to the default network interface.
CONFIG_NET_CONFIG_MY_IPV4_NETMASK	Static IPv4 netmask assigned to the IPv4 address.
CONFIG_NET_CONFIG_MY_IPV4_GW	Static IPv4 gateway address assigned to the default network interface.
CONFIG_NET_CONFIG_PEER_IPV4_ADDR	Peer static IPv4 address. This is mainly useful in testing setups where the application can connect to a pre-defined host.

Sample usage If CONFIG_NET_CONFIG_AUTO_INIT is set, then the configuration library is automatically enabled and run during the device boot. In this case, the library will call net_config_init() automatically and the application does not need to do any network configuration.

If you want to use the network configuration library but without automatic initialization, you can

call `net_config_init()` manually. The `flags` parameter can be used to give hints to the library about what kind of functionality the application wishes to have before the actual application starts.

Related code samples

zperf: Network Traffic Generator
Use the zperf shell utility to evaluate network bandwidth.

API Reference

group `net_config`

Network configuration library.

Defines

`NET_CONFIG_NEED_ROUTER`

Application needs routers to be set so that connectivity to remote network is possible.

For IPv6 networks, this means that the device should receive IPv6 router advertisement message before continuing.

`NET_CONFIG_NEED_IPV6`

Application needs IPv6 subsystem configured and initialized.

Typically this means that the device has IPv6 address set.

`NET_CONFIG_NEED_IPV4`

Application needs IPv4 subsystem configured and initialized.

Typically this means that the device has IPv4 address set.

Functions

int **net_config_init**(const char *app_info, uint32_t flags, int32_t timeout)

Initialize this network application.

This will call *net_config_init_by_iface()* with NULL network interface.

Parameters

- **app_info** – String describing this application.
- **flags** – Flags related to services needed by the client.
- **timeout** – How long to wait the network setup before continuing the startup.

Returns

0 if ok, <0 if error.

int **net_config_init_by_iface**(struct *net_if* *iface, const char *app_info, uint32_t flags, int32_t timeout)

Initialize this network application using a specific network interface.

If network interface is set to NULL, then the default one is used in the configuration.

Parameters

- **iface** – Initialize networking using this network interface.
- **app_info** – String describing this application.
- **flags** – Flags related to services needed by the client.
- **timeout** – How long to wait the network setup before continuing the startup.

Returns

0 if ok, <0 if error.

int **net_config_init_app**(const struct *device* *dev, const char *app_info)

Initialize this network application.

If CONFIG_NET_CONFIG_AUTO_INIT is set, then this function is called automatically when the device boots. If that is not desired, unset the config option and call the function manually when the application starts.

Parameters

- **dev** – Network device to use. The function will figure out what network interface to use based on the device. If the device is NULL, then default network interface is used by the function.
- **app_info** – String describing this application.

Returns

0 if ok, <0 if error.

DHCPv4

- *Overview*
- *Sample usage*
- *API Reference*

Overview The Dynamic Host Configuration Protocol (DHCP) is a network management protocol used on IPv4 networks. A DHCPv4 server dynamically assigns an IPv4 address and other network configuration parameters to each device on a network so they can communicate with other IP networks. See this DHCP Wikipedia article for a detailed overview of how DHCP works.

Note that Zephyr supports both DHCPv4 client and server functionality.

Sample usage See dhcpv4-client sample application for details.

Related code samples

DHCPv4 client
Start a DHCPv4 client to obtain an IPv4 address from a DHCPv4 server.

API Reference

group dhcpv4

DHCPv4.

Typedefs

typedef void (*net_dhcpv4_option_callback_handler_t)(struct net_dhcpv4_option_callback *cb, size_t length, enum *net_dhcpv4_msg_type* msg_type, struct *net_if* *iface)

Define the application callback handler function signature.

Note: cb pointer can be used to retrieve private data through *CONTAINER_OF()* if original struct net_dhcpv4_option_callback is stored in another private structure.

Param cb
Original struct net_dhcpv4_option_callback owning this handler

Param length
The length of data returned by the server. If this is greater than cb->max_length, only cb->max_length bytes will be available in cb->data

Param msg_type
Type of DHCP message that triggered the callback

Param iface
The interface on which the DHCP message was received

Enums

enum net_dhcpv4_msg_type

DHCPv4 message types.

These enumerations represent RFC2131 defined msy type codes, hence they should not be renumbered.

Additions, removald and reorders in this definition must be reflected within corresponding changes to net_dhcpv4_msg_type_name.

Values:

enumerator NET_DHCPV4_MSG_TYPE_DISCOVER = 1

Discover message.

enumerator NET_DHCPV4_MSG_TYPE_OFFER = 2

Offer message.

enumerator NET_DHCPV4_MSG_TYPE_REQUEST = 3

Request message.

enumerator NET_DHCPV4_MSG_TYPE_DECLINE = 4
Decline message.

enumerator NET_DHCPV4_MSG_TYPE_ACK = 5
Acknowledge message.

enumerator NET_DHCPV4_MSG_TYPE_NAK = 6
Negative acknowledge message.

enumerator NET_DHCPV4_MSG_TYPE_RELEASE = 7
Release message.

enumerator NET_DHCPV4_MSG_TYPE_INFORM = 8
Inform message.

Functions

static inline void net_dhcpv4_init_option_callback(struct net_dhcpv4_option_callback *callback, *net_dhcpv4_option_callback_handler_t* handler, uint8_t option, void *data, size_t max_length)

Helper to initialize a struct net_dhcpv4_option_callback properly.

Parameters

- callback – A valid Application's callback structure pointer.
- handler – A valid handler function pointer.
- option – The DHCP option the callback responds to.
- data – A pointer to a buffer for max_length bytes.
- max_length – The maximum length of the data returned.

int net_dhcpv4_add_option_callback(struct net_dhcpv4_option_callback *cb)

Add an application callback.

Parameters

- cb – A valid application's callback structure pointer.

Returns

0 if successful, negative errno code on failure.

int net_dhcpv4_remove_option_callback(struct net_dhcpv4_option_callback *cb)

Remove an application callback.

Parameters

- cb – A valid application's callback structure pointer.

Returns

0 if successful, negative errno code on failure.

static inline void **net_dhcpv4_init_option_vendor_callback**(struct net_dhcpv4_option_callback *callback, *net_dhcpv4_option_callback_handler_t* handler, uint8_t option, void *data, size_t max_length)

Helper to initialize a struct net_dhcpv4_option_callback for encapsulated vendor-specific options properly.

Parameters

- **callback** – A valid Application's callback structure pointer.
- **handler** – A valid handler function pointer.
- **option** – The DHCP encapsulated vendor-specific option the callback responds to.
- **data** – A pointer to a buffer for max_length bytes.
- **max_length** – The maximum length of the data returned.

int **net_dhcpv4_add_option_vendor_callback**(struct net_dhcpv4_option_callback *cb)

Add an application callback for encapsulated vendor-specific options.

Parameters

- **cb** – A valid application's callback structure pointer.

Returns

0 if successful, negative errno code on failure.

int **net_dhcpv4_remove_option_vendor_callback**(struct net_dhcpv4_option_callback *cb)

Remove an application callback for encapsulated vendor-specific options.

Parameters

- **cb** – A valid application's callback structure pointer.

Returns

0 if successful, negative errno code on failure.

void **net_dhcpv4_start**(struct *net_if* *iface)

Start DHCPv4 client on an iface.

Start DHCPv4 client on a given interface. DHCPv4 client will start negotiation for IPv4 address. Once the negotiation is success IPv4 address details will be added to interface.

Parameters

- **iface** – A valid pointer on an interface

void **net_dhcpv4_stop**(struct *net_if* *iface)

Stop DHCPv4 client on an iface.

Stop DHCPv4 client on a given interface. DHCPv4 client will remove all configuration obtained from a DHCP server from the interface and stop any further negotiation with the server.

Parameters

- **iface** – A valid pointer on an interface

void **net_dhcpv4_restart**(struct *net_if* *iface)

Restart DHCPv4 client on an iface.

Restart DHCPv4 client on a given interface. DHCPv4 client will restart the state machine without any of the initial delays used in start.

Parameters

- `iface` – A valid pointer on an interface

const char *`net_dhcpv4_msg_type_name`(enum *net_dhcpv4_msg_type* msg_type)

Return a text representation of the msg_type.

Parameters

- `msg_type` – The msg_type to be converted to text

Returns

A text representation of msg_type

DHCPv6

- *Overview*
- *API Reference*

Overview The Dynamic Host Configuration Protocol (DHCP) for IPv6 is a network management protocol used on IPv6 based networks. A DHCPv6 server dynamically assigns an IPv6 address and other network configuration parameters to each device on a network so they can communicate with other IP networks. See this DHCPv6 Wikipedia article for a detailed overview of how DHCPv6 works.

Note that Zephyr only supports DHCPv6 client functionality.

API Reference

group `dhcpv6`

DHCPv6.

Functions

void `net_dhcpv6_start`(struct *net_if* *iface, struct *net_dhcpv6_params* *params)

Start DHCPv6 client on an iface.

Start DHCPv6 client on a given interface. DHCPv6 client will start negotiation for IPv6 address and/or prefix, depending on the configuration. Once the negotiation is complete, IPv6 address/prefix details will be added to the interface.

Parameters

- `iface` – A valid pointer to a network interface
- `params` – DHCPv6 client configuration parameters.

void `net_dhcpv6_stop`(struct *net_if* *iface)

Stop DHCPv6 client on an iface.

Stop DHCPv6 client on a given interface. DHCPv6 client will remove all configuration obtained from a DHCP server from the interface and stop any further negotiation with the server.

Parameters

- `iface` – A valid pointer to a network interface

void **net_dhcpv6_restart**(struct *net_if* *iface)

Restart DHCPv6 client on an iface.

Restart DHCPv6 client on a given interface. DHCPv6 client will restart the state machine without any of the initial delays.

Parameters

- **iface** – A valid pointer to a network interface

struct **net_dhcpv6_params**

#include <*dhcpv6.h*> DHCPv6 client configuration parameters.

Public Members

bool **request_addr**

Request IPv6 address.

bool **request_prefix**

Request IPv6 prefix.

Hostname Configuration

- *Overview*
- *API Reference*

Overview A networked device might need a hostname, for example, if the device is configured to be a mDNS responder (see *DNS Resolve* for details) and needs to respond to `<hostname>.local` DNS queries.

The `CONFIG_NET_HOSTNAME_ENABLE` must be set in order to store the hostname and enable the relevant APIs. If the option is enabled, then the default hostname is set to be `zephyr` by `CONFIG_NET_HOSTNAME` option.

If the same firmware image is used to flash multiple boards, then it is not practical to use the same hostname in all of the boards. In that case, one can enable `CONFIG_NET_HOSTNAME_UNIQUE` which will add a unique postfix to the hostname. By default the link local address of the first network interface is used as a postfix. In Ethernet networks, the link local address refers to MAC address. For example, if the link local address is `01:02:03:04:05:06`, then the unique hostname could be `zephyr010203040506`. If you want to set the prefix yourself, then call `net_hostname_set_postfix()` before the network interfaces are created. For example for the Ethernet networks, the initialization priority is set by `CONFIG_ETH_INIT_PRIORITY` so you would need to set the postfix before that. The postfix can be set only once.

API Reference

group **net_hostname**

Network hostname configuration library.

Defines

NET_HOSTNAME_MAX_LEN

Maximum hostname length.

Functions

static inline const char *net_hostname_get(void)

Get the device hostname.

Return pointer to device hostname.

Returns
Pointer to hostname or NULL if not set.

static inline int net_hostname_set(char *host, size_t len)

Set the device hostname.

Parameters

- **host** – new hostname as char array.
- **len** – Length of the hostname array.

Returns
0 if ok, <0 on error

static inline void net_hostname_init(void)

Initialize and set the device hostname.

static inline int net_hostname_set_postfix(const uint8_t *hostname_postfix, int postfix_len)

Set the device hostname postfix.

Set the device hostname to some value. This is only used if CONFIG_NET_HOSTNAME_UNIQUE is set.

Parameters

- **hostname_postfix** – Usually link address. The function will convert this to a string.
- **postfix_len** – Length of the hostname_postfix array.

Returns
0 if ok, <0 if error

Network Core Helpers

- *Overview*
- *API Reference*

Overview The network subsystem contains two functions for sending and receiving data from the network. The `net_recv_data()` is typically used by network device driver when the received network data needs to be pushed up in the network stack for further processing. All the data is received via a network interface which is typically created by the device driver.

For sending, the `net_send_data()` can be used. Typically applications do not call this function directly as there is the *BSD Sockets* API for sending and receiving network data.

Related code samples

Telnet console
Access Zephyr shell over telnet.

mDNS responder
Listen and respond to mDNS queries.

API Reference

group `net_core`

Network core library.

Enums

enum `net_verdict`

Net Verdict.

Values:

enumerator `NET_OK`

Packet has been taken care of.

enumerator `NET_CONTINUE`

Packet has not been touched, other part should decide about its fate.

enumerator `NET_DROP`

Packet must be dropped.

Functions

int `net_recv_data`(struct *net_if* *iface, struct *net_pkt* *pkt)

Called by lower network stack or network device driver when a network packet has been received.

The function will push the packet up in the network stack for further processing.

Parameters

- `iface` – Network interface where the packet was received.
- `pkt` – Network packet data.

Returns
0 if ok, <0 if error.

int `net_send_data`(struct *net_pkt* *pkt)

Send data to network.

Send data to network. This should not be used normally by applications as it requires that the network packet is properly constructed.

Parameters

- `pkt` – Network packet.

Returns

0 if ok, <0 if error. If <0 is returned, then the caller needs to unref the pkt in order to avoid memory leak.

Network Interface

- *Overview*
- *Network interface state management*
- *API Reference*

Overview The network interface is a nexus that ties the network device drivers and the upper part of the network stack together. All the sent and received data is transferred via a network interface. The network interfaces cannot be created at runtime. A special linker section will contain information about them and that section is populated at linking time.

Network interfaces are created by `NET_DEVICE_INIT()` macro. For Ethernet network, a macro called `ETH_NET_DEVICE_INIT()` should be used instead as it will create VLAN interfaces automatically if `CONFIG_NET_VLAN` is enabled. These macros are typically used in network device driver source code.

The network interface can be turned ON by calling `net_if_up()` and OFF by calling `net_if_down()`. When the device is powered ON, the network interface is also turned ON by default.

The network interfaces can be referenced either by a `struct net_if *` pointer or by a network interface index. The network interface can be resolved from its index by calling `net_if_get_by_index()` and from interface pointer by calling `net_if_get_by_iface()`.

The IP address for network devices must be set for them to be connectable. In a typical dynamic network environment, IP addresses are set automatically by DHCPv4, for example. If needed though, the application can set a device's IP address manually. See the API documentation below for functions such as `net_if_ipv4_addr_add()` that do that.

The `net_if_get_default()` returns a *default* network interface. What this default interface means can be configured via options like `CONFIG_NET_DEFAULT_IF_FIRST` and `CONFIG_NET_DEFAULT_IF_ETHERNET`. See Kconfig file subsys/net/ip/Kconfig what options are available for selecting the default network interface.

The transmitted and received network packets can be classified via a network packet priority. This is typically done in Ethernet networks when virtual LANs (VLANs) are used. Higher priority packets can be sent or received earlier than lower priority packets. The traffic class setup can be configured by `CONFIG_NET_TC_TX_COUNT` and `CONFIG_NET_TC_RX_COUNT` options.

If the `CONFIG_NET_PROMISCUOUS_MODE` is enabled and if the underlying network technology supports promiscuous mode, then it is possible to receive all the network packets that the network device driver is able to receive. See *Promiscuous Mode* API for more details.

Network interface state management Zephyr distinguishes between two interface states: administrative state and operational state, as described in RFC 2863. The administrative state indicate whether an interface is turned ON or OFF. This state is represented by *`NET_IF_UP`* flag and is

controlled by the application. It can be changed by calling *net_if_up()* or *net_if_down()* functions. Network drivers or L2 implementations should not change administrative state on their own.

Bringing an interface up however not always means that the interface is ready to transmit packets. Because of that, operational state, which represents the internal interface status, was implemented. The operational state is updated whenever one of the following conditions take place:

- The interface is brought up/down by the application (administrative state changes).
- The interface is notified by the driver/L2 that PHY status has changed.
- The interface is notified by the driver/L2 that it joined/left a network.

The PHY status is represented with *NET_IF_LOWER_UP* flag and can be changed with *net_if_carrier_on()* and *net_if_carrier_off()*. By default, the flag is set on a newly initialized interface. An example of an event that changes the carrier state is Ethernet cable being plugged in or out.

The network association status is represented with *NET_IF_DORMANT* flag and can be changed with *net_if_dormant_on()* and *net_if_dormant_off()*. By default, the flag is cleared on a newly initialized interface. An example of an event that changes the dormant state is a Wi-Fi driver successfully connecting to an access point. In this scenario, driver should set the dormant state to ON during initialization, and once it detects that it connected to a Wi-Fi network, the dormant state should be set to OFF.

The operational state of an interface is updated as follows:

- `!net_if_is_admin_up()`

  Interface is in *NET_IF_OPER_DOWN*.

- `net_if_is_admin_up() && !net_if_is_carrier_ok()`

  Interface is in *NET_IF_OPER_DOWN* or *NET_IF_OPER_LOWERLAYERDOWN* if the interface is stacked (virtual).

- `net_if_is_admin_up() && net_if_is_carrier_ok() && net_if_is_dormant()`

  Interface is in *NET_IF_OPER_DORMANT*.

- `net_if_is_admin_up() && net_if_is_carrier_ok() && !net_if_is_dormant()`

  Interface is in *NET_IF_OPER_UP*.

Only after an interface enters *NET_IF_OPER_UP* state the *NET_IF_RUNNING* flag is set on the interface indicating that the interface is ready to be used by the application.

Related code samples

IPv4 autoconf client
: Perform IPv4 autoconfiguration and self-assign a random IPv4 address

Network management socket
: Listen to network management events using a network management socket.

Telnet console
: Access Zephyr shell over telnet.

Virtual LAN
: Setup two virtual LAN networks and use net-shell to view the networks' settings.

API Reference

group `net_if`

Network Interface abstraction layer.

Defines

`NET_DEVICE_INIT`(dev_id, name, init_fn, pm, data, config, prio, api, l2, l2_ctx_type, mtu)

Create a network interface and bind it to network device.

Parameters

- `dev_id` – Network device id.
- `name` – The name this instance of the driver exposes to the system.
- `init_fn` – Address to the init function of the driver.
- `pm` – Reference to struct *pm_device* associated with the device. (optional).
- `data` – Pointer to the device's private data.
- `config` – The address to the structure containing the configuration information for this instance of the driver.
- `prio` – The initialization level at which configuration occurs.
- `api` – Provides an initial pointer to the API function struct used by the driver. Can be NULL.
- `l2` – Network L2 layer for this network interface.
- `l2_ctx_type` – Type of L2 context data.
- `mtu` – Maximum transfer unit in bytes for this network interface.

`NET_DEVICE_DT_DEFINE`(node_id, init_fn, pm, data, config, prio, api, l2, l2_ctx_type, mtu)

Like NET_DEVICE_INIT but taking metadata from a devicetree node.

Create a network interface and bind it to network device.

Parameters

- `node_id` – The devicetree node identifier.
- `init_fn` – Address to the init function of the driver.
- `pm` – Reference to struct *pm_device* associated with the device. (optional).
- `data` – Pointer to the device's private data.
- `config` – The address to the structure containing the configuration information for this instance of the driver.
- `prio` – The initialization level at which configuration occurs.
- `api` – Provides an initial pointer to the API function struct used by the driver. Can be NULL.
- `l2` – Network L2 layer for this network interface.
- `l2_ctx_type` – Type of L2 context data.
- `mtu` – Maximum transfer unit in bytes for this network interface.

`NET_DEVICE_DT_INST_DEFINE`(inst, ...)

Like NET_DEVICE_DT_DEFINE for an instance of a DT_DRV_COMPAT compatible.

Parameters

- `inst` – instance number. This is replaced by `DT_DRV_COMPAT(inst)` in the call to NET_DEVICE_DT_DEFINE.
- `...` – other parameters as expected by NET_DEVICE_DT_DEFINE.

`NET_DEVICE_INIT_INSTANCE`(dev_id, name, instance, init_fn, pm, data, config, prio, api, l2, l2_ctx_type, mtu)

Create multiple network interfaces and bind them to network device.

If your network device needs more than one instance of a network interface, use this macro below and provide a different instance suffix each time (0, 1, 2, ... or a, b, c ... whatever works for you)

Parameters

- `dev_id` – Network device id.
- `name` – The name this instance of the driver exposes to the system.
- `instance` – Instance identifier.
- `init_fn` – Address to the init function of the driver.
- `pm` – Reference to struct *pm_device* associated with the device. (optional).
- `data` – Pointer to the device's private data.
- `config` – The address to the structure containing the configuration information for this instance of the driver.
- `prio` – The initialization level at which configuration occurs.
- `api` – Provides an initial pointer to the API function struct used by the driver. Can be NULL.
- `l2` – Network L2 layer for this network interface.
- `l2_ctx_type` – Type of L2 context data.
- `mtu` – Maximum transfer unit in bytes for this network interface.

`NET_DEVICE_DT_DEFINE_INSTANCE`(node_id, instance, init_fn, pm, data, config, prio, api, l2, l2_ctx_type, mtu)

Like NET_DEVICE_OFFLOAD_INIT but taking metadata from a devicetree.

Create multiple network interfaces and bind them to network device. If your network device needs more than one instance of a network interface, use this macro below and provide a different instance suffix each time (0, 1, 2, ... or a, b, c ... whatever works for you)

Parameters

- `node_id` – The devicetree node identifier.
- `instance` – Instance identifier.
- `init_fn` – Address to the init function of the driver.
- `pm` – Reference to struct *pm_device* associated with the device. (optional).
- `data` – Pointer to the device's private data.
- `config` – The address to the structure containing the configuration information for this instance of the driver.
- `prio` – The initialization level at which configuration occurs.
- `api` – Provides an initial pointer to the API function struct used by the driver. Can be NULL.
- `l2` – Network L2 layer for this network interface.

- **l2_ctx_type** – Type of L2 context data.
- **mtu** – Maximum transfer unit in bytes for this network interface.

NET_DEVICE_DT_INST_DEFINE_INSTANCE(inst, ...)

Like NET_DEVICE_DT_DEFINE_INSTANCE for an instance of a DT_DRV_COMPAT compatible.

Parameters

- **inst** – instance number. This is replaced by `DT_DRV_COMPAT(inst)` in the call to NET_DEVICE_DT_DEFINE_INSTANCE.
- **...** – other parameters as expected by NET_DEVICE_DT_DEFINE_INSTANCE.

NET_DEVICE_OFFLOAD_INIT(dev_id, name, init_fn, pm, data, config, prio, api, mtu)

Create a offloaded network interface and bind it to network device.

The offloaded network interface is implemented by a device vendor HAL or similar.

Parameters

- **dev_id** – Network device id.
- **name** – The name this instance of the driver exposes to the system.
- **init_fn** – Address to the init function of the driver.
- **pm** – Reference to struct *pm_device* associated with the device. (optional).
- **data** – Pointer to the device's private data.
- **config** – The address to the structure containing the configuration information for this instance of the driver.
- **prio** – The initialization level at which configuration occurs.
- **api** – Provides an initial pointer to the API function struct used by the driver. Can be NULL.
- **mtu** – Maximum transfer unit in bytes for this network interface.

NET_DEVICE_DT_OFFLOAD_DEFINE(node_id, init_fn, pm, data, config, prio, api, mtu)

Like NET_DEVICE_OFFLOAD_INIT but taking metadata from a devicetree node.

Create a offloaded network interface and bind it to network device. The offloaded network interface is implemented by a device vendor HAL or similar.

Parameters

- **node_id** – The devicetree node identifier.
- **init_fn** – Address to the init function of the driver.
- **pm** – Reference to struct *pm_device* associated with the device. (optional).
- **data** – Pointer to the device's private data.
- **config** – The address to the structure containing the configuration information for this instance of the driver.
- **prio** – The initialization level at which configuration occurs.
- **api** – Provides an initial pointer to the API function struct used by the driver. Can be NULL.
- **mtu** – Maximum transfer unit in bytes for this network interface.

NET_DEVICE_DT_INST_OFFLOAD_DEFINE(inst, ...)

Like NET_DEVICE_DT_OFFLOAD_DEFINE for an instance of a DT_DRV_COMPAT compatible.

Parameters

- **inst** – instance number. This is replaced by DT_DRV_COMPAT(inst) in the call to NET_DEVICE_DT_OFFLOAD_DEFINE.
- **...** – other parameters as expected by NET_DEVICE_DT_OFFLOAD_DEFINE.

NET_IFACE_COUNT(_dst)

Count the number of network interfaces.

Parameters

- **_dst** – **[out]** Pointer to location where result is written.

Typedefs

typedef int (***net_socket_create_t**)(int, int, int)

A function prototype to create an offloaded socket.

The prototype is compatible with *socket()* function.

typedef void (***net_if_ip_addr_cb_t**)(struct *net_if* *iface, struct *net_if_addr* *addr, void *user_data)

Callback used while iterating over network interface IP addresses.

Param iface
Pointer to the network interface the address belongs to

Param addr
Pointer to current IP address

Param user_data
A valid pointer to user data or NULL

typedef void (***net_if_ip_maddr_cb_t**)(struct *net_if* *iface, struct *net_if_mcast_addr* *maddr, void *user_data)

Callback used while iterating over network interface multicast IP addresses.

Param iface
Pointer to the network interface the address belongs to

Param maddr
Pointer to current multicast IP address

Param user_data
A valid pointer to user data or NULL

typedef void (***net_if_mcast_callback_t**)(struct *net_if* *iface, const struct net_addr *addr, bool is_joined)

Define a callback that is called whenever a IPv6 or IPv4 multicast address group is joined or left.

Param iface
A pointer to a struct *net_if* to which the multicast address is attached.

Param addr
IP multicast address.

Param is_joined
True if the multicast group is joined, false if group is left.

typedef void (*`net_if_link_callback_t`)(struct *net_if* *iface, struct *net_linkaddr* *dst, int status)

Define callback that is called after a network packet has been sent.

Param iface
A pointer to a struct *net_if* to which the *net_pkt* was sent to.

Param dst
Link layer address of the destination where the network packet was sent.

Param status
Send status, 0 is ok, < 0 error.

typedef void (*`net_if_cb_t`)(struct *net_if* *iface, void *user_data)

Callback used while iterating over network interfaces.

Param iface
Pointer to current network interface

Param user_data
A valid pointer to user data or NULL

Enums

enum `net_if_flag`

Network interface flags.

Values:

enumerator `NET_IF_UP`

Interface is admin up.

enumerator `NET_IF_POINTOPOINT`

Interface is pointopoint.

enumerator `NET_IF_PROMISC`

Interface is in promiscuous mode.

enumerator `NET_IF_NO_AUTO_START`

Do not start the interface immediately after initialization.

This requires that either the device driver or some other entity will need to manually take the interface up when needed. For example for Ethernet this will happen when the driver calls the *net_eth_carrier_on()* function.

enumerator `NET_IF_SUSPENDED`

Power management specific: interface is being suspended.

enumerator `NET_IF_FORWARD_MULTICASTS`

Flag defines if received multicasts of other interface are forwarded on this interface.

This activates multicast routing / forwarding for this interface.

enumerator NET_IF_IPV4
Interface supports IPv4.

enumerator NET_IF_IPV6
Interface supports IPv6.

enumerator NET_IF_RUNNING
Interface up and running (ready to receive and transmit).

enumerator NET_IF_LOWER_UP
Driver signals L1 is up.

enumerator NET_IF_DORMANT
Driver signals dormant.

enumerator NET_IF_IPV6_NO_ND
IPv6 Neighbor Discovery disabled.

enumerator NET_IF_IPV6_NO_MLD
IPv6 Multicast Listener Discovery disabled.

enumerator NET_IF_NO_TX_LOCK
Mutex locking on TX data path disabled on the interface.

enum net_if_oper_state
Network interface operational status (RFC 2863).
Values:

enumerator NET_IF_OPER_UNKNOWN
Initial (unknown) value.

enumerator NET_IF_OPER_NOTPRESENT
Hardware missing.

enumerator NET_IF_OPER_DOWN
Interface is down.

enumerator NET_IF_OPER_LOWERLAYERDOWN
Lower layer interface is down.

enumerator NET_IF_OPER_TESTING
Training mode.

enumerator NET_IF_OPER_DORMANT
Waiting external action.

enumerator NET_IF_OPER_UP
Interface is up.

enum **net_if_checksum_type**

Type of checksum for which support in the interface will be queried.

Values:

enumerator **NET_IF_CHECKSUM_IPV4_HEADER** = NET_IF_CHECKSUM_IPV4_HEADER_BIT

Interface supports IP version 4 header checksum calculation.

enumerator **NET_IF_CHECKSUM_IPV4_TCP** = NET_IF_CHECKSUM_IPV4_HEADER_BIT | NET_IF_CHECKSUM_TCP_BIT

Interface supports checksum calculation for TCP payload in IPv4.

enumerator **NET_IF_CHECKSUM_IPV4_UDP** = NET_IF_CHECKSUM_IPV4_HEADER_BIT | NET_IF_CHECKSUM_UDP_BIT

Interface supports checksum calculation for UDP payload in IPv4.

enumerator **NET_IF_CHECKSUM_IPV4_ICMP** = NET_IF_CHECKSUM_IPV4_ICMP_BIT

Interface supports checksum calculation for ICMP4 payload in IPv4.

enumerator **NET_IF_CHECKSUM_IPV6_HEADER** = NET_IF_CHECKSUM_IPV6_HEADER_BIT

Interface supports IP version 6 header checksum calculation.

enumerator **NET_IF_CHECKSUM_IPV6_TCP** = NET_IF_CHECKSUM_IPV6_HEADER_BIT | NET_IF_CHECKSUM_TCP_BIT

Interface supports checksum calculation for TCP payload in IPv6.

enumerator **NET_IF_CHECKSUM_IPV6_UDP** = NET_IF_CHECKSUM_IPV6_HEADER_BIT | NET_IF_CHECKSUM_UDP_BIT

Interface supports checksum calculation for UDP payload in IPv6.

enumerator **NET_IF_CHECKSUM_IPV6_ICMP** = NET_IF_CHECKSUM_IPV6_ICMP_BIT

Interface supports checksum calculation for ICMP6 payload in IPv6.

Functions

static inline void **net_if_flag_set**(struct *net_if* *iface, enum *net_if_flag* value)

Set a value in network interface flags.

Parameters

- **iface** – Pointer to network interface
- **value** – Flag value

static inline bool **net_if_flag_test_and_set**(struct *net_if* *iface, enum *net_if_flag* value)

Test and set a value in network interface flags.

Parameters

- **iface** – Pointer to network interface
- **value** – Flag value

Returns

true if the bit was set, false if it wasn't.

static inline void **net_if_flag_clear**(struct *net_if* *iface, enum *net_if_flag* value)

Clear a value in network interface flags.

Parameters

- **iface** – Pointer to network interface
- **value** – Flag value

static inline bool **net_if_flag_test_and_clear**(struct *net_if* *iface, enum *net_if_flag* value)

Test and clear a value in network interface flags.

Parameters

- **iface** – Pointer to network interface
- **value** – Flag value

Returns

true if the bit was set, false if it wasn't.

static inline bool **net_if_flag_is_set**(struct *net_if* *iface, enum *net_if_flag* value)

Check if a value in network interface flags is set.

Parameters

- **iface** – Pointer to network interface
- **value** – Flag value

Returns

True if the value is set, false otherwise

static inline enum *net_if_oper_state* **net_if_oper_state_set**(struct *net_if* *iface, enum *net_if_oper_state* oper_state)

Set an operational state on an interface.

Parameters

- **iface** – Pointer to network interface
- **oper_state** – Operational state to set

Returns

The new operational state of an interface

static inline enum *net_if_oper_state* **net_if_oper_state**(struct *net_if* *iface)

Get an operational state of an interface.

Parameters

- **iface** – Pointer to network interface

Returns

Operational state of an interface

enum *net_verdict* **net_if_send_data**(struct *net_if* *iface, struct *net_pkt* *pkt)

Send a packet through a net iface.

return verdict about the packet

Parameters

- **iface** – Pointer to a network interface structure
- **pkt** – Pointer to a net packet to send

static inline const struct *net_l2* ***net_if_l2**(struct *net_if* *iface)

Get a pointer to the interface L2.

Parameters

- **iface** – a valid pointer to a network interface structure

Returns

a pointer to the iface L2

enum *net_verdict* **net_if_recv_data**(struct *net_if* *iface, struct *net_pkt* *pkt)

Input a packet through a net iface.

Parameters

- **iface** – Pointer to a network interface structure
- **pkt** – Pointer to a net packet to input

Returns

verdict about the packet

static inline void ***net_if_l2_data**(struct *net_if* *iface)

Get a pointer to the interface L2 private data.

Parameters

- **iface** – a valid pointer to a network interface structure

Returns

a pointer to the iface L2 data

static inline const struct *device* ***net_if_get_device**(struct *net_if* *iface)

Get an network interface's device.

Parameters

- **iface** – Pointer to a network interface structure

Returns

a pointer to the device driver instance

void **net_if_queue_tx**(struct *net_if* *iface, struct *net_pkt* *pkt)

Queue a packet to the net interface TX queue.

Parameters

- **iface** – Pointer to a network interface structure
- **pkt** – Pointer to a net packet to queue

static inline bool **net_if_is_ip_offloaded**(struct *net_if* *iface)

Return the IP offload status.

Parameters

- **iface** – Network interface

Returns

True if IP offloading is active, false otherwise.

bool **net_if_is_offloaded**(struct *net_if* *iface)

Return offload status of a given network interface.

Parameters

- **iface** – Network interface

Returns

True if IP or socket offloading is active, false otherwise.

static inline struct net_offload *net_if_offload(struct *net_if* *iface)

Return the IP offload plugin.

Parameters

- iface – Network interface

Returns

NULL if there is no offload plugin defined, valid pointer otherwise

static inline bool net_if_is_socket_offloaded(struct *net_if* *iface)

Return the socket offload status.

Parameters

- iface – Network interface

Returns

True if socket offloading is active, false otherwise.

static inline void net_if_socket_offload_set(struct *net_if* *iface, *net_socket_create_t* socket_offload)

Set the function to create an offloaded socket.

Parameters

- iface – Network interface
- socket_offload – A function to create an offloaded socket

static inline *net_socket_create_t* net_if_socket_offload(struct *net_if* *iface)

Return the function to create an offloaded socket.

Parameters

- iface – Network interface

Returns

NULL if the interface is not socket offloaded, valid pointer otherwise

static inline struct *net_linkaddr* *net_if_get_link_addr(struct *net_if* *iface)

Get an network interface's link address.

Parameters

- iface – Pointer to a network interface structure

Returns

a pointer to the network link address

static inline struct *net_if_config* *net_if_get_config(struct *net_if* *iface)

Return network configuration for this network interface.

Parameters

- iface – Pointer to a network interface structure

Returns

Pointer to configuration

static inline void net_if_start_dad(struct *net_if* *iface)

Start duplicate address detection procedure.

Parameters

- iface – Pointer to a network interface structure

void **net_if_start_rs**(struct *net_if* *iface)

Start neighbor discovery and send router solicitation message.

Parameters

- **iface** – Pointer to a network interface structure

static inline void **net_if_stop_rs**(struct *net_if* *iface)

Stop neighbor discovery.

Parameters

- **iface** – Pointer to a network interface structure

static inline void **net_if_nbr_reachability_hint**(struct *net_if* *iface, const struct *in6_addr* *ipv6_addr)

Provide a reachability hint for IPv6 Neighbor Discovery.

This function is intended for upper-layer protocols to inform the IPv6 Neighbor Discovery process about an active link to a specific neighbor. By signaling a recent "forward progress" event, such as the reception of an ACK, this function can help reduce unnecessary ND traffic as per the guidelines in RFC 4861 (section 7.3).

Parameters

- **iface** – A pointer to the network interface.
- **ipv6_addr** – Pointer to the IPv6 address of the neighbor node.

static inline int **net_if_set_link_addr**(struct *net_if* *iface, uint8_t *addr, uint8_t len, enum *net_link_type* type)

Set a network interface's link address.

Parameters

- **iface** – Pointer to a network interface structure
- **addr** – A pointer to a uint8_t buffer representing the address. The buffer must remain valid throughout interface lifetime.
- **len** – length of the address buffer
- **type** – network bearer type of this link address

Returns

0 on success

static inline uint16_t **net_if_get_mtu**(struct *net_if* *iface)

Get an network interface's MTU.

Parameters

- **iface** – Pointer to a network interface structure

Returns

the MTU

static inline void **net_if_set_mtu**(struct *net_if* *iface, uint16_t mtu)

Set an network interface's MTU.

Parameters

- **iface** – Pointer to a network interface structure
- **mtu** – New MTU, note that we store only 16 bit mtu value.

static inline void **net_if_addr_set_lf**(struct *net_if_addr* *ifaddr, bool is_infinite)

Set the infinite status of the network interface address.

Parameters

- **ifaddr** – IP address for network interface
- **is_infinite** – Infinite status

struct *net_if* ***net_if_get_by_link_addr**(struct *net_linkaddr* *ll_addr)

Get an interface according to link layer address.

Parameters

- **ll_addr** – Link layer address.

Returns

Network interface or NULL if not found.

struct *net_if* ***net_if_lookup_by_dev**(const struct *device* *dev)

Find an interface from it's related device.

Parameters

- **dev** – A valid struct device pointer to relate with an interface

Returns

a valid struct *net_if* pointer on success, NULL otherwise

static inline struct *net_if_config* ***net_if_config_get**(struct *net_if* *iface)

Get network interface IP config.

Parameters

- **iface** – Interface to use.

Returns

NULL if not found or pointer to correct config settings.

void **net_if_router_rm**(struct *net_if_router* *router)

Remove a router from the system.

Parameters

- **router** – Pointer to existing router

void **net_if_set_default**(struct *net_if* *iface)

Set the default network interface.

Parameters

- **iface** – New default interface, or NULL to revert to the one set by Kconfig.

struct *net_if* ***net_if_get_default**(void)

Get the default network interface.

Returns

Default interface or NULL if no interfaces are configured.

struct *net_if* ***net_if_get_first_by_type**(const struct *net_l2* *l2)

Get the first network interface according to its type.

Parameters

- **l2** – Layer 2 type of the network interface.

Returns

First network interface of a given type or NULL if no such interfaces was found.

struct *net_if* *net_if_get_first_up(void)

Get the first network interface which is up.

Returns
First network interface which is up or NULL if all interfaces are down.

int net_if_config_ipv6_get(struct *net_if* *iface, struct *net_if_ipv6* **ipv6)

Allocate network interface IPv6 config.

This function will allocate new IPv6 config.

Parameters

- iface – Interface to use.
- ipv6 – Pointer to allocated IPv6 struct is returned to caller.

Returns
0 if ok, <0 if error

int net_if_config_ipv6_put(struct *net_if* *iface)

Release network interface IPv6 config.

Parameters

- iface – Interface to use.

Returns
0 if ok, <0 if error

struct *net_if_addr* *net_if_ipv6_addr_lookup(const struct *in6_addr* *addr, struct *net_if* **iface)

Check if this IPv6 address belongs to one of the interfaces.

Parameters

- addr – IPv6 address
- iface – Pointer to interface is returned

Returns
Pointer to interface address, NULL if not found.

struct *net_if_addr* *net_if_ipv6_addr_lookup_by_iface(struct *net_if* *iface, struct *in6_addr* *addr)

Check if this IPv6 address belongs to this specific interfaces.

Parameters

- iface – Network interface
- addr – IPv6 address

Returns
Pointer to interface address, NULL if not found.

int net_if_ipv6_addr_lookup_by_index(const struct *in6_addr* *addr)

Check if this IPv6 address belongs to one of the interface indices.

Parameters

- addr – IPv6 address

Returns
>0 if address was found in given network interface index, all other values mean address was not found

struct *net_if_addr* *net_if_ipv6_addr_add(struct *net_if* *iface, struct *in6_addr* *addr, enum *net_addr_type* addr_type, uint32_t vlifetime)

Add a IPv6 address to an interface.

Parameters

- `iface` – Network interface
- `addr` – IPv6 address
- `addr_type` – IPv6 address type
- `vlifetime` – Validity time for this address

Returns

Pointer to interface address, NULL if cannot be added

bool net_if_ipv6_addr_add_by_index(int index, struct *in6_addr* *addr, enum *net_addr_type* addr_type, uint32_t vlifetime)

Add a IPv6 address to an interface by index.

Parameters

- `index` – Network interface index
- `addr` – IPv6 address
- `addr_type` – IPv6 address type
- `vlifetime` – Validity time for this address

Returns

True if ok, false if address could not be added

void net_if_ipv6_addr_update_lifetime(struct *net_if_addr* *ifaddr, uint32_t vlifetime)

Update validity lifetime time of an IPv6 address.

Parameters

- `ifaddr` – Network IPv6 address
- `vlifetime` – Validity time for this address

bool net_if_ipv6_addr_rm(struct *net_if* *iface, const struct *in6_addr* *addr)

Remove an IPv6 address from an interface.

Parameters

- `iface` – Network interface
- `addr` – IPv6 address

Returns

True if successfully removed, false otherwise

bool net_if_ipv6_addr_rm_by_index(int index, const struct *in6_addr* *addr)

Remove an IPv6 address from an interface by index.

Parameters

- `index` – Network interface index
- `addr` – IPv6 address

Returns

True if successfully removed, false otherwise

void **net_if_ipv6_addr_foreach**(struct *net_if* *iface, *net_if_ip_addr_cb_t* cb, void *user_data)

Go through all IPv6 addresses on a network interface and call callback for each used address.

Parameters

- **iface** – Pointer to the network interface
- **cb** – User-supplied callback function to call
- **user_data** – User specified data

struct *net_if_mcast_addr* ***net_if_ipv6_maddr_add**(struct *net_if* *iface, const struct *in6_addr* *addr)

Add a IPv6 multicast address to an interface.

Parameters

- **iface** – Network interface
- **addr** – IPv6 multicast address

Returns

Pointer to interface multicast address, NULL if cannot be added

bool **net_if_ipv6_maddr_rm**(struct *net_if* *iface, const struct *in6_addr* *addr)

Remove an IPv6 multicast address from an interface.

Parameters

- **iface** – Network interface
- **addr** – IPv6 multicast address

Returns

True if successfully removed, false otherwise

void **net_if_ipv6_maddr_foreach**(struct *net_if* *iface, *net_if_ip_maddr_cb_t* cb, void *user_data)

Go through all IPv6 multicast addresses on a network interface and call callback for each used address.

Parameters

- **iface** – Pointer to the network interface
- **cb** – User-supplied callback function to call
- **user_data** – User specified data

struct *net_if_mcast_addr* ***net_if_ipv6_maddr_lookup**(const struct *in6_addr* *addr, struct *net_if* **iface)

Check if this IPv6 multicast address belongs to a specific interface or one of the interfaces.

Parameters

- **addr** – IPv6 address
- **iface** – If *iface is null, then pointer to interface is returned, otherwise the *iface value needs to be matched.

Returns

Pointer to interface multicast address, NULL if not found.

void **net_if_mcast_mon_register**(struct *net_if_mcast_monitor* *mon, struct *net_if* *iface, *net_if_mcast_callback_t* cb)

Register a multicast monitor.

Parameters

- **mon** – Monitor handle. This is a pointer to a monitor storage structure which should be allocated by caller, but does not need to be initialized.
- **iface** – Network interface or NULL for all interfaces
- **cb** – Monitor callback

void **net_if_mcast_mon_unregister**(struct *net_if_mcast_monitor* *mon)

Unregister a multicast monitor.

Parameters

- **mon** – Monitor handle

void **net_if_mcast_monitor**(struct *net_if* *iface, const struct net_addr *addr, bool is_joined)

Call registered multicast monitors.

Parameters

- **iface** – Network interface
- **addr** – Multicast address
- **is_joined** – Is this multicast address group joined (true) or not (false)

void **net_if_ipv6_maddr_join**(struct *net_if* *iface, struct *net_if_mcast_addr* *addr)

Mark a given multicast address to be joined.

Parameters

- **iface** – Network interface the address belongs to
- **addr** – IPv6 multicast address

static inline bool **net_if_ipv6_maddr_is_joined**(struct *net_if_mcast_addr* *addr)

Check if given multicast address is joined or not.

Parameters

- **addr** – IPv6 multicast address

Returns

True if address is joined, False otherwise.

void **net_if_ipv6_maddr_leave**(struct *net_if* *iface, struct *net_if_mcast_addr* *addr)

Mark a given multicast address to be left.

Parameters

- **iface** – Network interface the address belongs to
- **addr** – IPv6 multicast address

struct *net_if_ipv6_prefix* ***net_if_ipv6_prefix_get**(struct *net_if* *iface, const struct *in6_addr* *addr)

Return prefix that corresponds to this IPv6 address.

Parameters

- **iface** – Network interface
- **addr** – IPv6 address

Returns
Pointer to prefix, NULL if not found.

struct *net_if_ipv6_prefix* *`net_if_ipv6_prefix_lookup`(struct *net_if* *iface, struct *in6_addr* *addr, uint8_t len)

Check if this IPv6 prefix belongs to this interface.

Parameters

- `iface` – Network interface
- `addr` – IPv6 address
- `len` – Prefix length

Returns
Pointer to prefix, NULL if not found.

struct *net_if_ipv6_prefix* *`net_if_ipv6_prefix_add`(struct *net_if* *iface, struct *in6_addr* *prefix, uint8_t len, uint32_t lifetime)

Add a IPv6 prefix to an network interface.

Parameters

- `iface` – Network interface
- `prefix` – IPv6 address
- `len` – Prefix length
- `lifetime` – Prefix lifetime in seconds

Returns
Pointer to prefix, NULL if the prefix was not added.

bool `net_if_ipv6_prefix_rm`(struct *net_if* *iface, struct *in6_addr* *addr, uint8_t len)

Remove an IPv6 prefix from an interface.

Parameters

- `iface` – Network interface
- `addr` – IPv6 prefix address
- `len` – Prefix length

Returns
True if successfully removed, false otherwise

static inline void `net_if_ipv6_prefix_set_lf`(struct *net_if_ipv6_prefix* *prefix, bool is_infinite)

Set the infinite status of the prefix.

Parameters

- `prefix` – IPv6 address
- `is_infinite` – Infinite status

void `net_if_ipv6_prefix_set_timer`(struct *net_if_ipv6_prefix* *prefix, uint32_t lifetime)

Set the prefix lifetime timer.

Parameters

- `prefix` – IPv6 address
- `lifetime` – Prefix lifetime in seconds

void **net_if_ipv6_prefix_unset_timer**(struct *net_if_ipv6_prefix* *prefix)

Unset the prefix lifetime timer.

Parameters

- **prefix** – IPv6 address

bool **net_if_ipv6_addr_onlink**(struct *net_if* **iface, struct *in6_addr* *addr)

Check if this IPv6 address is part of the subnet of our network interface.

Parameters

- **iface** – Network interface. This is returned to the caller. The iface can be NULL in which case we check all the interfaces.
- **addr** – IPv6 address

Returns

True if address is part of our subnet, false otherwise

static inline struct *in6_addr* ***net_if_router_ipv6**(struct *net_if_router* *router)

Get the IPv6 address of the given router.

Parameters

- **router** – a network router

Returns

pointer to the IPv6 address, or NULL if none

struct *net_if_router* ***net_if_ipv6_router_lookup**(struct *net_if* *iface, struct *in6_addr* *addr)

Check if IPv6 address is one of the routers configured in the system.

Parameters

- **iface** – Network interface
- **addr** – IPv6 address

Returns

Pointer to router information, NULL if cannot be found

struct *net_if_router* ***net_if_ipv6_router_find_default**(struct *net_if* *iface, struct *in6_addr* *addr)

Find default router for this IPv6 address.

Parameters

- **iface** – Network interface. This can be NULL in which case we go through all the network interfaces to find a suitable router.
- **addr** – IPv6 address

Returns

Pointer to router information, NULL if cannot be found

void **net_if_ipv6_router_update_lifetime**(struct *net_if_router* *router, uint16_t lifetime)

Update validity lifetime time of a router.

Parameters

- **router** – Network IPv6 address
- **lifetime** – Lifetime of this router.

struct *net_if_router* *net_if_ipv6_router_add(struct *net_if* *iface, struct *in6_addr* *addr, uint16_t router_lifetime)

Add IPv6 router to the system.

Parameters

- `iface` – Network interface
- `addr` – IPv6 address
- `router_lifetime` – Lifetime of the router

Returns
Pointer to router information, NULL if could not be added

bool net_if_ipv6_router_rm(struct *net_if_router* *router)

Remove IPv6 router from the system.

Parameters

- `router` – Router information.

Returns
True if successfully removed, false otherwise

uint8_t net_if_ipv6_get_hop_limit(struct *net_if* *iface)

Get IPv6 hop limit specified for a given interface.

This is the default value but can be overridden by the user.

Parameters

- `iface` – Network interface

Returns
Hop limit

void net_if_ipv6_set_hop_limit(struct *net_if* *iface, uint8_t hop_limit)

Set the default IPv6 hop limit of a given interface.

Parameters

- `iface` – Network interface
- `hop_limit` – New hop limit

uint8_t net_if_ipv6_get_mcast_hop_limit(struct *net_if* *iface)

Get IPv6 multicast hop limit specified for a given interface.

This is the default value but can be overridden by the user.

Parameters

- `iface` – Network interface

Returns
Hop limit

void net_if_ipv6_set_mcast_hop_limit(struct *net_if* *iface, uint8_t hop_limit)

Set the default IPv6 multicast hop limit of a given interface.

Parameters

- `iface` – Network interface
- `hop_limit` – New hop limit

static inline void **net_if_ipv6_set_base_reachable_time**(struct *net_if* *iface, uint32_t reachable_time)

Set IPv6 reachable time for a given interface.

Parameters

- **iface** – Network interface
- **reachable_time** – New reachable time

static inline uint32_t **net_if_ipv6_get_reachable_time**(struct *net_if* *iface)

Get IPv6 reachable timeout specified for a given interface.

Parameters

- **iface** – Network interface

Returns

Reachable timeout

uint32_t **net_if_ipv6_calc_reachable_time**(struct *net_if_ipv6* *ipv6)

Calculate next reachable time value for IPv6 reachable time.

Parameters

- **ipv6** – IPv6 address configuration

Returns

Reachable time

static inline void **net_if_ipv6_set_reachable_time**(struct *net_if_ipv6* *ipv6)

Set IPv6 reachable time for a given interface.

This requires that base reachable time is set for the interface.

Parameters

- **ipv6** – IPv6 address configuration

static inline void **net_if_ipv6_set_retrans_timer**(struct *net_if* *iface, uint32_t retrans_timer)

Set IPv6 retransmit timer for a given interface.

Parameters

- **iface** – Network interface
- **retrans_timer** – New retransmit timer

static inline uint32_t **net_if_ipv6_get_retrans_timer**(struct *net_if* *iface)

Get IPv6 retransmit timer specified for a given interface.

Parameters

- **iface** – Network interface

Returns

Retransmit timer

static inline const struct *in6_addr* ***net_if_ipv6_select_src_addr**(struct *net_if* *iface, const struct *in6_addr* *dst)

Get a IPv6 source address that should be used when sending network data to destination.

Parameters

- **iface** – Interface that was used when packet was received. If the interface is not known, then NULL can be given.

- **dst** – IPv6 destination address

Returns
Pointer to IPv6 address to use, NULL if no IPv6 address could be found.

static inline const struct *in6_addr* ***net_if_ipv6_select_src_addr_hint**(struct *net_if* *iface, const struct *in6_addr* *dst, int flags)

Get a IPv6 source address that should be used when sending network data to destination.

Use a hint set to the socket to select the proper address.

Parameters

- **iface** – Interface that was used when packet was received. If the interface is not known, then NULL can be given.
- **dst** – IPv6 destination address
- **flags** – Hint from the related socket. See RFC 5014 for value details.

Returns
Pointer to IPv6 address to use, NULL if no IPv6 address could be found.

static inline struct *net_if* ***net_if_ipv6_select_src_iface**(const struct *in6_addr* *dst)

Get a network interface that should be used when sending IPv6 network data to destination.

Parameters

- **dst** – IPv6 destination address

Returns
Pointer to network interface to use, NULL if no suitable interface could be found.

struct *in6_addr* ***net_if_ipv6_get_ll**(struct *net_if* *iface, enum *net_addr_state* addr_state)

Get a IPv6 link local address in a given state.

Parameters

- **iface** – Interface to use. Must be a valid pointer to an interface.
- **addr_state** – IPv6 address state (preferred, tentative, deprecated)

Returns
Pointer to link local IPv6 address, NULL if no proper IPv6 address could be found.

struct *in6_addr* ***net_if_ipv6_get_ll_addr**(enum *net_addr_state* state, struct *net_if* **iface)

Return link local IPv6 address from the first interface that has a link local address matching give state.

Parameters

- **state** – IPv6 address state (ANY, TENTATIVE, PREFERRED, DEPRECATED)
- **iface** – Pointer to interface is returned

Returns
Pointer to IPv6 address, NULL if not found.

void **net_if_ipv6_dad_failed**(struct *net_if* *iface, const struct *in6_addr* *addr)

Stop IPv6 Duplicate Address Detection (DAD) procedure if we find out that our IPv6 address is already in use.

Parameters

- **iface** – Interface where the DAD was running.
- **addr** – IPv6 address that failed DAD

struct *in6_addr* ***net_if_ipv6_get_global_addr**(enum *net_addr_state* state, struct *net_if* **iface)

Return global IPv6 address from the first interface that has a global IPv6 address matching the given state.

Parameters

- **state** – IPv6 address state (ANY, TENTATIVE, PREFERRED, DEPRECATED)
- **iface** – Caller can give an interface to check. If iface is set to NULL, then all the interfaces are checked. Pointer to interface where the IPv6 address is defined is returned to the caller.

Returns

Pointer to IPv6 address, NULL if not found.

int **net_if_config_ipv4_get**(struct *net_if* *iface, struct *net_if_ipv4* **ipv4)

Allocate network interface IPv4 config.

This function will allocate new IPv4 config.

Parameters

- **iface** – Interface to use.
- **ipv4** – Pointer to allocated IPv4 struct is returned to caller.

Returns

0 if ok, <0 if error

int **net_if_config_ipv4_put**(struct *net_if* *iface)

Release network interface IPv4 config.

Parameters

- **iface** – Interface to use.

Returns

0 if ok, <0 if error

uint8_t **net_if_ipv4_get_ttl**(struct *net_if* *iface)

Get IPv4 time-to-live value specified for a given interface.

Parameters

- **iface** – Network interface

Returns

Time-to-live

void **net_if_ipv4_set_ttl**(struct *net_if* *iface, uint8_t ttl)

Set IPv4 time-to-live value specified to a given interface.

Parameters

- **iface** – Network interface
- **ttl** – Time-to-live value

uint8_t **net_if_ipv4_get_mcast_ttl**(struct *net_if* *iface)

Get IPv4 multicast time-to-live value specified for a given interface.

Parameters

- **iface** – Network interface

Returns

Time-to-live

void **net_if_ipv4_set_mcast_ttl**(struct *net_if* *iface, uint8_t ttl)

Set IPv4 multicast time-to-live value specified to a given interface.

Parameters

- **iface** – Network interface
- **ttl** – Time-to-live value

struct *net_if_addr* ***net_if_ipv4_addr_lookup**(const struct *in_addr* *addr, struct *net_if* **iface)

Check if this IPv4 address belongs to one of the interfaces.

Parameters

- **addr** – IPv4 address
- **iface** – Interface is returned

Returns

Pointer to interface address, NULL if not found.

struct *net_if_addr* ***net_if_ipv4_addr_add**(struct *net_if* *iface, struct *in_addr* *addr, enum *net_addr_type* addr_type, uint32_t vlifetime)

Add a IPv4 address to an interface.

Parameters

- **iface** – Network interface
- **addr** – IPv4 address
- **addr_type** – IPv4 address type
- **vlifetime** – Validity time for this address

Returns

Pointer to interface address, NULL if cannot be added

bool **net_if_ipv4_addr_rm**(struct *net_if* *iface, const struct *in_addr* *addr)

Remove a IPv4 address from an interface.

Parameters

- **iface** – Network interface
- **addr** – IPv4 address

Returns

True if successfully removed, false otherwise

int **net_if_ipv4_addr_lookup_by_index**(const struct *in_addr* *addr)

Check if this IPv4 address belongs to one of the interface indices.

Parameters

- **addr** – IPv4 address

Returns

>0 if address was found in given network interface index, all other values mean address was not found

bool **net_if_ipv4_addr_add_by_index**(int index, struct *in_addr* *addr, enum *net_addr_type* addr_type, uint32_t vlifetime)

Add a IPv4 address to an interface by network interface index.

Parameters

- **index** – Network interface index
- **addr** – IPv4 address
- **addr_type** – IPv4 address type
- **vlifetime** – Validity time for this address

Returns
True if ok, false if the address could not be added

bool **net_if_ipv4_addr_rm_by_index**(int index, const struct *in_addr* *addr)

Remove a IPv4 address from an interface by interface index.

Parameters

- **index** – Network interface index
- **addr** – IPv4 address

Returns
True if successfully removed, false otherwise

void **net_if_ipv4_addr_foreach**(struct *net_if* *iface, *net_if_ip_addr_cb_t* cb, void *user_data)

Go through all IPv4 addresses on a network interface and call callback for each used address.

Parameters

- **iface** – Pointer to the network interface
- **cb** – User-supplied callback function to call
- **user_data** – User specified data

struct *net_if_mcast_addr* ***net_if_ipv4_maddr_add**(struct *net_if* *iface, const struct *in_addr* *addr)

Add a IPv4 multicast address to an interface.

Parameters

- **iface** – Network interface
- **addr** – IPv4 multicast address

Returns
Pointer to interface multicast address, NULL if cannot be added

bool **net_if_ipv4_maddr_rm**(struct *net_if* *iface, const struct *in_addr* *addr)

Remove an IPv4 multicast address from an interface.

Parameters

- **iface** – Network interface
- **addr** – IPv4 multicast address

Returns
True if successfully removed, false otherwise

void **net_if_ipv4_maddr_foreach**(struct *net_if* *iface, *net_if_ip_maddr_cb_t* cb, void *user_data)

Go through all IPv4 multicast addresses on a network interface and call callback for each used address.

Parameters

- **iface** – Pointer to the network interface
- **cb** – User-supplied callback function to call
- **user_data** – User specified data

struct *net_if_mcast_addr* ***net_if_ipv4_maddr_lookup**(const struct *in_addr* *addr, struct *net_if* **iface)

Check if this IPv4 multicast address belongs to a specific interface or one of the interfaces.

Parameters

- **addr** – IPv4 address
- **iface** – If *iface is null, then pointer to interface is returned, otherwise the *iface value needs to be matched.

Returns

Pointer to interface multicast address, NULL if not found.

void **net_if_ipv4_maddr_join**(struct *net_if* *iface, struct *net_if_mcast_addr* *addr)

Mark a given multicast address to be joined.

Parameters

- **iface** – Network interface the address belongs to
- **addr** – IPv4 multicast address

static inline bool **net_if_ipv4_maddr_is_joined**(struct *net_if_mcast_addr* *addr)

Check if given multicast address is joined or not.

Parameters

- **addr** – IPv4 multicast address

Returns

True if address is joined, False otherwise.

void **net_if_ipv4_maddr_leave**(struct *net_if* *iface, struct *net_if_mcast_addr* *addr)

Mark a given multicast address to be left.

Parameters

- **iface** – Network interface the address belongs to
- **addr** – IPv4 multicast address

static inline struct *in_addr* ***net_if_router_ipv4**(struct *net_if_router* *router)

Get the IPv4 address of the given router.

Parameters

- **router** – a network router

Returns

pointer to the IPv4 address, or NULL if none

struct *net_if_router* *net_if_ipv4_router_lookup(struct *net_if* *iface, struct *in_addr* *addr)

Check if IPv4 address is one of the routers configured in the system.

Parameters

- `iface` – Network interface
- `addr` – IPv4 address

Returns
Pointer to router information, NULL if cannot be found

struct *net_if_router* *net_if_ipv4_router_find_default(struct *net_if* *iface, struct *in_addr* *addr)

Find default router for this IPv4 address.

Parameters

- `iface` – Network interface. This can be NULL in which case we go through all the network interfaces to find a suitable router.
- `addr` – IPv4 address

Returns
Pointer to router information, NULL if cannot be found

struct *net_if_router* *net_if_ipv4_router_add(struct *net_if* *iface, struct *in_addr* *addr, bool is_default, uint16_t router_lifetime)

Add IPv4 router to the system.

Parameters

- `iface` – Network interface
- `addr` – IPv4 address
- `is_default` – Is this router the default one
- `router_lifetime` – Lifetime of the router

Returns
Pointer to router information, NULL if could not be added

bool net_if_ipv4_router_rm(struct *net_if_router* *router)

Remove IPv4 router from the system.

Parameters

- `router` – Router information.

Returns
True if successfully removed, false otherwise

bool net_if_ipv4_addr_mask_cmp(struct *net_if* *iface, const struct *in_addr* *addr)

Check if the given IPv4 address belongs to local subnet.

Parameters

- `iface` – Interface to use. Must be a valid pointer to an interface.
- `addr` – IPv4 address

Returns
True if address is part of local subnet, false otherwise.

bool net_if_ipv4_is_addr_bcast(struct *net_if* *iface, const struct *in_addr* *addr)

Check if the given IPv4 address is a broadcast address.

Parameters

- **iface** – Interface to use. Must be a valid pointer to an interface.
- **addr** – IPv4 address, this should be in network byte order

Returns
True if address is a broadcast address, false otherwise.

static inline struct *net_if* ***net_if_ipv4_select_src_iface**(const struct *in_addr* *dst)

Get a network interface that should be used when sending IPv4 network data to destination.

Parameters

- **dst** – IPv4 destination address

Returns
Pointer to network interface to use, NULL if no suitable interface could be found.

static inline const struct *in_addr* ***net_if_ipv4_select_src_addr**(struct *net_if* *iface, const struct *in_addr* *dst)

Get a IPv4 source address that should be used when sending network data to destination.

Parameters

- **iface** – Interface to use when sending the packet. If the interface is not known, then NULL can be given.
- **dst** – IPv4 destination address

Returns
Pointer to IPv4 address to use, NULL if no IPv4 address could be found.

struct *in_addr* ***net_if_ipv4_get_ll**(struct *net_if* *iface, enum *net_addr_state* addr_state)

Get a IPv4 link local address in a given state.

Parameters

- **iface** – Interface to use. Must be a valid pointer to an interface.
- **addr_state** – IPv4 address state (preferred, tentative, deprecated)

Returns
Pointer to link local IPv4 address, NULL if no proper IPv4 address could be found.

struct *in_addr* ***net_if_ipv4_get_global_addr**(struct *net_if* *iface, enum *net_addr_state* addr_state)

Get a IPv4 global address in a given state.

Parameters

- **iface** – Interface to use. Must be a valid pointer to an interface.
- **addr_state** – IPv4 address state (preferred, tentative, deprecated)

Returns
Pointer to link local IPv4 address, NULL if no proper IPv4 address could be found.

struct *in_addr* **net_if_ipv4_get_netmask_by_addr**(struct *net_if* *iface, const struct *in_addr* *addr)

Get IPv4 netmask related to an address of an interface.

Parameters

- **iface** – Interface to use.

- **addr** – IPv4 address to check.

Returns
The netmask set on the interface related to the give address, unspecified address if not found.

struct *in_addr* **net_if_ipv4_get_netmask**(struct *net_if* *iface)

Get IPv4 netmask of an interface.

Deprecated:

Use *net_if_ipv4_get_netmask_by_addr()* instead.

Parameters

- **iface** – Interface to use.

Returns
The netmask set on the interface, unspecified address if not found.

void **net_if_ipv4_set_netmask**(struct *net_if* *iface, const struct *in_addr* *netmask)

Set IPv4 netmask for an interface.

Deprecated:

Use *net_if_ipv4_set_netmask_by_addr()* instead.

Parameters

- **iface** – Interface to use.
- **netmask** – IPv4 netmask

bool **net_if_ipv4_set_netmask_by_index**(int index, const struct *in_addr* *netmask)

Set IPv4 netmask for an interface index.

Deprecated:

Use *net_if_ipv4_set_netmask_by_addr()* instead.

Parameters

- **index** – Network interface index
- **netmask** – IPv4 netmask

Returns
True if netmask was added, false otherwise.

bool **net_if_ipv4_set_netmask_by_addr_by_index**(int index, const struct *in_addr* *addr, const struct *in_addr* *netmask)

Set IPv4 netmask for an interface index for a given address.

Parameters

- **index** – Network interface index
- **addr** – IPv4 address related to this netmask
- **netmask** – IPv4 netmask

Returns
True if netmask was added, false otherwise.

bool **net_if_ipv4_set_netmask_by_addr**(struct *net_if* *iface, const struct *in_addr* *addr, const struct *in_addr* *netmask)

Set IPv4 netmask for an interface index for a given address.

Parameters

- **iface** – Network interface
- **addr** – IPv4 address related to this netmask
- **netmask** – IPv4 netmask

Returns

True if netmask was added, false otherwise.

void **net_if_ipv4_set_gw**(struct *net_if* *iface, const struct *in_addr* *gw)

Set IPv4 gateway for an interface.

Parameters

- **iface** – Interface to use.
- **gw** – IPv4 address of an gateway

bool **net_if_ipv4_set_gw_by_index**(int index, const struct *in_addr* *gw)

Set IPv4 gateway for an interface index.

Parameters

- **index** – Network interface index
- **gw** – IPv4 address of an gateway

Returns

True if gateway was added, false otherwise.

struct *net_if* ***net_if_select_src_iface**(const struct *sockaddr* *dst)

Get a network interface that should be used when sending IPv6 or IPv4 network data to destination.

Parameters

- **dst** – IPv6 or IPv4 destination address

Returns

Pointer to network interface to use. Note that the function will return the default network interface if the best network interface is not found.

void **net_if_register_link_cb**(struct *net_if_link_cb* *link, *net_if_link_callback_t* cb)

Register a link callback.

Parameters

- **link** – Caller specified handler for the callback.
- **cb** – Callback to register.

void **net_if_unregister_link_cb**(struct *net_if_link_cb* *link)

Unregister a link callback.

Parameters

- **link** – Caller specified handler for the callback.

void **net_if_call_link_cb**(struct *net_if* *iface, struct *net_linkaddr* *lladdr, int status)

Call a link callback function.

Parameters

- **iface** – Network interface.

- **lladdr** – Destination link layer address
- **status** – 0 is ok, < 0 error

bool **net_if_need_calc_rx_checksum**(struct *net_if* *iface, enum *net_if_checksum_type* chksum_type)

Check if received network packet checksum calculation can be avoided or not.

For example many ethernet devices support network packet offloading in which case the IP stack does not need to calculate the checksum.

Parameters

- **iface** – Network interface
- **chksum_type** – L3 and/or L4 protocol for which to compute checksum

Returns
True if checksum needs to be calculated, false otherwise.

bool **net_if_need_calc_tx_checksum**(struct *net_if* *iface, enum *net_if_checksum_type* chksum_type)

Check if network packet checksum calculation can be avoided or not when sending the packet.

For example many ethernet devices support network packet offloading in which case the IP stack does not need to calculate the checksum.

Parameters

- **iface** – Network interface
- **chksum_type** – L3 and/or L4 protocol for which to compute checksum

Returns
True if checksum needs to be calculated, false otherwise.

struct *net_if* ***net_if_get_by_index**(int index)

Get interface according to index.

This is a syscall only to provide access to the object for purposes of assigning permissions.

Parameters

- **index** – Interface index

Returns
Pointer to interface or NULL if not found.

int **net_if_get_by_iface**(struct *net_if* *iface)

Get interface index according to pointer.

Parameters

- **iface** – Pointer to network interface

Returns
Interface index

void **net_if_foreach**(*net_if_cb_t* cb, void *user_data)

Go through all the network interfaces and call callback for each interface.

Parameters

- **cb** – User-supplied callback function to call
- **user_data** – User specified data

int **net_if_up**(struct *net_if* *iface)

Bring interface up.

Parameters

- **iface** – Pointer to network interface

Returns

0 on success

static inline bool **net_if_is_up**(struct *net_if* *iface)

Check if interface is up and running.

Parameters

- **iface** – Pointer to network interface

Returns

True if interface is up, False if it is down.

int **net_if_down**(struct *net_if* *iface)

Bring interface down.

Parameters

- **iface** – Pointer to network interface

Returns

0 on success

static inline bool **net_if_is_admin_up**(struct *net_if* *iface)

Check if interface was brought up by the administrator.

Parameters

- **iface** – Pointer to network interface

Returns

True if interface is admin up, false otherwise.

void **net_if_carrier_on**(struct *net_if* *iface)

Underlying network device has detected the carrier (cable connected).

The function should be used by the respective network device driver or L2 implementation to update its state on a network interface.

Parameters

- **iface** – Pointer to network interface

void **net_if_carrier_off**(struct *net_if* *iface)

Underlying network device has lost the carrier (cable disconnected).

The function should be used by the respective network device driver or L2 implementation to update its state on a network interface.

Parameters

- **iface** – Pointer to network interface

static inline bool **net_if_is_carrier_ok**(struct *net_if* *iface)

Check if carrier is present on network device.

Parameters

- **iface** – Pointer to network interface

Returns

True if carrier is present, false otherwise.

void **net_if_dormant_on**(struct *net_if* *iface)

Mark interface as dormant.

Dormant state indicates that the interface is not ready to pass packets yet, but is waiting for some event (for example Wi-Fi network association).

The function should be used by the respective network device driver or L2 implementation to update its state on a network interface.

Parameters

- **iface** – Pointer to network interface

void **net_if_dormant_off**(struct *net_if* *iface)

Mark interface as not dormant.

The function should be used by the respective network device driver or L2 implementation to update its state on a network interface.

Parameters

- **iface** – Pointer to network interface

static inline bool **net_if_is_dormant**(struct *net_if* *iface)

Check if the interface is dormant.

Parameters

- **iface** – Pointer to network interface

Returns

True if interface is dormant, false otherwise.

static inline int **net_if_set_promisc**(struct *net_if* *iface)

Set network interface into promiscuous mode.

Note that not all network technologies will support this.

Parameters

- **iface** – Pointer to network interface

Returns

0 on success, <0 if error

static inline void **net_if_unset_promisc**(struct *net_if* *iface)

Set network interface into normal mode.

Parameters

- **iface** – Pointer to network interface

static inline bool **net_if_is_promisc**(struct *net_if* *iface)

Check if promiscuous mode is set or not.

Parameters

- **iface** – Pointer to network interface

Returns

True if interface is in promisc mode, False if interface is not in promiscuous mode.

static inline bool **net_if_are_pending_tx_packets**(struct *net_if* *iface)

Check if there are any pending TX network data for a given network interface.

Parameters

- **iface** – Pointer to network interface

Returns
True if there are pending TX network packets for this network interface, False otherwise.

bool **net_if_is_wifi**(struct *net_if* *iface)

Check if the network interface supports Wi-Fi.

Parameters

- **iface** – Pointer to network interface

Returns
True if interface supports Wi-Fi, False otherwise.

struct *net_if* ***net_if_get_first_wifi**(void)

Get first Wi-Fi network interface.

Returns
Pointer to network interface, NULL if not found.

struct *net_if* ***net_if_get_wifi_sta**(void)

Get Wi-Fi network station interface.

Returns
Pointer to network interface, NULL if not found.

struct *net_if* ***net_if_get_wifi_sap**(void)

Get first Wi-Fi network Soft-AP interface.

Returns
Pointer to network interface, NULL if not found.

int **net_if_get_name**(struct *net_if* *iface, char *buf, int len)

Get network interface name.

If interface name support is not enabled, empty string is returned.

Parameters

- **iface** – Pointer to network interface
- **buf** – User supplied buffer
- **len** – Length of the user supplied buffer

Returns
Length of the interface name copied to buf, -EINVAL if invalid parameters, -ERANGE if name cannot be copied to the user supplied buffer, -ENOTSUP if interface name support is disabled,

int **net_if_set_name**(struct *net_if* *iface, const char *buf)

Set network interface name.

Normally this function is not needed to call as the system will automatically assign a name to the network interface.

Parameters

- **iface** – Pointer to network interface
- **buf** – User supplied name

Returns
0 name is set correctly -ENOTSUP interface name support is disabled -EINVAL if invalid parameters are given, -ENAMETOOLONG if name is too long

int **net_if_get_by_name**(const char *name)

Get interface index according to its name.

Parameters

- **name** – Name of the network interface

Returns

Interface index

struct **net_if_addr**

#include <*net_if.h*> Network Interface unicast IP addresses.

Stores the unicast IP addresses assigned to this network interface.

Public Members

struct net_addr **address**

IP address.

atomic_t **atomic_ref**

Reference counter.

This is used to prevent address removal if there are sockets that have bound the local endpoint to this address.

enum *net_addr_type* **addr_type**

How the IP address was set.

enum *net_addr_state* **addr_state**

What is the current state of the address.

uint8_t **is_infinite**

Is the IP address valid forever.

uint8_t **is_used**

Is this IP address used or not.

uint8_t **is_mesh_local**

Is this IP address usage limited to the subnet (mesh) or not.

uint8_t **is_temporary**

Is this IP address temporary and generated for example by IPv6 privacy extension (RFC 8981)

struct **net_if_mcast_addr**

#include <*net_if.h*> Network Interface multicast IP addresses.

Stores the multicast IP addresses assigned to this network interface.

Public Members

struct net_addr **address**
IP address.

uint8_t **is_used**
Is this multicast IP address used or not.

uint8_t **is_joined**
Did we join to this group.

struct **net_if_ipv6_prefix**
#include <*net_if.h*> Network Interface IPv6 prefixes.

Stores the IPV6 prefixes assigned to this network interface.

Public Members

struct *net_timeout* **lifetime**
Prefix lifetime.

struct *in6_addr* **prefix**
IPv6 prefix.

struct *net_if* ***iface**
Backpointer to network interface where this prefix is used.

uint8_t **len**
Prefix length.

uint8_t **is_infinite**
Is the IP prefix valid forever.

uint8_t **is_used**
Is this prefix used or not.

struct **net_if_router**
#include <*net_if.h*> Information about routers in the system.

Stores the router information.

Public Members

sys_snode_t **node**
Slist lifetime timer node.

struct net_addr **address**
IP address.

struct *net_if* *iface
Network interface the router is connected to.

uint32_t life_start
Router life timer start.

uint16_t lifetime
Router lifetime.

uint8_t is_used
Is this router used or not.

uint8_t is_default
Is default router.

uint8_t is_infinite
Is the router valid forever.

struct net_if_ipv6
#include <*net_if.h*> IPv6 configuration.

Public Members

struct *net_if_addr* unicast[NET_IF_MAX_IPV6_ADDR]
Unicast IP addresses.

struct *net_if_mcast_addr* mcast[NET_IF_MAX_IPV6_MADDR]
Multicast IP addresses.

struct *net_if_ipv6_prefix* prefix[NET_IF_MAX_IPV6_PREFIX]
Prefixes.

uint32_t base_reachable_time
Default reachable time (RFC 4861, page 52)

uint32_t reachable_time
Reachable time (RFC 4861, page 20)

uint32_t retrans_timer
Retransmit timer (RFC 4861, page 52)

uint8_t hop_limit
IPv6 hop limit.

uint8_t mcast_hop_limit
IPv6 multicast hop limit.

struct **net_if_addr_ipv4**

#include <*net_if.h*> Network Interface unicast IPv4 address and netmask.

Stores the unicast IPv4 address and related netmask.

Public Members

struct *net_if_addr* **ipv4**

IPv4 address.

struct *in_addr* **netmask**

Netmask.

struct **net_if_ipv4**

#include <*net_if.h*> IPv4 configuration.

Public Members

struct *net_if_addr_ipv4* **unicast**[NET_IF_MAX_IPV4_ADDR]

Unicast IP addresses.

struct *net_if_mcast_addr* **mcast**[NET_IF_MAX_IPV4_MADDR]

Multicast IP addresses.

struct *in_addr* **gw**

Gateway.

uint8_t **ttl**

IPv4 time-to-live.

uint8_t **mcast_ttl**

IPv4 time-to-live for multicast packets.

struct **net_if_ip**

#include <*net_if.h*> Network interface IP address configuration.

struct **net_if_config**

#include <*net_if.h*> IP and other configuration related data for network interface.

struct **net_traffic_class**

#include <*net_if.h*> Network traffic class.

Traffic classes are used when sending or receiving data that is classified with different priorities. So some traffic can be marked as high priority and it will be sent or received first. Each network packet that is transmitted or received goes through a fifo to a thread that will transmit it.

Public Members

struct k_fifo **fifo**

Fifo for handling this Tx or Rx packet.

struct *k_thread* **handler**

Traffic class handler thread.

k_thread_stack_t ***stack**

Stack for this handler.

struct **net_if_dev**

#include <*net_if.h*> Network Interface Device structure.

Used to handle a network interface on top of a device driver instance. There can be many *net_if_dev* instance against the same device.

Such interface is mainly to be used by the link layer, but is also tight to a network context: it then makes the relation with a network context and the network device.

Because of the strong relationship between a device driver and such network interface, each *net_if_dev* should be instantiated by one of the network device init macros found in net_if.h.

Public Members

const struct *device* ***dev**

The actually device driver instance the *net_if* is related to.

const struct *net_l2* *const **l2**

Interface's L2 layer.

void ***l2_data**

Interface's private L2 data pointer.

atomic_t **flags**[*ATOMIC_BITMAP_SIZE*(NET_IF_NUM_FLAGS)]

For internal use.

struct *net_linkaddr* **link_addr**

The hardware link address.

uint16_t **mtu**

The hardware MTU.

enum *net_if_oper_state* **oper_state**

RFC 2863 operational status.

struct **net_if**

#include <*net_if.h*> Network Interface structure.

Used to handle a network interface on top of a *net_if_dev* instance. There can be many *net_if* instance against the same *net_if_dev* instance.

Public Members

struct *net_if_dev* *`if_dev`

The *net_if_dev* instance the *net_if* is related to.

struct *net_if_config* `config`

Network interface instance configuration.

struct *k_mutex* `lock`

Mutex protecting this network interface instance.

struct *k_mutex* `tx_lock`

Mutex used when sending data.

uint8_t `pe_enabled`

Network interface specific flags.

Enable IPv6 privacy extension (RFC 8981), this is enabled by default if PE support is enabled in configuration.

uint8_t `pe_prefer_public`

If PE is enabled, then this tells whether public addresses are preferred over temporary ones for this interface.

struct `net_if_mcast_monitor`

#include <*net_if.h*> Multicast monitor handler struct.

Stores the multicast callback information. Caller must make sure that the variable pointed by this is valid during the lifetime of registration. Typically this means that the variable cannot be allocated from stack.

Public Members

sys_snode_t `node`

Node information for the slist.

struct *net_if* *`iface`

Network interface.

net_if_mcast_callback_t `cb`

Multicast callback.

struct `net_if_link_cb`

#include <*net_if.h*> Link callback handler struct.

Stores the link callback information. Caller must make sure that the variable pointed by this is valid during the lifetime of registration. Typically this means that the variable cannot be allocated from stack.

Public Members

sys_snode_t node
Node information for the slist.

net_if_link_callback_t cb
Link callback.

L2 Layer Management

- *Overview*
- *L2 layer API*
- *Network Device drivers*
  - *Ethernet device driver*
  - *IEEE 802.15.4 device driver*
- *API Reference*

Overview The L2 stack is designed to hide the whole networking link-layer part and the related device drivers from the upper network stack. This is made through a *net_if* declared in include/zephyr/net/net_if.h.

The upper layers are unaware of implementation details beyond the net_if object and the generic API provided by the L2 layer in include/zephyr/net/net_l2.h as *net_l2*.

Only the L2 layer can talk to the device driver, linked to the net_if object. The L2 layer dictates the API provided by the device driver, specific for that device, and optimized for working together.

Currently, there are L2 layers for *Ethernet*, *IEEE 802.15.4 Soft-MAC*, *CANBUS*, *OpenThread*, Wi-Fi, and a dummy layer example that can be used as a template for writing a new one.

L2 layer API In order to create an L2 layer, or a driver for a specific L2 layer, one needs to understand how the L3 layer interacts with it and how the L2 layer is supposed to behave. See also *network stack architecture* for more details. The generic L2 API has these functions:

- `recv()`: All device drivers, once they receive a packet which they put into a *net_pkt*, will push this buffer to the network stack via *net_recv_data()*. At this point, the network stack does not know what to do with it. Instead, it passes the buffer along to the L2 stack's `recv()` function for handling. The L2 stack does what it needs to do with the packet, for example, parsing the link layer header, or handling link-layer only packets. The `recv()` function will return `NET_DROP` in case of an erroneous packet, `NET_OK` if the packet was fully consumed by the L2, or `NET_CONTINUE` if the network stack should then handle it.
- `send()`: Similar to receive function, the network stack will call this function to actually send a network packet. All relevant link-layer content will be generated and added by this function. The `send()` function returns the number of bytes sent, or a negative error code if there was a failure sending the network packet.
- `enable()`: This function is used to enable/disable traffic over a network interface. The function returns `<0` if error and `>=0` if no error.
- `get_flags()`: This function will return the capabilities of an L2 driver, for example whether the L2 supports multicast or promiscuous mode.

Network Device drivers Network device drivers fully follows Zephyr device driver model as a basis. Please refer to *Device Driver Model*.

There are, however, two differences:

- The driver_api pointer must point to a valid `net_if_api` pointer.
- The network device driver must use *NET_DEVICE_INIT_INSTANCE()* or *ETH_NET_DEVICE_INIT()* for Ethernet devices. These macros will call the *DEVICE_DEFINE()* macro, and also instantiate a unique *net_if* related to the created device driver instance.

Implementing a network device driver depends on the L2 stack it belongs to: *Ethernet*, *IEEE 802.15.4*, etc. In the next section, we will describe how a device driver should behave when receiving or sending a network packet. The rest is hardware dependent and is not detailed here.

Ethernet device driver On reception, it is up to the device driver to fill-in the network packet with as many data buffers as required. The network packet itself is a *net_pkt* and should be allocated through `net_pkt_rx_alloc_with_buffer()`. Then all data buffers will be automatically allocated and filled by *net_pkt_write()*.

After all the network data has been received, the device driver needs to call *net_recv_data()*. If that call fails, it will be up to the device driver to unreference the buffer via *net_pkt_unref()*.

On sending, the device driver send function will be called, and it is up to the device driver to send the network packet all at once, with all the buffers.

Each Ethernet device driver will need, in the end, to call `ETH_NET_DEVICE_INIT()` like this:

```
ETH_NET_DEVICE_INIT(..., CONFIG_ETH_INIT_PRIORITY,
                    &the_valid_net_if_api_instance, 1500);
```

IEEE 802.15.4 device driver Device drivers for IEEE 802.15.4 L2 work basically the same as for Ethernet. What has been described above, especially for `recv()`, applies here as well. There are two specific differences however:

- It requires a dedicated device driver API: *ieee802154_radio_api*, which overloads `net_if_api`. This is because 802.15.4 L2 needs more from the device driver than just `send()` and `recv()` functions. This dedicated API is declared in include/zephyr/net/ieee802154_radio.h. Each and every IEEE 802.15.4 device driver must provide a valid pointer on such relevantly filled-in API structure.
- Sending a packet is slightly different than in Ethernet. Most IEEE 802.15.4 PHYs support relatively small frames only, 127 bytes all inclusive: frame header, payload and frame checksum. Buffers to be sent over the radio will often not fit this frame size limitation, e.g. a buffer containing an IPv6 packet will often have to be split into several fragments and IP6 packet headers and fragments need to be compressed using a protocol like 6LoWPAN before being passed on to the radio driver. Additionally the IEEE 802.15.4 standard defines medium access (e.g. CSMA/CA), frame retransmission, encryption and other pre-processing procedures (e.g. addition of information elements) that individual radio drivers should not have to care about. This is why the *ieee802154_radio_api* requires a tx function pointer which differs from the `net_if_api` send function pointer. Zephyr's native IEEE 802.15.4 L2 implementation provides a generic `ieee802154_send()` instead, meant to be given as *net_if* send function. The implementation of `ieee802154_send()` takes care of IEEE 802.15.4 standard packet preparation procedures, splitting the packet into possibly compressed, encrypted and otherwise pre-processed fragment buffers, sending one buffer at a time through *ieee802154_radio_api* tx function and unreferencing the network packet only when the transmission as a whole was either successful or failed.

Interaction between IEEE 802.15.4 radio device drivers and L2 is bidirectional:

- L2 -> L1: Methods as `ieee802154_send()` and several IEEE 802.15.4 net management calls will call into the driver, e.g. to send a packet over the radio link or re-configure

the driver at runtime. These incoming calls will all be handled by the methods in the *ieee802154_radio_api*.

- L1 -> L2: There are several situations in which the driver needs to initiate calls into the L2/MAC layer. Zephyr's IEEE 802.15.4 L1 -> L2 adaptation API employs an "inversion-of-control" pattern in such cases avoids duplication of complex logic across independent driver implementations and ensures implementation agnostic loose coupling and clean separation of concerns between MAC (L2) and PHY (L1) whenever reverse information transfer or close co-operation between hardware and L2 is required. During driver initialization, for example, the driver calls *ieee802154_init()* to pass the interface's MAC address as well as other hardware-related configuration to L2. Similarly, drivers may indicate performance or timing critical radio events to L2 that require close integration with the hardware (e.g. *ieee802154_handle_ack()*). Calls from L1 into L2 are not implemented as methods in *ieee802154_radio_api* but are standalone functions declared and documented as such in include/zephyr/net/ieee802154_radio.h. The API documentation will clearly state which functions must be implemented by all L2 stacks as part of the L1 -> L2 "inversion-of-control" adaptation API.

Note: Standalone functions in include/zephyr/net/ieee802154_radio.h that are not explicitly documented as callbacks are considered to be helper functions within the PHY (L1) layer implemented independently of any specific L2 stack, see for example *ieee802154_is_ar_flag_set()*.

As all net interfaces, IEEE 802.15.4 device driver implementations will have to call `NET_DEVICE_INIT_INSTANCE()` in the end:

```
NET_DEVICE_INIT_INSTANCE(...,
                         the_device_init_prio,
                         &the_valid_ieee802154_radio_api_instance,
                         IEEE802154_L2,
                         NET_L2_GET_CTX_TYPE(IEEE802154_L2), 125);
```

Related code samples

Link Layer Discovery Protocol (LLDP)
Enable LLDP support and setup VLANs.

Virtual LAN
Setup two virtual LAN networks and use net-shell to view the networks' settings.

API Reference

group `net_l2`

Network Layer 2 abstraction layer.

Enums

enum `net_l2_flags`

L2 flags.

Values:

enumerator `NET_L2_MULTICAST` = *BIT*(0)

IP multicast supported.

enumerator NET_L2_MULTICAST_SKIP_JOIN_SOLICIT_NODE = *BIT*(1)

Do not join solicited node multicast group.

enumerator NET_L2_PROMISC_MODE = *BIT*(2)

Is promiscuous mode supported.

enumerator NET_L2_POINT_TO_POINT = *BIT*(3)

Is this L2 point-to-point with tunneling so no need to have IP address etc to network interface.

struct net_l2

#include <net_l2.h> Network L2 structure.

Used to provide an interface to lower network stack.

Public Members

enum *net_verdict* (*recv)(struct *net_if* *iface, struct *net_pkt* *pkt)

This function is used by net core to get iface's L2 layer parsing what's relevant to itself.

int (*send)(struct *net_if* *iface, struct *net_pkt* *pkt)

This function is used by net core to push a packet to lower layer (interface's L2), which in turn might work on the packet relevantly.

(adding proper header etc...) Returns a negative error code, or the number of bytes sent otherwise.

int (*enable)(struct *net_if* *iface, bool state)

This function is used to enable/disable traffic over a network interface.

The function returns <0 if error and >=0 if no error.

enum *net_l2_flags* (*get_flags)(struct *net_if* *iface)

Return L2 flags for the network interface.

Network Traffic Offloading

- *Network Offloading*
  - *Overview*
  - *API Reference*
- *Socket Offloading*
  - *Overview*

Network Offloading

Overview The network offloading API provides hooks that a device vendor can use to provide an alternate implementation for an IP stack. This means that the actual network connection creation, data transfer, etc., is done in the vendor HAL instead of the Zephyr network stack.

API Reference

group `net_offload`

Network offloading interface.

Socket Offloading

Overview In addition to the network offloading API, Zephyr allows offloading of networking functionality at the socket API level. With this approach, vendors who provide an alternate implementation of the networking stack, exposing socket API for their networking devices, can easily integrate it with Zephyr.

See drivers/wifi/simplelink/simplelink_sockets.c for a sample implementation on how to integrate network offloading at socket level.

Link Layer Address Handling

- *Overview*
- *API Reference*

Overview The link layer addresses are set for network interfaces so that L2 connectivity works correctly in the network stack. Typically the link layer addresses are 6 bytes long like in Ethernet but for IEEE 802.15.4 the link layer address length is 8 bytes.

API Reference

group `net_linkaddr`

Network link address library.

Defines

`NET_LINK_ADDR_MAX_LENGTH`

Maximum length of the link address.

Enums

enum `net_link_type`

Type of the link address.

This indicates the network technology that this address is used in. Note that in order to save space we store the value into a uint8_t variable, so please do not introduce any values > 255 in this enum.

Values:

enumerator NET_LINK_UNKNOWN = 0

Unknown link address type.

enumerator NET_LINK_IEEE802154

IEEE 802.15.4 link address.

enumerator NET_LINK_BLUETOOTH

Bluetooth IPSP link address.

enumerator NET_LINK_ETHERNET

Ethernet link address.

enumerator NET_LINK_DUMMY

Dummy link address.

Used in testing apps and loopback support.

enumerator NET_LINK_CANBUS_RAW

CANBUS link address.

Functions

static inline bool **net_linkaddr_cmp**(struct *net_linkaddr* *lladdr1, struct *net_linkaddr* *lladdr2)

Compare two link layer addresses.

Parameters

- **lladdr1** – Pointer to a link layer address
- **lladdr2** – Pointer to a link layer address

Returns

True if the addresses are the same, false otherwise.

static inline int **net_linkaddr_set**(struct *net_linkaddr_storage* *lladdr_store, uint8_t *new_addr, uint8_t new_len)

Set the member data of a link layer address storage structure.

Parameters

- **lladdr_store** – The link address storage structure to change.
- **new_addr** – Array of bytes containing the link address.
- **new_len** – Length of the link address array. This value should always be <= NET_LINK_ADDR_MAX_LENGTH.

struct **net_linkaddr**

#include <net_linkaddr.h> Hardware link address structure.

Used to hold the link address information

Public Members

uint8_t *addr
: The array of byte representing the address.

uint8_t len
: Length of that address array.

uint8_t type
: What kind of address is this for.

struct net_linkaddr_storage
: *#include* <*net_linkaddr.h*> Hardware link address structure.

  Used to hold the link address information. This variant is needed when we have to store the link layer address.

  Note that you cannot cast this to *net_linkaddr* as uint8_t * is handled differently than uint8_t addr[] and the fields are purposely in different order.

Public Members

uint8_t type
: What kind of address is this for.

uint8_t len
: The real length of the ll address.

uint8_t addr[6]
: The array of bytes representing the address.

Ethernet Management

- *Overview*
- *API Reference*

Overview Ethernet management API provides functions to manage the Ethernet network interface low level status. The caller of these functions can:

- raise `carrier ON` or `carrier OFF` management events
- raise `VLAN enabled` or `VLAN disabled` management events

Typically the `carrier OFF` event would be generated by the Ethernet device driver when it notices that the Ethernet cable is disconnected. The `carrier ON` event would be generated if the Ethernet device driver notices that the Ethernet cable is re-connected.

Currently the VLAN events are generated by the Ethernet L2 layer when a specific VLAN tag is either enabled or disabled.

The user application can monitor these events if it needs to act when the corresponding status changes.

API Reference

group `ethernet_mgmt`

Ethernet library.

Functions

void `ethernet_mgmt_raise_carrier_on_event`(struct *net_if* *iface)

Raise CARRIER_ON event when Ethernet is connected.

Parameters

- `iface` – Ethernet network interface.

void `ethernet_mgmt_raise_carrier_off_event`(struct *net_if* *iface)

Raise CARRIER_OFF event when Ethernet is disconnected.

Parameters

- `iface` – Ethernet network interface.

void `ethernet_mgmt_raise_vlan_enabled_event`(struct *net_if* *iface, uint16_t tag)

Raise VLAN_ENABLED event when VLAN is enabled.

Parameters

- `iface` – Ethernet network interface.
- `tag` – VLAN tag which is enabled.

void `ethernet_mgmt_raise_vlan_disabled_event`(struct *net_if* *iface, uint16_t tag)

Raise VLAN_DISABLED event when VLAN is disabled.

Parameters

- `iface` – Ethernet network interface.
- `tag` – VLAN tag which is disabled.

Traffic Classification

Overview Traffic classification is an automated process that categorizes computer network traffic according to various parameters. For Zephyr, the VLAN priority code point (PCP) is used to classify both received and sent network packets. See more information about VLAN priority at IEEE 802.1Q.

By default, all network traffic is treated equal in Zephyr. If desired, the option `CONFIG_NET_TC_TX_COUNT` can be used to set the number of transmit queues. The option `CONFIG_NET_TC_RX_COUNT` can be used to set the number of receive queues. Each traffic class queue corresponds to a specific kernel work queue. Each kernel work queue has a priority. The VLAN priority is mapped to a certain traffic class according to rules specified in IEEE 802.1Q spec chapter I.3, chapter 8.6.6 table 8-4, and chapter 34.5 table 34-1. Each traffic class is in turn mapped to a certain kernel work queue. The maximum number of traffic classes for both Rx and Tx is 8.

See subsys/net/ip/net_tc.c for details of how various mappings are done.

Network Packet Filtering

- *Overview*
- *Examples*
- *API Reference*

Overview The Network Packet Filtering facility provides the infrastructure to construct custom rules for accepting and/or denying packet transmission and reception. This can be used to create a basic firewall, control network traffic, etc.

The `CONFIG_NET_PKT_FILTER` must be set in order to enable the relevant APIs.

Both the transmission and reception paths may have a list of filter rules. Each rule is made of a set of conditions and a packet outcome. Every packet is subjected to the conditions attached to a rule. When all the conditions for a given rule are true then the packet outcome is immediately determined as specified by the current rule and no more rules are considered. If one condition is false then the next rule in the list is considered.

Packet outcome is either `NET_OK` to accept the packet or `NET_DROP` to drop it.

A rule is represented by a *`npf_rule`* object. It can be inserted to, appended to or removed from a rule list contained in a *`npf_rule_list`* object using *`npf_insert_rule()`*, *`npf_append_rule()`*, and *`npf_remove_rule()`*. Currently, two such rule lists exist: `npf_send_rules` for outgoing packets, and `npf_recv_rules` for incoming packets.

If a filter rule list is empty then `NET_OK` is assumed. If a non-empty rule list runs to the end then `NET_DROP` is assumed. However it is recommended to always terminate a non-empty rule list with an explicit default termination rule, either `npf_default_ok` or `npf_default_drop`.

Rule conditions are represented by a *`npf_test`*. This structure can be embedded into a larger structure when a specific condition requires extra test data. It is up to the test function for such conditions to retrieve the outer structure from the provided `npf_test` structure pointer.

Convenience macros are provided in include/zephyr/net/net_pkt_filter.h to statically define condition instances for various conditions, and *`NPF_RULE()`* to create a rule instance to tie them.

Examples Here's an example usage:

```
static NPF_SIZE_MAX(maxsize_200, 200);
static NPF_ETH_TYPE_MATCH(ip_packet, NET_ETH_PTYPE_IP);

static NPF_RULE(small_ip_pkt, NET_OK, ip_packet, maxsize_200);

void install_my_filter(void)
{
    npf_insert_recv_rule(&npf_default_drop);
    npf_insert_recv_rule(&small_ip_pkt);
}
```

The above would accept IP packets that are 200 bytes or smaller, and drop all other packets.

Another (less efficient) way to achieve the same result could be:

```
static NPF_SIZE_MIN(minsize_201, 201);
static NPF_ETH_TYPE_UNMATCH(not_ip_packet, NET_ETH_PTYPE_IP);

static NPF_RULE(reject_big_pkts, NET_DROP, minsize_201);
```

(continues on next page)

(continued from previous page)

```
static NPF_RULE(reject_non_ip, NET_DROP, not_ip_packet);

void install_my_filter(void) {
    npf_append_recv_rule(&reject_big_pkts);
    npf_append_recv_rule(&reject_non_ip);
    npf_append_recv_rule(&npf_default_ok);
}
```

API Reference

group net_pkt_filter

Network Packet Filter API.

Defines

NPF_RULE(_name, _result, ...)

Statically define one packet filter rule.

This creates a rule from a variable amount of filter conditions. This rule can then be inserted or appended to the rule list for a given network packet path.

Example:

```
static NPF_SIZE_MAX(maxsize_200, 200);
static NPF_ETH_TYPE_MATCH(ip_packet, NET_ETH_PTYPE_IP);

static NPF_RULE(small_ip_pkt, NET_OK, ip_packet, maxsize_200);

void install_my_filter(void)
{
    npf_insert_recv_rule(&npf_default_drop);
    npf_insert_recv_rule(&small_ip_pkt);
}
```

The above would accept IP packets that are 200 bytes or smaller, and drop all other packets.

Another (less efficient) way to create the same result could be:

```
static NPF_SIZE_MIN(minsize_201, 201);
static NPF_ETH_TYPE_UNMATCH(not_ip_packet, NET_ETH_PTYPE_IP);

static NPF_RULE(reject_big_pkts, NET_DROP, minsize_201);
static NPF_RULE(reject_non_ip, NET_DROP, not_ip_packet);

void install_my_filter(void) {
    npf_append_recv_rule(&reject_big_pkts);
    npf_append_recv_rule(&reject_non_ip);
    npf_append_recv_rule(&npf_default_ok);
}
```

The first rule in the list for which all conditions are true determines the fate of the packet. If one condition is false then the next rule in the list is evaluated.

Parameters

- **_name** – Name for this rule.
- **_result** – Fate of the packet if all conditions are true, either NET_OK or NET_DROP.

- ... – List of conditions for this rule.

Functions

void `npf_insert_rule`(struct *npf_rule_list* *rules, struct *npf_rule* *rule)

Insert a rule at the front of given rule list.

Parameters

- `rules` – the affected rule list
- `rule` – the rule to be inserted

void `npf_append_rule`(struct *npf_rule_list* *rules, struct *npf_rule* *rule)

Append a rule at the end of given rule list.

Parameters

- `rules` – the affected rule list
- `rule` – the rule to be appended

bool `npf_remove_rule`(struct *npf_rule_list* *rules, struct *npf_rule* *rule)

Remove a rule from the given rule list.

Parameters

- `rules` – the affected rule list
- `rule` – the rule to be removed

Return values

`true` – if given rule was found in the rule list and removed

bool `npf_remove_all_rules`(struct *npf_rule_list* *rules)

Remove all rules from the given rule list.

Parameters

- `rules` – the affected rule list

Return values

`true` – if at least one rule was removed from the rule list

Variables

struct *npf_rule* `npf_default_ok`

Default rule list termination for accepting a packet.

struct *npf_rule* `npf_default_drop`

Default rule list termination for rejecting a packet.

struct *npf_rule_list* `npf_send_rules`

rule list applied to outgoing packets

struct *npf_rule_list* `npf_recv_rules`

rule list applied to incoming packets

struct *npf_rule_list* **npf_local_in_recv_rules**
rule list applied for local incoming packets

struct *npf_rule_list* **npf_ipv4_recv_rules**
rule list applied for IPv4 incoming packets

struct *npf_rule_list* **npf_ipv6_recv_rules**
rule list applied for IPv6 incoming packets

struct **npf_test**
#include <net_pkt_filter.h> common filter test structure to be embedded into larger structures

Public Members

npf_test_fn_t ***fn**
packet condition test function

struct **npf_rule**
#include <net_pkt_filter.h> filter rule structure

Public Members

sys_snode_t **node**
Slist rule list node.

enum *net_verdict* **result**
result if all tests pass

uint32_t **nb_tests**
number of tests for this rule

struct *npf_test* ***tests**[]
pointers to *npf_test* instances

struct **npf_rule_list**
#include <net_pkt_filter.h> rule set for a given test location

Public Members

sys_slist_t **rule_head**
List head.

struct *k_spinlock* **lock**
Lock protecting the list access.

group **npf_basic_cond**

Defines

NPF_IFACE_MATCH(_name, _iface)

Statically define an “interface match” packet filter condition.

Parameters

- **_name** – Name of the condition
- **_iface** – Interface to match

NPF_IFACE_UNMATCH(_name, _iface)

Statically define an “interface unmatch” packet filter condition.

Parameters

- **_name** – Name of the condition
- **_iface** – Interface to exclude

NPF_ORIG_IFACE_MATCH(_name, _iface)

Statically define an “orig interface match” packet filter condition.

Parameters

- **_name** – Name of the condition
- **_iface** – Interface to match

NPF_ORIG_IFACE_UNMATCH(_name, _iface)

Statically define an “orig interface unmatch” packet filter condition.

Parameters

- **_name** – Name of the condition
- **_iface** – Interface to exclude

NPF_SIZE_MIN(_name, _size)

Statically define a “data minimum size” packet filter condition.

Parameters

- **_name** – Name of the condition
- **_size** – Lower bound of the packet’s data size

NPF_SIZE_MAX(_name, _size)

Statically define a “data maximum size” packet filter condition.

Parameters

- **_name** – Name of the condition
- **_size** – Higher bound of the packet’s data size

NPF_SIZE_BOUNDS(_name, _min_size, _max_size)

Statically define a “data bounded size” packet filter condition.

Parameters

- **_name** – Name of the condition
- **_min_size** – Lower bound of the packet’s data size
- **_max_size** – Higher bound of the packet’s data size

NPF_IP_SRC_ADDR_ALLOWLIST(_name, _ip_addr_array, _ip_addr_num, _af)

Statically define a “ip address allowlist” packet filter condition.

This tests if the packet source ip address matches any of the ip addresses contained in the provided set.

Parameters

- **_name** – Name of the condition
- **_ip_addr_array** – Array of struct *in_addr* or struct *in6_addr* items to test against
- **_ip_addr_num** – number of IP addresses in the array
- **_af** – Addresses family type (AF_INET / AF_INET6) in the array

NPF_IP_SRC_ADDR_BLOCKLIST(_name, _ip_addr_array, _ip_addr_num, _af)

Statically define a “ip address blocklist” packet filter condition.

This tests if the packet source ip address matches any of the ip addresses contained in the provided set.

Parameters

- **_name** – Name of the condition
- **_ip_addr_array** – Array of struct *in_addr* or struct *in6_addr* items to test against
- **_ip_addr_num** – number of IP addresses in the array
- **_af** – Addresses family type (AF_INET / AF_INET6) in the array

group npf_eth_cond

Defines

NPF_ETH_SRC_ADDR_MATCH(_name, _addr_array)

Statically define a “source address match” packet filter condition.

This tests if the packet source address matches any of the Ethernet addresses contained in the provided set.

Parameters

- **_name** – Name of the condition
- **_addr_array** – Array of struct *net_eth_addr* items to test against

NPF_ETH_SRC_ADDR_UNMATCH(_name, _addr_array)

Statically define a “source address unmatch” packet filter condition.

This tests if the packet source address matches none of the Ethernet addresses contained in the provided set.

Parameters

- **_name** – Name of the condition
- **_addr_array** – Array of struct *net_eth_addr* items to test against

NPF_ETH_DST_ADDR_MATCH(_name, _addr_array)

Statically define a “destination address match” packet filter condition.

This tests if the packet destination address matches any of the Ethernet addresses contained in the provided set.

Parameters

- **_name** – Name of the condition
- **_addr_array** – Array of `struct` *net_eth_addr* items to test against

`NPF_ETH_DST_ADDR_UNMATCH`(_name, _addr_array)

Statically define a "destination address unmatch" packet filter condition.

This tests if the packet destination address matches none of the Ethernet addresses contained in the provided set.

Parameters

- **_name** – Name of the condition
- **_addr_array** – Array of `struct` *net_eth_addr* items to test against

`NPF_ETH_SRC_ADDR_MASK_MATCH`(_name, _addr_array, ...)

Statically define a "source address match with mask" packet filter condition.

This tests if the packet source address matches any of the Ethernet addresses contained in the provided set after applying specified mask.

Parameters

- **_name** – Name of the condition
- **_addr_array** – Array of `struct` *net_eth_addr* items to test against
- **...** – up to 6 mask bytes

`NPF_ETH_DST_ADDR_MASK_MATCH`(_name, _addr_array, ...)

Statically define a "destination address match with mask" packet filter condition.

This tests if the packet destination address matches any of the Ethernet addresses contained in the provided set after applying specified mask.

Parameters

- **_name** – Name of the condition
- **_addr_array** – Array of `struct` *net_eth_addr* items to test against
- **...** – up to 6 mask bytes

`NPF_ETH_TYPE_MATCH`(_name, _type)

Statically define an "Ethernet type match" packet filter condition.

Parameters

- **_name** – Name of the condition
- **_type** – Ethernet type to match

`NPF_ETH_TYPE_UNMATCH`(_name, _type)

Statically define an "Ethernet type unmatch" packet filter condition.

Parameters

- **_name** – Name of the condition
- **_type** – Ethernet type to exclude

Network Shell Network shell provides helpers for figuring out network status, enabling/disabling features, and issuing commands like ping or DNS resolving. Note that `net-shell` should probably not be used in production code as it will require extra memory. See also *generic shell* for detailed shell information.

The following net-shell commands are implemented:

Table 36: net-shell commands

Command	Description
net allocs	Print network memory allocations. Only available if CONFIG_NET_DEBUG_NET_PKT_ALLOC is set.
net arp	Print information about IPv4 ARP cache. Only available if CONFIG_NET_ARP is set in IPv4 enabled networks.
net capture	Monitor network traffic See *Monitor Network Traffic* for details.
net conn	Print information about network connections.
net dns	Show how DNS is configured. The command can also be used to resolve a DNS name. Only available if CONFIG_DNS_RESOLVER is set.
net events	Enable network event monitoring. Only available if CONFIG_NET_MGMT_EVENT_MONITOR is set.
net gptp	Print information about gPTP support. Only available if CONFIG_NET_GPTP is set.
net iface	Print information about network interfaces.
net ipv6	Print IPv6 specific information and configuration. Only available if CONFIG_NET_IPV6 is set.
net mem	Print information about network memory usage. The command will print more information if CONFIG_NET_BUF_POOL_USAGE is set.
net nbr	Print neighbor information. Only available if CONFIG_NET_IPV6 is set.
net ping	Ping a network host.
net route	Show IPv6 network routes. Only available if CONFIG_NET_ROUTE is set.
net sockets	Show network socket information and statistics. Only available if CONFIG_NET_SOCKETS_OBJ_CORE and CONFIG_OBJ_CORE are set.
net stats	Show network statistics.
net tcp	Connect/send data/close TCP connection. Only available if CONFIG_NET_TCP is set.
net vlan	Show Ethernet virtual LAN information. Only available if CONFIG_NET_VLAN is set.

TLS Credentials Shell The TLS Credentials shell provides a command-line interface for managing installed TLS credentials.

Commands

Buffer Credential (buf) Buffer data incrementally into the credential buffer so that it can be added using the *Add Credential (add)* command.

Alternatively, clear the credential buffer.

Usage To append <DATA> to the credential buffer, use:

```
cred buf <DATA>
```

Use this as many times as needed to load the full credential into the credential buffer, then use the *Add Credential (add)* command to store it.

To clear the credential buffer, use:

```
cred buf clear
```

Arguments

Argument	Description
<DATA>	Text data to be appended to credential buffer. It can be either text, or base64-encoded binary. See *Add Credential (add)* and *Storage/Retrieval Formats* for details.

Add Credential (add) Add a TLS credential to the TLS Credential store.

Credential contents can be provided in-line with the call to `cred add`, or will otherwise be sourced from the credential buffer.

Usage To add a TLS credential using the data from the credential buffer, use:

```
cred add <SECTAG> <TYPE> <BACKEND> <FORMAT>
```

To add a TLS credential using data provided with the same command, use:

```
cred add <SECTAG> <TYPE> <BACKEND> <FORMAT> <DATA>
```

Arguments

Argument	Description
<SECTAG>	The sectag to use for the new credential. Can be any non-negative integer.
<TYPE>	The type of credential to add. See *Credential Types* for valid values.
<BACKEND>	Reserved. Must always be `DEFAULT` (case-insensitive).
<FORMAT>	Specifies the storage format of the provided credential. See *Storage/Retrieval Formats* for valid values.
<DATA>	If provided, this argument will be used as the credential data, instead of any data in the credential buffer. Can be either text, or base64-encoded binary.

Delete Credential (del) Delete a specified credential from the credential store.

Usage To delete a credential matching a specified sectag and credential type (if it exists), use:

```
cred del <SECTAG> <TYPE>
```

Arguments

Argument	Description
<SECTAG>	The sectag of the credential to delete. Can be any non-negative integer.
<TYPE>	The type of credential to delete. See *Credential Types* for valid values.

Get Credential Contents (get) Retrieve and print the contents of a specified credential.

Usage To retrieve and print a credential matching a specified sectag and credential type (if it exists), use:

```
cred get <SECTAG> <TYPE> <FORMAT>
```

Arguments

Argument	Description
<SECTAG>	The sectag of the credential to get. Can be any non-negative integer.
<TYPE>	The type of credential to get. See *Credential Types* for valid values.
<FORMAT>	Specifies the retrieval format for the provided credential. See *Storage/Retrieval Formats* for valid values.

List Credentials (`list`) List TLS credentials in the credential store.

Usage To list all available credentials, use:

```
cred list
```

To list all credentials with a specified sectag, use:

```
cred list <SECTAG>
```

To list all credentials with a specified credential type, use:

```
cred list any <TYPE>
```

To list all credentials with a specified credential type and sectag, use:

```
cred list <SECTAG> <TYPE>
```

Arguments

Argument	Description
<SECTAG>	Optional. If provided, only list credentials with this sectag. Pass any or omit to allow any sectag. Otherwise, can be any non-negative integer.
<TYPE>	Optional. If provided, only list credentials with this credential type. Pass any or omit to allow any credential type. Otherwise, see *Credential Types* for valid values.

Output The command outputs all matching credentials in the following (CSV-compliant) format:

```
<SECTAG>,<TYPE>,<DIGEST>,<STATUS>
```

Where:

Symbol	Value
<SECTAG>	The sectag of the listed credential. A non-negative integer.
<TYPE>	Credential type short-code (see *Credential Types* for details) of the listed credential.
<DIGEST>	A string digest representing the credential contents. The exact nature of this digest may vary depending on credentials storage backend, but currently for all backends this is a base64 encoded SHA256 hash of the raw credential contents (so different storage formats for essentially identical credentials will have different digests).
<STATUS>	Status code indicating success or failure with generating a digest of the listed credential. 0 if successful, negative error code specific to the storage backend otherwise. Lines for which status is not zero will be printed with error formatting.

After the list is printed, a final summary of the found credentials will be printed in the form:

```
<N> credentials found.
```

Where <*N*> is the number of credentials found, and is zero if none are found.

Credential Types The following keywords (case-insensitive) may be used to specify a credential type:

Keyword(s)	Meaning
CA_CERT, CA	A trusted CA certificate.
SERVER_CERT, SELF_CERT, CLIENT_CERT, CLIENT, SELF, SERV	Self or server certificate.
PRIVATE_KEY, PK	A private key.
PRE_SHARED_K PSK	A pre-shared key.
PRE_SHARED_K PSK_ID	ID for pre-shared key.

Storage/Retrieval Formats The *tls_credentials* module treats stored credentials as arbitrary binary buffers.

For convenience, the TLS credentials shell offers four formats for providing and later retrieving these buffers using the shell.

These formats and their (case-insensitive) keywords are as follows:

Kw	Meaning	Behavior during storage (cred add)	Behavior during retrieval (cred get)
BI	Credential is handled by shell as base64 and stored without NULL termination.	Data entered into shell will be decoded from base64 into raw binary before storage. No terminator will be appended.	Stored data will be encoded into base64 before being printed.
BI	Credential is handled by shell as base64 and stored with NULL termination.	Data entered into shell will be decoded from base64 into raw binary and a NULL terminator will be appended before storage.	NULL terminator will be truncated from stored data before said data is encoded into base64 and then printed.
ST	Credential is handled by shell as literal string and stored without NULL termination.	Text data entered into shell will be passed into storage as-written, without a NULL terminator.	Stored data will be printed as text. Non-printable characters will be printed as ?
ST	Credential is handled by shell as literal string and stored with NULL-termination.	Text data entered into shell will be passed into storage as-written, with a NULL terminator.	NULL terminator will be truncated from stored data before said data is printed as text. Non-printable characters will be printed as ?

The `BIN` format can be used to install credentials of any type, since base64 can be used to encode any concievable binary buffer. The remaining three formats are provided for convenience in special use-cases.

For example:

- To install printable pre-shared-keys, use `STR` to enter the PSK without first encoding it. This ensures it is stored without a NULL terminator.
- To install DER-formatted X.509 certificates (or other raw-binary credentials, such as non-printable PSKs) base64-encode the binary and use the `BIN` format.
- To install PEM-formatted X.509 certificates or certificate chains, base64 encode the full PEM string (including new-lines and `----BEGIN X ----` / `----END X----` markers), and then use the `BINT` format to make sure the stored string is NULL-terminated. This is required because Zephyr does not support multi-line strings in the shell. Otherwise, the `STRT` format could be used for this purpose without base64 encoding. It is possible to use `BIN` instead if you manually encode a NULL terminator into the base64.

Time Sensitive Networking

generic Precision Time Protocol (gPTP)

- *Overview*
- *Supported features*
- *Supported hardware*
- *Enabling the stack*
- *Application interfaces*
- *Testing*
- *API Reference*

Overview This gPTP stack supports the protocol and procedures as defined in the IEEE 802.1AS-2011 standard (Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks).

Supported features The stack handles communications and state machines defined in the IEEE 802.1AS-2011 standard. Mandatory requirements for a full-duplex point-to-point link end-point, as defined in Annex A of the standard, are supported.

The stack is in principle capable of handling communications on multiple network interfaces (also defined as "ports" in the standard) and thus act as a 802.1AS bridge. However, this mode of operation has not been validated on the Zephyr OS.

Supported hardware Although the stack itself is hardware independent, Ethernet frame timestamping support must be enabled in ethernet drivers.

Boards supported:

- frdm_k64f
- nucleo_h743zi_board
- nucleo_h745zi_q_board
- nucleo_f767zi_board
- sam_e70_xplained
- native_sim (only usable for simple testing, limited capabilities due to lack of hardware clock)
- qemu_x86 (emulated, limited capabilities due to lack of hardware clock)

Enabling the stack The following configuration option must me enabled in `prj.conf` file.

- `CONFIG_NET_GPTP`

Application interfaces Only two Application Interfaces as defined in section 9 of the standard are available:

- `ClockTargetPhaseDiscontinuity` interface (*`gptp_register_phase_dis_cb()`*)
- `ClockTargetEventCapture` interface (*`gptp_event_capture()`*)

Testing The stack has been informally tested using the OpenAVnu gPTP and Linux ptp4l daemons. The gPTP sample application from the Zephyr source distribution can be used for testing.

Related code samples

gPTP
Enable gPTP support and monitor functionality using net-shell.

API Reference

group **gptp**

generic Precision Time Protocol (gPTP) support

Typedefs

typedef void (*`gptp_phase_dis_callback_t`)(uint8_t *gm_identity, uint16_t *time_base, struct *gptp_scaled_ns* *last_gm_ph_change, double *last_gm_freq_change)

Define callback that is called after a phase discontinuity has been sent by the grandmaster.

Param gm_identity
A pointer to first element of a ClockIdentity array. The size of the array is GPTP_CLOCK_ID_LEN.

Param time_base
A pointer to the value of timeBaseIndicator of the current grandmaster.

Param last_gm_ph_change
A pointer to the value of lastGmPhaseChange received from grandmaster.

Param last_gm_freq_change
A pointer to the value of lastGmFreqChange received from the grandmaster.

typedef void (*`gptp_port_cb_t`)(int port, struct *net_if* *iface, void *user_data)

Callback used while iterating over gPTP ports.

Param port
Port number

Param iface
Pointer to network interface

Param user_data
A valid pointer to user data or NULL

Functions

void `gptp_register_phase_dis_cb`(struct *gptp_phase_dis_cb* *phase_dis, *gptp_phase_dis_callback_t* cb)

Register a phase discontinuity callback.

Parameters

- `phase_dis` – Caller specified handler for the callback.
- `cb` – Callback to register.

void `gptp_unregister_phase_dis_cb`(struct *gptp_phase_dis_cb* *phase_dis)

Unregister a phase discontinuity callback.

Parameters

- `phase_dis` – Caller specified handler for the callback.

void `gptp_call_phase_dis_cb`(void)

Call a phase discontinuity callback function.

int `gptp_event_capture`(struct *net_ptp_time* *slave_time, bool *gm_present)

Get gPTP time.

Parameters

- `slave_time` – A pointer to structure where timestamp will be saved.

- **gm_present** – A pointer to a boolean where status of the presence of a grand master will be saved.

Returns
Error code. 0 if no error.

char *gptp_sprint_clock_id(const uint8_t *clk_id, char *output, size_t output_len)

Utility function to print clock id to a user supplied buffer.

Parameters

- **clk_id** – Clock id
- **output** – Output buffer
- **output_len** – Output buffer len

Returns
Pointer to output buffer

void gptp_foreach_port(*gptp_port_cb_t* cb, void *user_data)

Go through all the gPTP ports and call callback for each of them.

Parameters

- **cb** – User-supplied callback function to call
- **user_data** – User specified data

struct gptp_domain *gptp_get_domain(void)

Get gPTP domain.

This contains all the configuration / status of the gPTP domain.

Returns
Pointer to domain or NULL if not found.

void gptp_clk_src_time_invoke(struct *gptp_clk_src_time_invoke_params* *arg)

This interface is used by the ClockSource entity to provide time to the ClockMaster entity of a time-aware system.

Parameters

- **arg** – Current state and parameters of the ClockSource entity.

struct *gptp_hdr* *gptp_get_hdr(struct *net_pkt* *pkt)

Return pointer to gPTP packet header in network packet.

Parameters

- **pkt** – Network packet (received or sent)

Returns
Pointer to gPTP header.

struct gptp_scaled_ns

#include <gptp.h> Scaled Nanoseconds.

Public Members

int32_t high

High half.

int64_t **low**
Low half.

struct **gptp_uscaled_ns**
#include <*gptp.h*> UScaled Nanoseconds.

Public Members

uint32_t **high**
High half.

uint64_t **low**
Low half.

struct **gptp_port_identity**
#include <*gptp.h*> Port Identity.

Public Members

uint8_t **clk_id**[GPTP_CLOCK_ID_LEN]
Clock identity of the port.

uint16_t **port_number**
Number of the port.

struct **gptp_flags**
#include <*gptp.h*> gPTP message flags

Public Members

uint8_t **octets**[2]
Byte access.

uint16_t **all**
Whole field access.

struct **gptp_hdr**
#include <*gptp.h*> gPTP message header

Public Members

uint8_t **message_type**
Type of the message.

uint8_t **transport_specific**
: Transport specific, always 1.

uint8_t **ptp_version**
: Version of the PTP, always 2.

uint8_t **reserved0**
: Reserved field.

uint16_t **message_length**
: Total length of the message from the header to the last TLV.

uint8_t **domain_number**
: Domain number, always 0.

uint8_t **reserved1**
: Reserved field.

struct *gptp_flags* **flags**
: Message flags.

int64_t **correction_field**
: Correction Field.

  The content depends of the message type.

uint32_t **reserved2**
: Reserved field.

struct *gptp_port_identity* **port_id**
: Port Identity of the sender.

uint16_t **sequence_id**
: Sequence Id.

uint8_t **control**
: Control value.

  Sync: 0, Follow-up: 2, Others: 5.

int8_t **log_msg_interval**
: Message Interval in Log2 for Sync and Announce messages.

struct **gptp_phase_dis_cb**
: *#include <gptp.h>* Phase discontinuity callback structure.

  Stores the phase discontinuity callback information. Caller must make sure that the variable pointed by this is valid during the lifetime of registration. Typically this means that the variable cannot be allocated from stack.

Public Members

sys_snode_t **node**
Node information for the slist.

gptp_phase_dis_callback_t **cb**
Phase discontinuity callback.

struct **gptp_clk_src_time_invoke_params**
#include <*gptp.h*> ClockSourceTime.invoke function parameters.

Parameters passed by ClockSourceTime.invoke function.

Public Members

double **last_gm_freq_change**
Frequency change on the last Time Base Indicator Change.

struct *net_ptp_extended_time* **src_time**
The time this function is invoked.

struct *gptp_scaled_ns* **last_gm_phase_change**
Phase change on the last Time Base Indicator Change.

uint16_t **time_base_indicator**
Time Base - changed only if Phase or Frequency changes.

Network time representation in the network stack

API Reference

group **net_time**

Defines

NET_TIME_MAX
The largest positive time value that can be represented by net_time_t.

NET_TIME_MIN
The smallest negative time value that can be represented by net_time_t.

NET_TIME_SEC_MAX
The largest positive number of seconds that can be safely represented by net_time_t.

NET_TIME_SEC_MIN
The smallest negative number of seconds that can be safely represented by net_time_t.

Typedefs

typedef int64_t **net_time_t**

Any occurrence of net_time_t specifies a concept of nanosecond resolution scalar time span, future (positive) or past (negative) relative time or absolute timestamp referred to some local network uptime reference clock that does not wrap during uptime and is - in a certain, well-defined sense - common to all local network interfaces, sometimes even to remote interfaces on the same network.

This type is EXPERIMENTAL. Usage is currently restricted to representation of time within the network subsystem.

Timed network protocols (PTP, TDMA, ...) usually require several local or remote interfaces to share a common notion of elapsed time within well-defined tolerances. Network uptime therefore differs from time represented by a single hardware counter peripheral in that it will need to be represented in several distinct hardware peripherals with different frequencies, accuracy and precision. To co-operate, these hardware counters will have to be "syntonized" or "disciplined" (i.e. frequency and phase locked) with respect to a common local or remote network reference time signal. Be aware that while syntonized clocks share the same frequency and phase, they do not usually share the same epoch (zero-point).

This also explains why network time, if represented as a cycle value of some specific hardware counter, will never be "precise" but only can be "good

enough" with respect to the tolerances (resolution, drift, jitter) required by a given network protocol. All counter peripherals involved in a timed network protocol must comply with these tolerances.

Please use specific cycle/tick counter values rather than net_time_t whenever possible especially when referring to the kernel system clock or values of any single counter peripheral.

net_time_t cannot represent general clocks referred to an arbitrary epoch as it only covers roughly +/- ~290 years. It also cannot be used to represent time according to a more complex timescale (e.g. including leap seconds, time adjustments, complex calendars or time zones). In these cases you may use timespec (C11, POSIX.1-2001), timeval (POSIX.1-2001) or broken down time as in tm (C90). The advantage of net_time_t over these structured time representations is lower memory footprint, faster and simpler scalar arithmetic and easier conversion from/to low-level hardware counter values. Also net_time_t can be used in the network stack as well as in applications while POSIX concepts cannot. Converting net_time_t from/to structured time representations is possible in a limited way but - except for timespec - requires concepts that must be implemented by higher-level APIs. Utility functions converting from/to timespec will be provided as part of the net_time_t API as and when needed.

If you want to represent more coarse grained scalar time in network applications, use time_t (C99, POSIX.1-2001) which is specified to represent seconds or suseconds_t (POSIX.1-2001) for microsecond resolution. Kernel *k_ticks_t* and cycles (both specific to Zephyr) have an unspecified resolution but are useful to represent kernel timer values and implement high resolution spinning.

If you need even finer grained time resolution, you may want to look at (g)PTP concepts, see *net_ptp_extended_time*.

The reason why we don't use int64_t directly to represent scalar nanosecond resolution times in the network stack is that it has been shown in the past that fields using generic type will often not be used correctly (e.g. with the wrong resolution or to represent underspecified concepts of time with unclear syntonization semantics).

Any API that exposes or consumes net_time_t values SHALL ensure that it maintains the specified contract including all protocol specific tolerances and therefore clients can rely on common semantics of this type. This makes times coming from different hardware peripherals and even from different network nodes comparable within well-defined limits and therefore net_time_t is the ideal intermediate building block for timed network protocols.

Precision Time Protocol (PTP) time format

- *Overview*
- *API Reference*

Overview The PTP time struct can store time information in high precision format (nanoseconds). The extended timestamp format can store the time in fractional nanoseconds accuracy. The PTP time format is used in *generic Precision Time Protocol (gPTP)* implementation.

Related code samples

PTP
Enable PTP support and monitor messages and events via logging.

gPTP
Enable gPTP support and monitor functionality using net-shell.

API Reference

group `ptp_time`

Precision Time Protocol time specification.

Functions

static inline *net_time_t* `net_ptp_time_to_ns`(struct *net_ptp_time* *ts)

Convert a PTP timestamp to a nanosecond precision timestamp, both related to the local network reference clock.

Note

Only timestamps representing up to ~290 years can be converted to nanosecond timestamps. Larger timestamps will return the maximum representable nanosecond precision timestamp.

Parameters

- `ts` – the PTP timestamp

Returns
the corresponding nanosecond precision timestamp

static inline struct *net_ptp_time* **ns_to_net_ptp_time**(*net_time_t* nsec)

Convert a nanosecond precision timestamp to a PTP timestamp, both related to the local network reference clock.

Parameters

- **nsec** – a nanosecond precision timestamp

Returns

the corresponding PTP timestamp

struct **net_ptp_time**

#include <*ptp_time.h*> (Generalized) Precision Time Protocol Timestamp format.

This structure represents a timestamp according to the Precision Time Protocol standard ("PTP", IEEE 1588, section 5.3.3), the Generalized Precision Time Protocol standard ("gPTP", IEEE 802.1AS, section 6.4.3.4), or any other well-defined context in which precision structured timestamps are required on network messages in Zephyr.

Seconds are encoded as a 48 bits unsigned integer. Nanoseconds are encoded as a 32 bits unsigned integer.

In the context of (g)PTP, *timestamps* designate the time, relative to a local clock ("LocalClock") at which the message timestamp point passes a reference plane marking the boundary between the PTP Instance and the network medium (IEEE 1855, section 7.3.4.2; IEEE 802.1AS, section 8.4.3).

The exact definitions of the *message timestamp point* and *reference plane* depends on the network medium and use case.

For (g)PTP the media-specific message timestamp points and reference planes are defined in the standard. In non-PTP contexts specific to Zephyr, timestamps are measured relative to the same local clock but with a context-specific message timestamp point and reference plane, defined below per use case.

A *"LocalClock"* is a freerunning clock, embedded into a well-defined entity (e.g. a PTP Instance) and provides a common time to that entity relative to an arbitrary epoch (IEEE 1855, section 3.1.26, IEEE 802.1AS, section 3.16).

In Zephyr, the local clock is usually any instance of a kernel system clock driver, counter driver, RTC API driver or low-level counter/timer peripheral (e.g. an ethernet peripheral with hardware timestamp support or a radio timer) with sufficient precision for the context in which it is used.

See IEEE 802.1AS, Annex B for specific performance requirements regarding conformance of local clocks in the gPTP context. See IEEE 1588, Annex A, section A5.4 for general performance requirements regarding PTP local clocks. See IEEE 802.15.4-2020, section 15.7 for requirements in the context of ranging applications and ibid., section 6.7.6 for the relation between guard times and clock accuracy which again influence the precision required for subprotocols like CSL, TSCH, RIT, etc.

Applications that use timestamps across different subsystems or media must ensure that they understand the definition of the respective reference planes and interpret timestamps accordingly. Applications must further ensure that timestamps are either all referenced to the same local clock or convert between clocks based on sufficiently precise conversion algorithms.

Timestamps may be measured on ingress (RX timestamps) or egress (TX timestamps) of network messages. Timestamps can also be used to schedule a network message to a well-defined point in time in the future at which it is to be sent over the medium (timed TX). A future timestamp and a duration, both referenced to the local clock, may be given to specify a time window at which a network device should expect incoming messages (RX window).

In Zephyr this timestamp structure is currently used in the following contexts:

- gPTP for Full Duplex Point-to-Point IEEE 802.3 links (IEEE 802.1AS, section 11): the reference plane and message timestamp points are as defined in the standard.
- IEEE 802.15.4 timed TX and RX: Timestamps designate the point in time at which the end of the last symbol of the start-of-frame delimiter (SFD) (or equivalently, the start of the first symbol of the PHY header) is at the local antenna. The standard also refers to this as the "RMARKER" (IEEE 802.15.4-2020, section 6.9.1) or "symbol boundary" (ibid., section 6.5.2), depending on the context. In the context of beacon timestamps, the difference between the timestamp measurement plane and the reference plane is defined by the MAC PIB attribute "macSyncSymbolOffset", ibid., section 8.4.3.1, table 8-94.

If further use cases are added to Zephyr using this timestamp structure, their clock performance requirements, message timestamp points and reference plane definition SHALL be added to the above list.

Public Members

uint64_t **second**

Second value.

union net_ptp_time

Seconds encoded on 48 bits.

uint32_t **nanosecond**

Nanoseconds.

struct **net_ptp_extended_time**

#include <ptp_time.h> Generalized Precision Time Protocol Extended Timestamp format.

This structure represents an extended timestamp according to the Generalized Precision Time Protocol standard (IEEE 802.1AS), see section 6.4.3.5.

Seconds are encoded as 48 bits unsigned integer. Fractional nanoseconds are encoded as 48 bits, their unit is 2*(-16) ns.

A precise definition of PTP timestamps and their uses in Zephyr is given in the description of *net_ptp_time*.

Public Members

uint64_t **second**

Second value.

union net_ptp_extended_time

Seconds encoded on 48 bits.

uint64_t **fract_nsecond**

Fractional nanoseconds value.

union net_ptp_extended_time
: Fractional nanoseconds on 48 bits.

zperf: Network Traffic Generator

- *Overview*
- *Sample Usage*

Overview zperf is a shell utility which allows to generate network traffic in Zephyr. The tool may be used to evaluate network bandwidth.

zperf is compatible with iPerf_2.0.5. Note that in newer iPerf versions, an error message like this is printed and the server reported statistics are missing.

```
LAST PACKET NOT RECEIVED!!!
```

zperf can be enabled in any application, a dedicated sample is also present in Zephyr. See zperf sample application for details.

Sample Usage If Zephyr acts as a client, iPerf must be executed in server mode. For example, the following command line must be used for UDP testing:

```
$ iperf -s -l 1K -u -V -B 2001:db8::2
```

For TCP testing, the command line would look like this:

```
$ iperf -s -l 1K -V -B 2001:db8::2
```

In the Zephyr console, zperf can be executed as follows:

```
zperf udp upload 2001:db8::2 5001 10 1K 1M
```

For TCP the zperf command would look like this:

```
zperf tcp upload 2001:db8::2 5001 10 1K 1M
```

If the IP addresses of Zephyr and the host machine are specified in the config file, zperf can be started as follows:

```
zperf udp upload2 v6 10 1K 1M
```

or like this if you want to test TCP:

```
zperf tcp upload2 v6 10 1K 1M
```

If Zephyr is acting as a server, set the download mode as follows for UDP:

```
zperf udp download 5001
```

or like this for TCP:

```
zperf tcp download 5001
```

and in the host side, iPerf must be executed with the following command line if you are testing UDP:

```
$ iperf -l 1K -u -V -c 2001:db8::1 -p 5001
```

and this if you are testing TCP:

```
$ iperf -l 1K -V -c 2001:db8::1 -p 5001
```

iPerf output can be limited by using the -b option if Zephyr is not able to receive all the packets in orderly manner.

6.3.7 Connection Manager

Overview

Connection Manager is a collection of optional Zephyr features that aim to allow applications to monitor and control connectivity (access to IP-capable networks) with minimal concern for the specifics of underlying network technologies.

Using Connection Manager, applications can use a single abstract API to control network association and monitor Internet access, and avoid excessive use of technology-specific boilerplate.

This allows an application to potentially support several very different connectivity technologies (for example, Wi-Fi and LTE) with a single codebase.

Applications can also use Connection Manager to generically manage and use multiple connectivity technologies simultaneously.

Structure Connection Manager is split into the following two subsystems:

- *Connectivity monitoring* (header file `include/zephyr/net/conn_mgr_monitoring.h`) monitors all available *Zephyr network interfaces (ifaces)* and triggers *network management* events indicating when IP connectivity is gained or lost.
- *Connectivity control* (header file `include/zephyr/net/conn_mgr_connectivity.h`) provides an abstract API for controlling iface network association.

Connectivity monitoring

Connectivity monitoring tracks all available ifaces (whether or not they support *Connectivity control*) as they transition through various *operational states* and acquire or lose assigned IP addresses.

Each available iface is considered ready if it meets the following criteria:

- The iface is admin-up
  - This means the iface has been instructed to become operational-up (ready for use). This is done by a call to *net_if_up()*.
- The iface is oper-up
  - This means the interface is completely ready for use; It is online, and if applicable, has associated with a network.
  - See *Network interface state management* for details.
- The iface has at least one assigned IP address
  - Both IPv4 and IPv6 addresses are acceptable. This condition is met as soon as one or both of these is assigned.

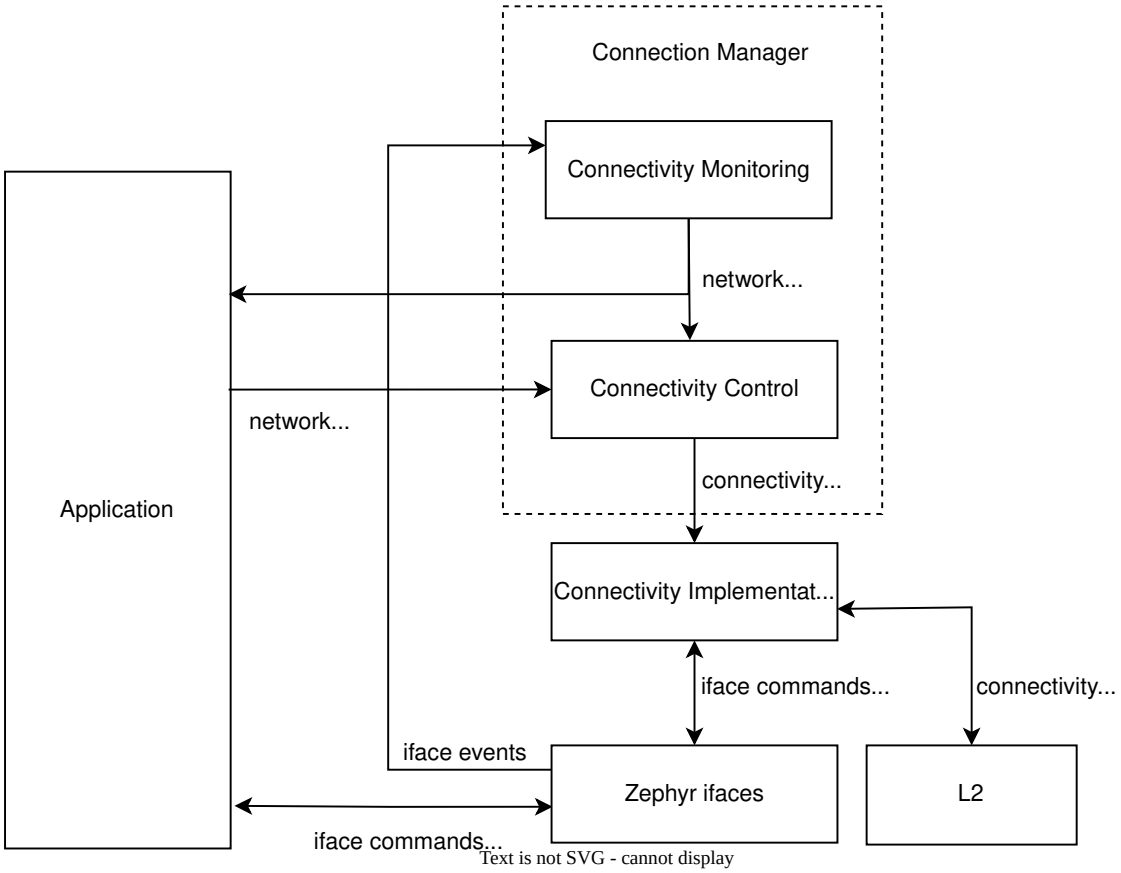


Fig. 18: A simplified view of how Connection Manager integrates with Zephyr and the application.

See *here* for a more detailed version.

  - See *Network Interface* for details on iface IP assignment.
- The iface has not been ignored
  - Ignored ifaces are always treated as unready.
  - See *Ignoring ifaces* for more details.

> **Note**
>
> Typically, iface state and IP assignment are updated either by the iface's *L2 implementation* or bound *connectivity implementation*.
>
> See *Implement iface state reporting* for details.

A ready iface ceases to be ready the moment any of the above conditions is lost.

When at least one iface is ready, the *NET_EVENT_L4_CONNECTED network management* event is triggered, and IP connectivity is said to be ready.

Afterwards, ifaces can become ready or unready without firing additional events, so long as there always remains at least one ready iface.

When there are no longer any ready ifaces left, the *NET_EVENT_L4_DISCONNECTED network management* event is triggered, and IP connectivity is said to be unready.

> **Note**
>
> Connection Manager also fires the following more specific `CONNECTED` / `DISCONNECTED` events:
>
> - *NET_EVENT_L4_IPV4_CONNECTED*
> - *NET_EVENT_L4_IPV4_DISCONNECTED*
> - *NET_EVENT_L4_IPV6_CONNECTED*
> - *NET_EVENT_L4_IPV6_DISCONNECTED*
>
> These are similar to *NET_EVENT_L4_CONNECTED* and *NET_EVENT_L4_DISCONNECTED*, but specifically track whether IPv4- and IPv6-capable ifaces are ready.

Usage Connectivity monitoring is enabled if the `CONFIG_NET_CONNECTION_MANAGER` Kconfig option is enabled.

To receive connectivity updates, create and register a listener for the *NET_EVENT_L4_CONNECTED* and *NET_EVENT_L4_DISCONNECTED network management* events:

```
/* Callback struct where the callback will be stored */
struct net_mgmt_event_callback l4_callback;

/* Callback handler */
static void l4_event_handler(struct net_mgmt_event_callback *cb,
                             uint32_t event, struct net_if *iface)
{
        if (event == NET_EVENT_L4_CONNECTED) {
                LOG_INF("Network connectivity gained!");
        } else if (event == NET_EVENT_L4_DISCONNECTED) {
                LOG_INF("Network connectivity lost!");
        }

        /* Otherwise, it's some other event type we didn't register for. */
}
```

(continues on next page)

(continued from previous page)

```
/* Call this before Connection Manager monitoring initializes */
static void my_application_setup(void)
{
        /* Configure the callback struct to respond to (at least) the L4_CONNECTED
         * and L4_DISCONNECTED events.
         *
         *
         * Note that the callback may also be triggered for events other than those␣
↪specified here!
         * (See the net_mgmt documentation)
         */
        net_mgmt_init_event_callback(
                &l4_callback, l4_event_handler,
                NET_EVENT_L4_CONNECTED | NET_EVENT_L4_DISCONNECTED
        );

        /* Register the callback */
        net_mgmt_add_event_callback(&l4_callback);
}
```

See *Listening to network events* for more details on listening for net_mgmt events.

Note

To avoid missing initial connectivity events, you should register your listener(s) before Connection Manager monitoring initializes. See *Avoiding missed notifications* for strategies to ensure this.

Avoiding missed notifications Connectivity monitoring may trigger events immediately upon initialization.

If your application registers its event listeners after connectivity monitoring initializes, it is possible to miss this first wave of events, and not be informed the first time network connectivity is gained.

If this is a concern, your application should *register its event listeners* before connectivity monitoring initializes.

Connectivity monitoring initializes using the `SYS_INIT` `APPLICATION` initialization priority specified by the `CONFIG_NET_CONNECTION_MANAGER_MONITOR_PRIORITY` Kconfig option.

You can register your callbacks before this initialization by using `SYS_INIT` with an earlier initialization priority than this value, for instance priority 0:

```
static int my_application_setup(void)
{
        /* Register callbacks here */
        return 0;
}

SYS_INIT(my_application_setup, APPLICATION, 0);
```

If this is not feasible, you can instead request that connectivity monitoring resend the latest connectivity events at any time by calling *conn_mgr_mon_resend_status()*:

```
static void my_late_application_setup(void)
{
  /* Register callbacks here */
```

(continues on next page)

(continued from previous page)

```
  /* Once done, request that events be re-triggered */
  conn_mgr_mon_resend_status();
}
```

Ignoring ifaces Applications can request that ifaces be ignored by Connection Manager by calling *conn_mgr_ignore_iface()* with the iface to be ignored.

Alternatively, an entire *L2 implementation* can be ignored by calling *conn_mgr_ignore_l2()*.

This has the effect of individually ignoring all the ifaces using that *L2 implementation*.

While ignored, the iface is treated by Connection Manager as though it were unready for network traffic, no matter its actual state.

This may be useful, for instance, if your application has configured one or more ifaces that cannot (or for whatever reason should not) be used to contact the wider Internet.

Bulk convenience functions optionally skip ignored ifaces.

See *conn_mgr_ignore_iface()* and *conn_mgr_watch_iface()* for more details.

Connectivity monitoring API Include header file `include/zephyr/net/conn_mgr_monitoring.h` to access these.

group **conn_mgr**

Connection Manager API.

Functions

void **conn_mgr_mon_resend_status**(void)

Resend either NET_L4_CONNECTED or NET_L4_DISCONNECTED depending on whether connectivity is currently available.

void **conn_mgr_ignore_iface**(struct *net_if* *iface)

Mark an iface to be ignored by conn_mgr.

Ignoring an iface forces conn_mgr to consider it unready/disconnected.

This means that events related to the iface connecting/disconnecting will not be fired, and if the iface was connected before being ignored, events will be fired as though it disconnected at that moment.

Parameters

- **iface** – iface to be ignored.

void **conn_mgr_watch_iface**(struct *net_if* *iface)

Watch (stop ignoring) an iface.

conn_mgr will no longer be forced to consider the iface unreadly/disconnected.

Events related to the iface connecting/disconnecting will no longer be blocked, and if the iface was connected before being watched, events will be fired as though it connected in that moment.

All ifaces default to watched at boot.

Parameters

- **iface** – iface to no longer ignore.

bool **conn_mgr_is_iface_ignored**(struct *net_if* *iface)

Check whether the provided iface is currently ignored.

Parameters

- **iface** – The iface to check.

Return values

- **true** – if the iface is being ignored by conn_mgr.
- **false** – if the iface is being watched by conn_mgr.

void **conn_mgr_ignore_l2**(const struct *net_l2* *l2)

Mark an L2 to be ignored by conn_mgr.

This is a wrapper for conn_mgr_ignore_iface that ignores all ifaces that use the L2.

Parameters

- **l2** – L2 to be ignored.

void **conn_mgr_watch_l2**(const struct *net_l2* *l2)

Watch (stop ignoring) an L2.

This is a wrapper for conn_mgr_watch_iface that watches all ifaces that use the L2.

Parameters

- **l2** – L2 to watch.

Connectivity control

Many network interfaces require a network association procedure to be completed before being usable.

For such ifaces, connectivity control can provide a generic API to request network association (*conn_mgr_if_connect()*) and disassociation (*conn_mgr_if_disconnect()*). Network interfaces implement support for this API by *binding themselves to a connectivity implementation*.

Using this API, applications can associate with networks with minimal technology-specific boilerplate.

Connectivity control also provides the following additional features:

- Standardized *persistence and timeout* behaviors during association.
- *Bulk functions* for controlling the admin state and network association of all available ifaces simultaneously.
- Optional *convenience automations* for common connectivity actions.

Basic operation The following sections outline the basic operation of Connection Manager's connectivity control.

Binding Before an iface can be commanded to associate or disassociate using Connection Manager, it must first be bound to a *connectivity implementation*. Binding is performed by the provider of the iface, not by the application (see *Binding an iface to an implementation*), and can be thought of as an extension of the iface declaration.

Once an iface is bound, all connectivity commands passed to it (such as *conn_mgr_if_connect()* or *conn_mgr_if_disconnect()*) will be routed to the corresponding implementation function in the connectivity implementation.

Note

To avoid inconsistent behavior, all connectivity implementations must adhere to the *implementation guidelines*.

Connecting Once a bound iface is admin-up (see *Network interface state management*), `conn_mgr_if_connect()` can be called to cause it to associate with a network.

If association succeeds, the connectivity implementation will mark the iface as operational-up (see *Network interface state management*).

If association fails unrecoverably, the *fatal error event* will be triggered.

You can configure an optional *timeout* for this process.

Note

The `conn_mgr_if_connect()` function is intentionally minimalistic, and does not take any kind of configuration. Each connectivity implementation should provide a way to pre-configure or automatically configure any required association settings or credentials. See *Allow connectivity pre-configuration* for details.

Connection loss If connectivity is lost due to external factors, the connectivity implementation will mark the iface as operational-down.

Depending on whether *persistence* is set, the iface may then attempt to reconnect.

Manual disconnection The application can also request that connectivity be intentionally abandoned by calling `conn_mgr_if_disconnect()`.

In this case, the connectivity implementation will disassociate the iface from its network and mark the iface as operational-down (see *Network interface state management*). A new connection attempt will not be initiated, regardless of whether persistence is enabled.

Timeouts and Persistence Connection Manager requires that all connectivity implementations support the following standard key features:

- *Connection timeouts*
- *Connection persistence*

These features describe how ifaces should behave during connect and disconnect events. You can individually set them for each iface.

Note

It is left to connectivity implementations to successfully and accurately implement these two features as described below. See *Implementing timeouts and persistence* for more details from the connectivity implementation perspective.

Connection Timeouts When `conn_mgr_if_connect()` is called on an iface, a connection attempt begins.

The connection attempt continues indefinitely until it succeeds, unless a timeout has been specified for the iface (using *conn_mgr_if_set_timeout()*).

In that case, the connection attempt will be abandoned if the timeout elapses before it succeeds. If this happens, the *timeout event* is raised.

Connection Persistence Each iface also has a connection persistence setting that you can enable or disable by setting the *CONN_MGR_IF_PERSISTENT* flag with *conn_mgr_binding_set_flag()*.

This setting specifies how the iface should handle unintentional connection loss.

If persistence is enabled, any unintentional connection loss will initiate a new connection attempt, with a new timeout if applicable.

Otherwise, the iface will not attempt to reconnect.

Note

Persistence not does affect connection attempt behavior. Only the timeout setting affects this.

For instance, if a connection attempt on an iface times out, the iface will not attempt to reconnect, even if it is persistent.

Conversely, if there is not a specified timeout, the iface will try to connect forever until it succeeds, even if it is not persistent.

See *Persistence during connection attempts* for the equivalent implementation guideline.

Control events Connectivity control triggers *network management* events to inform the application of important state changes.

See *Trigger connectivity control events* for the corresponding connectivity implementation guideline.

Fatal Error The *NET_EVENT_CONN_IF_FATAL_ERROR* event is raised when an iface encounters an error from which it cannot recover (meaning any subsequent attempts to associate are guaranteed to fail, and all such attempts should be abandoned).

Handlers of this event will be passed a pointer to the iface for which the fatal error occurred. Individual connectivity implementations may also pass an application-specific data pointer.

Timeout The *NET_EVENT_CONN_IF_TIMEOUT* event is raised when an *iface association* attempt *times out*.

Handlers of this event will be passed a pointer to the iface that timed out attempting to associate.

Listening for control events You can listen for control events as follows:

```
/* Declare a net_mgmt callback struct to store the callback */
struct net_mgmt_event_callback my_conn_evt_callback;

/* Declare a handler to receive control events */
static void my_conn_evt_handler(struct net_mgmt_event_callback *cb,
                                uint32_t event, struct net_if *iface)
{
        if (event == NET_EVENT_CONN_IF_TIMEOUT) {
                /* Timeout occurred, handle it */
```

(continues on next page)

(continued from previous page)

```
        } else if (event == NET_EVENT_CONN_IF_FATAL_ERROR) {
                /* Fatal error occurred, handle it */
        }

        /* Otherwise, it's some other event type we didn't register for. */
}

int main()
{
        /* Configure the callback struct to respond to (at least) the CONN_IF_TIMEOUT
         * and CONN_IF_FATAL_ERROR events.
         *
         * Note that the callback may also be triggered for events other than those␣
↪specified here!
         * (See the net_mgmt documentation)
         */

        net_mgmt_init_event_callback(
                &conn_mgr_conn_callback, conn_mgr_conn_handler,
                    NET_EVENT_CONN_IF_TIMEOUT | NET_EVENT_CONN_IF_FATAL_ERROR
        );

        /* Register the callback */
        net_mgmt_add_event_callback(&conn_mgr_conn_callback);
        return 0;
}
```

See *Listening to network events* for more details on listening for net_mgmt events.

Automated behaviors There are a few actions related to connectivity that are (by default at least) performed automatically for the user.

Automatic admin-up

In Zephyr, ifaces are automatically taken admin-up (see *Network interface state management* for details on iface states) during initialization.

Applications can disable this behavior by setting the *NET_IF_NO_AUTO_START* interface flag with *net_if_flag_set()*.

Automatic connect

By default, Connection Manager will automatically connect any *bound* iface that becomes admin-up.

Applications can disable this by setting the *CONN_MGR_IF_NO_AUTO_CONNECT* connectivity flag with *conn_mgr_if_set_flag()*.

Automatic admin-down

By default, Connection Manager will automatically take any bound iface admin-down if it has given up on associating.

Applications can disable this for all ifaces by disabling the CONFIG_NET_CONNECTION_MANAGER_AUTO_IF_DOWN Kconfig option, or for individual ifaces by setting the *CONN_MGR_IF_NO_AUTO_DOWN* connectivity flag with *conn_mgr_if_set_flag()*.

Connectivity control API Include header file include/zephyr/net/conn_mgr_connectivity.h to access these.

group conn_mgr_connectivity

Connection Manager Connectivity API.

Since
3.4

Version
0.1.0

Defines

NET_EVENT_CONN_IF_TIMEOUT

net_mgmt event raised when a connection attempt times out

NET_EVENT_CONN_IF_FATAL_ERROR

net_mgmt event raised when a non-recoverable connectivity error occurs on an iface

CONN_MGR_IF_NO_TIMEOUT

Value to use with *conn_mgr_if_set_timeout* and *conn_mgr_conn_binding::timeout* to indicate no timeout.

Enums

enum conn_mgr_if_flag

Per-iface connectivity flags.

Values:

enumerator CONN_MGR_IF_PERSISTENT

Persistent.

When set, indicates that the connectivity implementation bound to this iface should attempt to persist connectivity by automatically reconnecting after connection loss.

enumerator CONN_MGR_IF_NO_AUTO_CONNECT

No auto-connect.

When set, conn_mgr will not automatically attempt to connect this iface when it reaches admin-up.

enumerator CONN_MGR_IF_NO_AUTO_DOWN

No auto-down.

When set, conn_mgr will not automatically take the iface admin-down when it stops trying to connect, even if CONFIG_NET_CONNECTION_MANAGER_AUTO_IF_DOWN is enabled.

Functions

int **conn_mgr_if_connect**(struct *net_if* *iface)

Connect interface.

If the provided iface has been bound to a connectivity implementation, initiate network connect/association.

Automatically takes the iface admin-up (by calling *net_if_up*) if it isn't already.

Non-Blocking.

Parameters

- **iface** – Pointer to network interface

Return values

- **0** – on success.
- **-ESHUTDOWN** – if the iface is not admin-up.
- **-ENOTSUP** – if the iface does not have a connectivity implementation.
- **implementation-specific** – status code otherwise.

int **conn_mgr_if_disconnect**(struct *net_if* *iface)

Disconnect interface.

If the provided iface has been bound to a connectivity implementation, disconnect/disassociate it from the network, and cancel any pending attempts to connect/associate.

Does nothing if the iface is currently admin-down.

Parameters

- **iface** – Pointer to network interface

Return values

- **0** – on success.
- **-ENOTSUP** – if the iface does not have a connectivity implementation.
- **implementation-specific** – status code otherwise.

bool **conn_mgr_if_is_bound**(struct *net_if* *iface)

Check whether the provided network interface supports connectivity / has been bound to a connectivity implementation.

Parameters

- **iface** – Pointer to the iface to check.

Return values

- **true** – if connectivity is supported (a connectivity implementation has been bound).
- **false** – otherwise.

int **conn_mgr_if_set_opt**(struct *net_if* *iface, int optname, const void *optval, size_t optlen)

Set implementation-specific connectivity options.

If the provided iface has been bound to a connectivity implementation that supports it, implementation-specific connectivity options related to the iface.

Parameters

- **iface** – Pointer to the network interface.
- **optname** – Integer value representing the option to set. The meaning of values is up to the *conn_mgr_conn_api* implementation. Some settings may affect multiple ifaces.
- **optval** – Pointer to the value to be assigned to the option.
- **optlen** – Length (in bytes) of the value to be assigned to the option.

Return values

- **0** – if successful.
- **-ENOTSUP** – if conn_mgr_if_set_opt not implemented by the iface.
- **-ENOBUFS** – if optlen is too long.
- **-EINVAL** – if NULL optval pointer provided.
- **-ENOPROTOOPT** – if the optname is not recognized.
- **implementation-specific** – error code otherwise.

int **conn_mgr_if_get_opt**(struct *net_if* *iface, int optname, void *optval, size_t *optlen)

Get implementation-specific connectivity options.

If the provided iface has been bound to a connectivity implementation that supports it, retrieves implementation-specific connectivity options related to the iface.

optlen will always be set to the total number of bytes written, regardless of whether an error is returned, even if zero bytes were written.

Parameters

- **iface** – Pointer to the network interface.
- **optname** – Integer value representing the option to set. The meaning of values is up to the *conn_mgr_conn_api* implementation. Some settings may be shared by multiple ifaces.
- **optval** – Pointer to where the retrieved value should be stored.
- **optlen** – Pointer to length (in bytes) of the destination buffer available for storing the retrieved value. If the available space is less than what is needed, -ENOBUFS is returned. If the available space is invalid, -EINVAL is returned.

Return values

- **0** – if successful.
- **-ENOTSUP** – if conn_mgr_if_get_opt is not implemented by the iface.
- **-ENOBUFS** – if retrieval buffer is too small.
- **-EINVAL** – if invalid retrieval buffer length is provided, or if NULL optval or optlen pointer provided.
- **-ENOPROTOOPT** – if the optname is not recognized.

- `implementation-specific` – error code otherwise.

bool `conn_mgr_if_get_flag`(struct *net_if* *iface, enum *conn_mgr_if_flag* flag)

Check the value of connectivity flags.

If the provided iface is bound to a connectivity implementation, retrieves the value of the specified connectivity flag associated with that iface.

Parameters

- `iface` – - Pointer to the network interface to check.
- `flag` – - The flag to check.

Returns

True if the flag is set, otherwise False. Also returns False if the provided iface is not bound to a connectivity implementation, or the requested flag doesn't exist.

int `conn_mgr_if_set_flag`(struct *net_if* *iface, enum *conn_mgr_if_flag* flag, bool value)

Set the value of a connectivity flags.

If the provided iface is bound to a connectivity implementation, sets the value of the specified connectivity flag associated with that iface.

Parameters

- `iface` – - Pointer to the network interface to modify.
- `flag` – - The flag to set.
- `value` – - Whether the flag should be enabled or disabled.

Return values

- `0` – on success.
- `-EINVAL` – if the flag does not exist.
- `-ENOTSUP` – if the provided iface is not bound to a connectivity implementation.

int `conn_mgr_if_get_timeout`(struct *net_if* *iface)

Get the connectivity timeout for an iface.

If the provided iface is bound to a connectivity implementation, retrieves the timeout setting in seconds for it.

Parameters

- `iface` – - Pointer to the iface to check.

Returns

int - The connectivity timeout value (in seconds) if it could be retrieved, otherwise CONN_MGR_IF_NO_TIMEOUT.

int `conn_mgr_if_set_timeout`(struct *net_if* *iface, int timeout)

Set the connectivity timeout for an iface.

If the provided iface is bound to a connectivity implementation, sets the timeout setting in seconds for it.

Parameters

- `iface` – - Pointer to the network interface to modify.
- `timeout` – - The timeout value to set (in seconds). Pass *CONN_MGR_IF_NO_TIMEOUT* to disable the timeout.

Return values

- `0` – on success.
- `-ENOTSUP` – if the provided iface is not bound to a connectivity implementation.

Bulk API Connectivity control provides several bulk functions allowing all ifaces to be controlled at once.

You can restrict these functions to operate only on non-*ignored* ifaces if desired.

Include header file `include/zephyr/net/conn_mgr_connectivity.h` to access these.

group `conn_mgr_connectivity_bulk`

Connection Manager Bulk API.

Functions

int `conn_mgr_all_if_up`(bool skip_ignored)

Convenience function that takes all available ifaces into the admin-up state.

Essentially a wrapper for *net_if_up*.

Parameters

- `skip_ignored` – - If true, only affect ifaces that aren't ignored by conn_mgr. Otherwise, affect all ifaces.

Returns

0 if all net_if_up calls returned 0, otherwise the first nonzero value returned by a net_if_up call.

int `conn_mgr_all_if_down`(bool skip_ignored)

Convenience function that takes all available ifaces into the admin-down state.

Essentially a wrapper for *net_if_down*.

Parameters

- `skip_ignored` – - If true, only affect ifaces that aren't ignored by conn_mgr. Otherwise, affect all ifaces.

Returns

0 if all net_if_down calls returned 0, otherwise the first nonzero value returned by a net_if_down call.

int `conn_mgr_all_if_connect`(bool skip_ignored)

Convenience function that takes all available ifaces into the admin-up state, and connects those that support connectivity.

Essentially a wrapper for *net_if_up* and *conn_mgr_if_connect*.

Parameters

- `skip_ignored` – - If true, only affect ifaces that aren't ignored by conn_mgr. Otherwise, affect all ifaces.

Returns

0 if all net_if_up and conn_mgr_if_connect calls returned 0, otherwise the first nonzero value returned by either net_if_up or conn_mgr_if_connect.

int **conn_mgr_all_if_disconnect**(bool skip_ignored)

Convenience function that disconnects all available ifaces that support connectivity without putting them into admin-down state (unless auto-down is enabled for the iface).

Essentially a wrapper for *net_if_down*.

Parameters

- **skip_ignored** – - If true, only affect ifaces that aren't ignored by conn_mgr. Otherwise, affect all ifaces.

Returns

0 if all net_if_up and conn_mgr_if_connect calls returned 0, otherwise the first nonzero value returned by either net_if_up or conn_mgr_if_connect.

Connectivity Implementations

Overview Connectivity implementations are technology-specific modules that allow specific Zephyr ifaces to support *Connectivity Control*. They are responsible for translating generic *connectivity control API* calls into hardware-specific operations. They are also responsible for implementing standardized *persistence and timeout* behaviors.

See the *implementation guidelines* for details on writing conformant connectivity implementations.

Architecture The *implementation API* allows connectivity implementations to be *defined* at build time using *CONN_MGR_CONN_DEFINE*.

This creates a static instance of the *conn_mgr_conn_impl* struct, which then stores a reference to the passed in *conn_mgr_conn_api* struct (which should be populated with implementation callbacks).

Once defined, you can reference implementations by name and bind them to any unbound iface using *CONN_MGR_BIND_CONN*. Make sure not to accidentally bind two connectivity implementations to a single iface.

Once the iface is bound, *connectivity control API* functions can be called on the iface, and they will be translated to the corresponding implementation functions in *conn_mgr_conn_api*.

Binding an iface does not directly modify its *iface struct*.

Instead, an instance of *conn_mgr_conn_binding* is created and appended an internal *iterable section*.

This binding structure will contain a reference to the bound iface, the connectivity implementation it is bound to, as well as a pointer to a per-iface *context pointer*.

This iterable section can then be iterated over to find out what (if any) connectivity implementation has been bound to a given iface. This search process is used by most of the functions in the *connectivity control API*. As such, these functions should be called sparingly, due to their relatively high search cost.

A single connectivity implementation may be bound to multiple ifaces. See *Do not instance implementations* for more details.

Context Pointer Since a single connectivity implementation may be shared by several Zephyr ifaces, each binding instantiates a context container (of *configurable type*) unique to that binding. Each binding is then instantiated with a reference to that container, which implementations can then use to access per-iface state information.

See also *Do not access bindings without locking them* and *Do not instance implementations*.

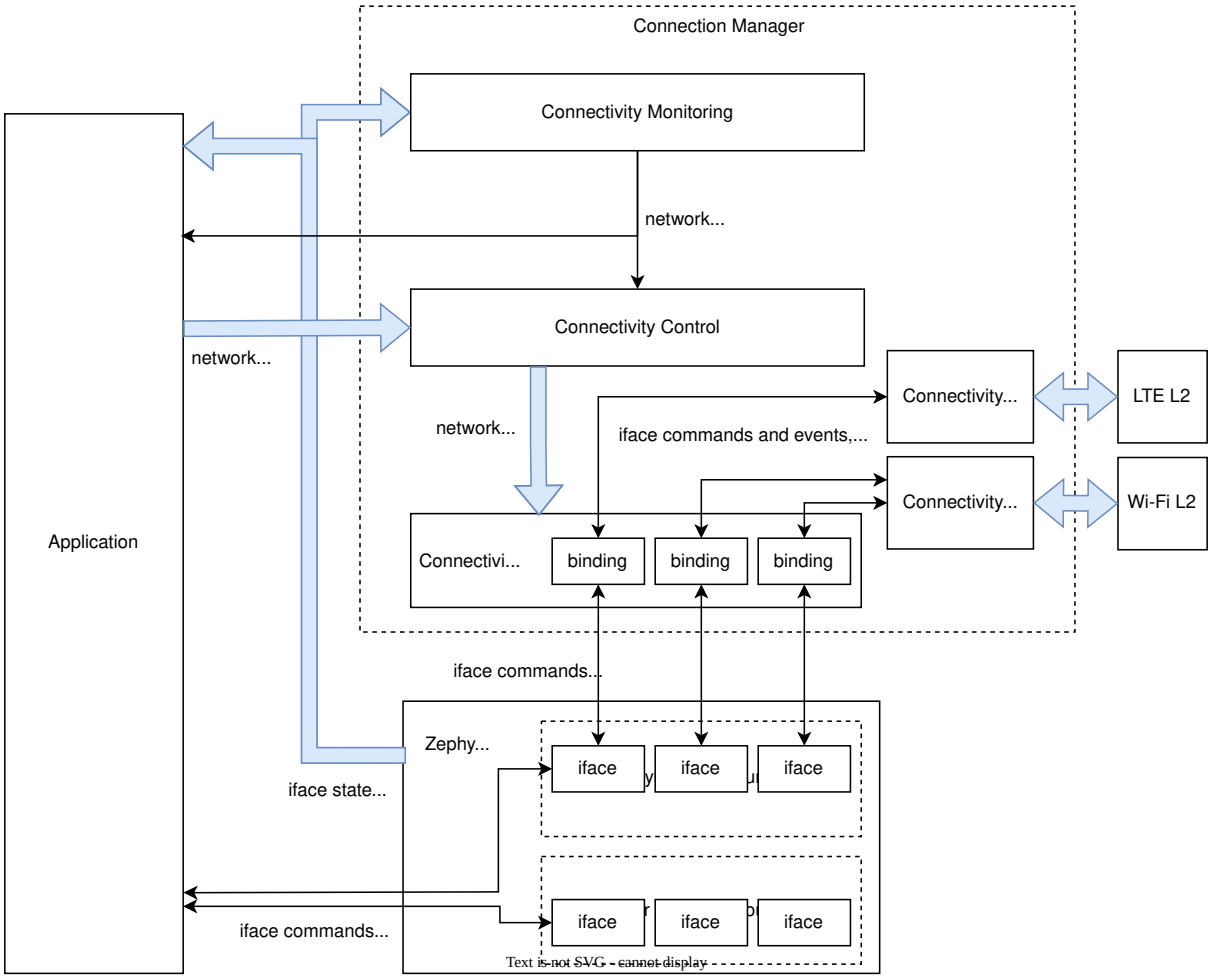


Fig. 19: A detailed view of how Connection Manager integrates with Zephyr and the application. See *here* for a simplified version.

Defining an implementation A connectivity implementation may be defined as follows:

```
/* Create the API implementation functions */
int my_connect_impl(struct conn_mgr_conn_binding *const binding) {
        /* Cause your underlying technology to associate */
}
int my_disconnect_impl(struct conn_mgr_conn_binding *const binding) {
        /* Cause your underlying technology to disassociate */
}
void my_init_impl(struct conn_mgr_conn_binding *const binding) {
        /* Perform any required initialization for your underlying technology */
}

/* Declare the API struct */
static struct conn_mgr_conn_api my_impl_api = {
        .connect = my_connect_impl,
        .disconnect = my_disconnect_impl,
        .init = my_init_impl,
        /* ... so on */
};

/* Define the implementation (named MY_CONNECTIVITY_IMPL) */
CONN_MGR_CONN_DEFINE(MY_CONNECTIVITY_IMPL, &my_impl_api);
```

Note

This does not work unless you also *declare the context pointer type*.

Declaring an implementation publicly Once defined, you can make a connectivity implementation available to other compilation units by declaring it (in a header file) as follows:

Listing 8: `my_connectivity_header.h`

```
CONN_MGR_CONN_DECLARE_PUBLIC(MY_CONNECTIVITY_IMPL);
```

The header file that contains this declaration must be included in any compilation units that need to reference the implementation.

Declaring a context type For *CONN_MGR_CONN_DEFINE* to work, you must declare a corresponding context pointer type. This is because all connectivity bindings contain a *Context Pointer* of their associated context pointer type.

If you are using *CONN_MGR_CONN_DECLARE_PUBLIC*, declare this type alongside the declaration:

Listing 9: `my_connectivity_impl.h`

```
#define MY_CONNECTIVITY_IMPL_CTX_TYPE struct my_context_type *
CONN_MGR_CONN_DECLARE_PUBLIC(MY_CONNECTIVITY_IMPL);
```

Then, make sure to include the header file before calling *CONN_MGR_CONN_DEFINE*:

Listing 10: `my_connectivity_impl.c`

```
#include "my_connectivity_impl.h"

CONN_MGR_CONN_DEFINE(MY_CONNECTIVITY_IMPL, &my_impl_api);
```

Otherwise, it is sufficient to simply declare the context pointer type immediately before the call to *CONN_MGR_CONN_DEFINE*:

```
#define MY_CONNECTIVITY_IMPL_CTX_TYPE struct my_context_type *
CONN_MGR_CONN_DEFINE(MY_CONNECTIVITY_IMPL, &my_impl_api);
```

Note

Naming is important. Your context pointer type declaration must use the same name as your implementation declaration, but with `_CTX_TYPE` appended.

In the previous example, the context type is named `MY_CONNECTIVITY_IMPL_CTX_TYPE`, because `MY_CONNECTIVITY_IMPL` was used as the connectivity implementation name.

If your connectivity implementation does not need a context pointer, simply declare the type as void:

```
#define MY_CONNECTIVITY_IMPL_CTX_TYPE void *
```

Binding an iface to an implementation A defined connectivity implementation may be bound to an iface by calling *CONN_MGR_BIND_CONN* anywhere after the iface's device definition:

```
/* Define an iface */
NET_DEVICE_INIT(my_iface,
        /* ... the specifics here don't matter ... */
);

/* Now bind MY_CONNECTIVITY_IMPL to that iface --
 * the name used should match with the above
 */
CONN_MGR_BIND_CONN(my_iface, MY_CONNECTIVITY_IMPL);
```

Connectivity implementation guidelines Rather than implement all features centrally, Connection Manager relies on each connectivity implementation to implement many behaviors and features individually.

This approach allows Connection Manager to remain lean, and allows each connectivity implementation to choose the most appropriate approach to these behaviors for itself. However, it relies on trust that all connectivity implementations will faithfully implement the features that have been delegated to them.

To maintain consistency between all connectivity implementations, observe the following guidelines when writing your own implementation:

Completely implement timeout and persistence All connectivity implementations must offer complete support for *timeout and persistence*, such that a user can disable or enable these features, regardless of the inherent behavior of the underlying technology. In other words, no matter how the underlying technology behaves, your implementation must make it appear to the end user to behave exactly as specified in the *Timeouts and Persistence* section.

See *Implementing timeouts and persistence* for a detailed technical discussion on implementing timeouts and persistence.

Conform to API specifications Each *implementation API function* you implement should behave as-described in the corresponding connectivity control API function.

For example, your implementation of *conn_mgr_conn_api.connect* should conform to the behavior described for *conn_mgr_if_connect()*.

Allow connectivity pre-configuration Connectivity implementations should provide means for applications to pre-configure all necessary connection parameters (for example, network SSID, or PSK, if applicable), before the call to *conn_mgr_if_connect()*. It should not be necessary to provide this information as part of, or following the *conn_mgr_if_connect()* call, although implementations *should await this information if it is not provided*.

Await valid connectivity configuration If network association fails because the application pre-configured invalid connection parameters, or did not configure connection parameters at all, this should be treated as a network failure.

In other words, the connectivity implementation should not give up on the connection attempt, even if valid connection parameters have not been configured.

Instead, the connectivity implementation should asynchronously wait for valid connection parameters to be configured, either indefinitely, or until the configured *connectivity timeout* elapses.

Implement iface state reporting All connectivity implementations must keep bound iface state up to date.

To be specific:

- Set the iface to dormant, carrier-down, or both during *binding init*.
  - See *Network interface state management* for details regarding iface carrier and dormant states.
- Update dormancy and carrier state so that the iface is non-dormant and carrier-up whenever (and only when) association is complete and connectivity is ready.
- Set the iface either to dormant or to carrier-down as soon as interruption of service is detected.
  - It is acceptable to gate this behind a small timeout (separate from the connection timeout) for network technologies where service is commonly intermittent.
- If the technology also handles IP assignment, ensure those IP addresses are *assigned to the iface*.

Note

iface state updates do not necessarily need to be performed directly by connectivity implementations.

For instance:

- IP assignment is not necessary if *DHCP* is used for the iface.
- The connectivity implementation does not need to update iface dormancy if the underlying *L2 implementation* already does so.

Do not use iface state as implementation state Zephyr ifaces may be accessed from other threads without respecting the binding mutex. As such, Zephyr iface state may change unpredictably during connectivity implementation callbacks.

Therefore, do not base implementation behaviors on iface state.

Keep iface state updated to reflect network availability, but do not read iface state for any purpose.

If you need to keep track of dormancy or IP assignment, use a separate state variable stored in the *context pointer*.

Remain non-interferent Connectivity implementations should not prevent applications from interacting directly with associated technology-specific APIs.

In other words, it should be possible for an application to directly use your underlying technology without breaking your connectivity implementation.

If exceptions to this are absolutely necessary, they should be constrained to specific API calls and should be documented.

Note

While connectivity implementations must not break, it is acceptable for implementations to have potentially unexpected behavior if applications attempt to directly control the association state.

For instance, if an application directly instructs an underlying technology to disassociate, it would be acceptable for the connectivity implementation to interpret this as an unexpected connection loss and immediately attempt to re-associate.

Remain non-blocking All connectivity implementation callbacks should be non-blocking.

For instance, calls to `conn_mgr_conn_api.connect` should initiate a connection process and return immediately.

One exception is `conn_mgr_conn_api.init`, whose implementations are permitted to block.

However, bear in mind that blocking during this callback will delay system init, so still consider offloading time-consuming tasks to a background thread.

Make API immediately ready Connectivity implementations must be ready to receive API calls immediately after `conn_mgr_conn_api.init`.

For instance, a call to `conn_mgr_conn_api.connect` must eventually lead to an association attempt, even if called immediately after `conn_mgr_conn_api.init`.

If the underlying technology cannot be made ready for connect commands immediately when `conn_mgr_conn_api.init` is called, calls to `conn_mgr_conn_api.connect` must be queued in a non-blocking fashion, and then executed later when ready.

Do not store state information outside the context pointer Connection Manager provides a context pointer to each binding.

Connectivity implementations should store all state information in this context pointer.

The only exception is connectivity implementations that are meant to be bound to only a single iface. Such implementations may use statically declared state instead.

See also *Do not instance implementations*.

Access ifaces only through binding structs Do not use statically declared ifaces or externally acquire references to ifaces.

For example, do not use `net_if_get_default()` under the assumption that the bound iface will be the default iface.

Instead, always use the `iface pointer` provided by the relevant `binding struct`. See also *Do not access bindings without locking them*.

Make implementations optional at compile-time Connectivity implementations should provide a Kconfig option to enable or disable the implementation without affecting bound iface availability.

In other words, it should be possible to configure builds that include Connectivity Manager, as well as the iface that would have been bound to the implementation, but not the implementation itself, nor its binding.

Do not instance implementations Do not declare a separate connectivity implementation for every iface you are going to bind to.

Instead, bind one global connectivity implementation to all of your ifaces, and use the context pointer to store state relevant to individual ifaces.

See also *Do not access bindings without locking them* and *Access ifaces only through binding structs*.

Do not access bindings without locking them Bindings may be accessed and modified at random by multiple threads, so modifying or reading from a binding without first *locking it* may lead to unpredictable behavior.

This applies to all descendents of the binding, including anything in the *context container*.

Make sure to *unlock* the binding when you are done accessing it.

Note

A possible exception to this rule is if the resource in question is inherently thread-safe.

However, be careful taking advantage of this exception. It may still be possible to create a race condition, for instance when accessing multiple thread-safe resources simultaneously.

Therefore, it is recommended to simply always lock the binding, whether or not the resource being accessed is inherently thread-safe.

Do not disable built-in features Do not attempt to prevent the use of built-in features (such as *Timeouts and Persistence* or *Automated behaviors*).

All connectivity implementations must fully support these features. Implementations must not attempt to force certain features to be always enabled or always disabled.

Trigger connectivity control events Connectivity control *network management* events are not triggered automatically by Connection Manager.

Connectivity implementations must trigger these events themselves.

Trigger `NET_EVENT_CONN_CMD_IF_TIMEOUT` when a connection *timeout* occurs. See *Timeout* for details.

Trigger *NET_EVENT_CONN_IF_FATAL_ERROR* when a fatal (non-recoverable) connection error occurs. See *Fatal Error* for details.

See *Network Management* for details on firing network management events.

Implementing timeouts and persistence First, see *Timeouts and Persistence* for a high-level description of the expected behavior of timeouts and persistence.

Connectivity implementations must fully conform to that description, regardless of the behavior of the underlying connectivity technology.

Sometimes this means writing extra logic in the connectivity implementation to fake certain behaviors. The following sections discuss various common edge-cases and nuances and how to handle them.

Inherently persistent technologies If the underlying technology automatically attempts to reconnect or retry connection after connection loss or failure, the connectivity implementation must manually cancel such attempts when they are in conflict with timeout or persistence settings.

For example:

- If the underlying technology automatically attempts to reconnect after losing connection, and persistence is disabled for the iface, the connectivity implementation should immediately cancel this reconnection attempt.
- If a connection attempt times out on an iface whose underlying technology does not have a built-in timeout, the connectivity implementation must simulate a timeout by cancelling the connection attempt manually.

Technologiess that give up on connection attempts If the underlying technology has no mechanism to retry connection attempts, or would give up on them before the user-configured timeout, or would not reconnect after connection loss, the connectivity implementation must manually re-request connection to counteract these deviances.

- If your underlying technology is not persistent, you must manually trigger reconnect attempts when persistence is enabled.
- If your underlying technology does not support a timeout, you must manually cancel connection attempts if the timeout is enabled.
- If your underlying technology forces a timeout, you must manually trigger a new connection attempts if that timeout is shorter than the Connection Manager timeout.

Technologies with association retry Many underlying technologies do not usually associate in a single attempt.

Instead, these underlying technologies may need to make multiple back-to-back association attempts in a row, usually with a small delay.

In these situations, the connectivity implementation should treat this series of back-to-back association sub-attempts as a single unified connection attempt.

For instance, after a sub-attempt failure, persistence being disabled should not prevent further sub-attempts, since they all count as one single overall connection attempt. See also *Persistence during connection attempts*.

At which point a series of failed sub-attempts should be considered a failure of the connection attempt as a whole is up to each implementation to decide.

If the connection attempt crosses this threshold, but the configured timeout has not yet elapsed, or there is no timeout, sub-attempts should continue.

Persistence during connection attempts Persistence should not affect any aspect of implementation behavior during a connection attempt. Persistence should only affect whether or not connection attempts are automatically triggered after a connection loss.

The configured timeout should fully determine whether connection retry should be performed.

Implementation API Include header file `include/zephyr/net/conn_mgr_connectivity_impl.h` to access these.

Only for use by connectivity implementations.

group `conn_mgr_connectivity_impl`

Connection Manager Connectivity Implementation API.

Defines

`CONN_MGR_CONN_DEFINE`(conn_id, conn_api)

Define a conn_mgr connectivity implementation that can be bound to network devices.

Parameters

- `conn_id` – The name of the new connectivity implementation
- `conn_api` – A pointer to a *conn_mgr_conn_api* struct

`CONN_MGR_CONN_DECLARE_PUBLIC`(conn_id)

Helper macro to make a conn_mgr connectivity implementation publicly available.

`CONN_MGR_BIND_CONN_INST`(dev_id, inst, conn_id)

Associate a connectivity implementation with an existing network device instance.

Parameters

- `dev_id` – Network device id.
- `inst` – Network device instance.
- `conn_id` – Name of the connectivity implementation to associate.

`CONN_MGR_BIND_CONN`(dev_id, conn_id)

Associate a connectivity implementation with an existing network device.

Parameters

- `dev_id` – Network device id.
- `conn_id` – Name of the connectivity implementation to associate.

Functions

static inline struct *conn_mgr_conn_binding* *`conn_mgr_if_get_binding`(struct *net_if* *iface)

Retrieves the conn_mgr binding struct for a provided iface if it exists.

Bindings for connectivity implementations with missing API structs are ignored.

For use only by connectivity implementations.

Parameters

- `iface` – - bound network interface to obtain the binding struct for.

Returns

struct conn_mgr_conn_binding* Pointer to the retrieved binding struct if it exists, NULL otherwise.

static inline void **conn_mgr_binding_lock**(struct *conn_mgr_conn_binding* *binding)

Lock the passed-in binding, making it safe to access.

Call this whenever accessing binding data, unless inside a *conn_mgr_conn_api* callback, where it is called automatically by conn_mgr.

Reentrant.

For use only by connectivity implementations.

Parameters

- **binding** – - Binding to lock

static inline void **conn_mgr_binding_unlock**(struct *conn_mgr_conn_binding* *binding)

Unlocks the passed-in binding.

Call this after any call to *conn_mgr_binding_lock* once done accessing binding data.

Reentrant.

For use only by connectivity implementations.

Parameters

- **binding** – - Binding to unlock

static inline void **conn_mgr_binding_set_flag**(struct *conn_mgr_conn_binding* *binding, enum *conn_mgr_if_flag* flag, bool value)

Set the value of the specified connectivity flag for the provided binding.

Can be used from any thread or callback without calling *conn_mgr_binding_lock*.

For use only by connectivity implementations

Parameters

- **binding** – The binding to check
- **flag** – The flag to check
- **value** – New value for the specified flag

static inline bool **conn_mgr_binding_get_flag**(struct *conn_mgr_conn_binding* *binding, enum *conn_mgr_if_flag* flag)

Check the value of the specified connectivity flag for the provided binding.

Can be used from any thread or callback without calling *conn_mgr_binding_lock*.

For use only by connectivity implementations

Parameters

- **binding** – The binding to check
- **flag** – The flag to check

Returns

bool The value of the specified flag

struct **conn_mgr_conn_api**

#include <*conn_mgr_connectivity_impl.h*> Connectivity Manager Connectivity API structure.

Used to provide generic access to network association parameters and procedures

Public Members

int (***connect**)(struct *conn_mgr_conn_binding* *const binding)

When called, the connectivity implementation should start attempting to establish connectivity (association with a network) for the bound iface pointed to by if_conn->iface.

Must be non-blocking.

Called by *conn_mgr_if_connect*.

int (***disconnect**)(struct *conn_mgr_conn_binding* *const binding)

When called, the connectivity implementation should disconnect (disassociate), or stop any in-progress attempts to associate to a network, the bound iface pointed to by if_conn->iface.

Must be non-blocking.

Called by *conn_mgr_if_disconnect*.

void (***init**)(struct *conn_mgr_conn_binding* *const binding)

Called once for each iface that has been bound to a connectivity implementation using this API.

Connectivity implementations should use this callback to perform any required per-bound-iface initialization.

Implementations may choose to gracefully handle invalid buffer lengths with partial writes, rather than raise errors, if deemed appropriate.

int (***set_opt**)(struct *conn_mgr_conn_binding* *const binding, int optname, const void *optval, size_t optlen)

Implementation callback for conn_mgr_if_set_opt.

Used to set implementation-specific connectivity settings.

Calls to conn_mgr_if_set_opt on an iface will result in calls to this callback with the *conn_mgr_conn_binding* struct bound to that iface.

It is up to the connectivity implementation to interpret optname. Options can be specific to the bound iface (pointed to by if_conn->iface), or can apply to the whole connectivity implementation.

See the description of conn_mgr_if_set_opt for more details. set_opt implementations should conform to that description.

Implementations may choose to gracefully handle invalid buffer lengths with partial reads, rather than raise errors, if deemed appropriate.

int (***get_opt**)(struct *conn_mgr_conn_binding* *const binding, int optname, void *optval, size_t *optlen)

Implementation callback for conn_mgr_if_get_opt.

Used to retrieve implementation-specific connectivity settings.

Calls to conn_mgr_if_get_opt on an iface will result in calls to this callback with the *conn_mgr_conn_binding* struct bound to that iface.

It is up to the connectivity implementation to interpret optname. Options can be specific to the bound iface (pointed to by if_conn->iface), or can apply to the whole connectivity implementation.

See the description of conn_mgr_if_get_opt for more details. get_opt implementations should conform to that description.

struct **conn_mgr_conn_impl**

#include <*conn_mgr_connectivity_impl.h*> Connectivity Implementation struct.

Declares a conn_mgr connectivity layer implementation with the provided API

Public Members

struct *conn_mgr_conn_api* ***api**

The connectivity API used by the implementation.

struct **conn_mgr_conn_binding**

#include <*conn_mgr_connectivity_impl.h*> Connectivity Manager network interface binding structure.

Binds a conn_mgr connectivity implementation to an iface / network device. Stores per-iface state for the connectivity implementation.

Generic connectivity state

uint32_t **flags**

Connectivity flags.

Public boolean state and configuration values supported by all bindings. See conn_mgr_if_flag for options.

int **timeout**

Timeout (seconds)

Indicates to the connectivity implementation how long it should attempt to establish connectivity for during a connection attempt before giving up.

The connectivity implementation should give up on establishing connectivity after this timeout, even if persistence is enabled.

Set to *CONN_MGR_IF_NO_TIMEOUT* to indicate that no timeout should be used.

Public Members

struct *net_if* ***iface**

The network interface the connectivity implementation is bound to.

const struct *conn_mgr_conn_impl* ***impl**

The connectivity implementation the network device is bound to.

void ***ctx**

Pointer to private, per-iface connectivity context.

6.4 LoRa and LoRaWAN

6.4.1 Overview

LoRa (abbrev. for Long Range) is a proprietary low-power wireless communication protocol developed by the Semtech Corporation.

LoRa acts as the physical layer (PHY) based on the chirp spread spectrum (CSS) modulation technique.

LoRaWAN (for Long Range Wide Area Network) defines a networking layer on top of the LoRa PHY.

Zephyr provides APIs for LoRa to send raw data packets directly over the wireless interface as well as APIs for LoRaWAN to connect the end device to the internet through a gateway.

The Zephyr implementation is based on Semtech's LoRaMac-node library, which is included as a Zephyr module.

The LoRaWAN specification is published by the LoRa Alliance.

6.4.2 Configuration Options

LoRa PHY

Related configuration options can be found under drivers/lora/Kconfig.

- CONFIG_LORA
- CONFIG_LORA_SHELL
- CONFIG_LORA_INIT_PRIORITY

LoRaWAN

Related configuration options can be found under subsys/lorawan/Kconfig.

- CONFIG_LORAWAN
- CONFIG_LORAWAN_SYSTEM_MAX_RX_ERROR
- CONFIG_LORAMAC_REGION_AS923
- CONFIG_LORAMAC_REGION_AU915
- CONFIG_LORAMAC_REGION_CN470
- CONFIG_LORAMAC_REGION_CN779
- CONFIG_LORAMAC_REGION_EU433
- CONFIG_LORAMAC_REGION_EU868
- CONFIG_LORAMAC_REGION_KR920
- CONFIG_LORAMAC_REGION_IN865
- CONFIG_LORAMAC_REGION_US915
- CONFIG_LORAMAC_REGION_RU864

6.4.3 API Reference

LoRa PHY

Related code samples

LoRa receive
Receive packets in both synchronous and asynchronous mode using the LoRa radio.

LoRa send
Transmit a preconfigured payload every second using the LoRa radio.

group `lora_api`

Since
2.2

Version
0.1.0

Enums

enum `lora_signal_bandwidth`
LoRa signal bandwidth.

Values:

enumerator `BW_125_KHZ` = 0

enumerator `BW_250_KHZ`

enumerator `BW_500_KHZ`

enum `lora_datarate`
LoRa data-rate.

Values:

enumerator `SF_6` = 6

enumerator `SF_7`

enumerator `SF_8`

enumerator `SF_9`

enumerator `SF_10`

enumerator `SF_11`

enumerator SF_12

enum lora_coding_rate

LoRa coding rate.

Values:

enumerator CR_4_5 = 1

enumerator CR_4_6 = 2

enumerator CR_4_7 = 3

enumerator CR_4_8 = 4

Functions

static inline int lora_config(const struct *device* *dev, struct *lora_modem_config* *config)

Configure the LoRa modem.

Parameters

- dev – LoRa device
- config – Data structure containing the intended configuration for the modem

Returns

0 on success, negative on error

static inline int lora_send(const struct *device* *dev, uint8_t *data, uint32_t data_len)

Send data over LoRa.

> **Note**
>
> This blocks until transmission is complete.

Parameters

- dev – LoRa device
- data – Data to be sent
- data_len – Length of the data to be sent

Returns

0 on success, negative on error

static inline int lora_send_async(const struct *device* *dev, uint8_t *data, uint32_t data_len, struct *k_poll_signal* *async)

Asynchronously send data over LoRa.

> **Note**
>
> This returns immediately after starting transmission, and locks the LoRa modem until the transmission completes.

Parameters

- `dev` – LoRa device
- `data` – Data to be sent
- `data_len` – Length of the data to be sent
- `async` – A pointer to a valid and ready to be signaled struct *k_poll_signal*. (Note: if NULL this function will not notify the end of the transmission).

Returns
0 on success, negative on error

static inline int `lora_recv`(const struct *device* *dev, uint8_t *data, uint8_t size, *k_timeout_t* timeout, int16_t *rssi, int8_t *snr)

Receive data over LoRa.

> **Note**
>
> This is a blocking call.

Parameters

- `dev` – LoRa device
- `data` – Buffer to hold received data
- `size` – Size of the buffer to hold the received data. Max size allowed is 255.
- `timeout` – Duration to wait for a packet.
- `rssi` – RSSI of received data
- `snr` – SNR of received data

Returns
Length of the data received on success, negative on error

static inline int `lora_recv_async`(const struct *device* *dev, lora_recv_cb cb)

Receive data asynchronously over LoRa.

Receive packets continuously under the configuration previously setup by *lora_config*.

Reception is cancelled by calling this function again with cb = NULL. This can be done within the callback handler.

Parameters

- `dev` – Modem to receive data on.
- `cb` – Callback to run on receiving data. If NULL, any pending asynchronous receptions will be cancelled.

Returns
0 when reception successfully setup, negative on error

static inline int `lora_test_cw`(const struct *device* *dev, uint32_t frequency, int8_t tx_power, uint16_t duration)

Transmit an unmodulated continuous wave at a given frequency.

Note

Only use this functionality in a test setup where the transmission does not interfere with other devices.

Parameters

- `dev` – LoRa device
- `frequency` – Output frequency (Hertz)
- `tx_power` – TX power (dBm)
- `duration` – Transmission duration in seconds.

Returns

0 on success, negative on error

struct `lora_modem_config`

#include <*lora.h*> Structure containing the configuration of a LoRa modem.

Public Members

uint32_t `frequency`

Frequency in Hz to use for transceiving.

enum *lora_signal_bandwidth* `bandwidth`

The bandwidth to use for transceiving.

enum *lora_datarate* `datarate`

The data-rate to use for transceiving.

enum *lora_coding_rate* `coding_rate`

The coding rate to use for transceiving.

uint16_t `preamble_len`

Length of the preamble.

int8_t `tx_power`

TX-power in dBm to use for transmission.

bool `tx`

Set to true for transmission, false for receiving.

bool `iq_inverted`

Invert the In-Phase and Quadrature (IQ) signals.

Normally this should be set to false. In advanced use-cases where a differentation is needed between “uplink” and “downlink” traffic, the IQ can be inverted to create two different channels on the same frequency

bool `public_network`

Sets the sync-byte to use:

- false: for using the private network sync-byte
- true: for using the public network sync-byte The public network sync-byte is only intended for advanced usage. Normally the private network sync-byte should be used for peer to peer communications and the LoRaWAN APIs should be used for interacting with a public network.

LoRaWAN

Related code samples

LoRaWAN FUOTA
Perform a LoRaWAN firmware-upgrade over the air (FUOTA) operation.

LoRaWAN class A device
Join a LoRaWAN network and send a message periodically.

group `lorawan_api`

Since
2.5

Version
0.1.0

Defines

`LW_RECV_PORT_ANY`

Flag to indicate receiving on any port.

Typedefs

typedef uint8_t (*`lorawan_battery_level_cb_t`)(void)

Defines the battery level callback handler function signature.

Retval 0
if the node is connected to an external power source

Retval 1..254
battery level, where 1 is the minimum and 254 is the maximum value

Retval 255
if the node was not able to measure the battery level

typedef void (*`lorawan_dr_changed_cb_t`)(enum *lorawan_datarate* dr)

Defines the datarate changed callback handler function signature.

Param dr
Updated datarate.

Enums

enum `lorawan_class`

LoRaWAN class types.

Values:

enumerator `LORAWAN_CLASS_A` = 0x00

Class A device.

enumerator `LORAWAN_CLASS_B` = 0x01

Class B device.

enumerator `LORAWAN_CLASS_C` = 0x02

Class C device.

enum `lorawan_act_type`

LoRaWAN activation types.

Values:

enumerator `LORAWAN_ACT_OTAA` = 0

Over-the-Air Activation (OTAA)

enumerator `LORAWAN_ACT_ABP`

Activation by Personalization (ABP)

enum `lorawan_channels_mask_size`

LoRaWAN channels mask sizes.

Values:

enumerator `LORAWAN_CHANNELS_MASK_SIZE_AS923` = 1

Region AS923 mask size.

enumerator `LORAWAN_CHANNELS_MASK_SIZE_AU915` = 6

Region AU915 mask size.

enumerator `LORAWAN_CHANNELS_MASK_SIZE_CN470` = 6

Region CN470 mask size.

enumerator `LORAWAN_CHANNELS_MASK_SIZE_CN779` = 1

Region CN779 mask size.

enumerator `LORAWAN_CHANNELS_MASK_SIZE_EU433` = 1

Region EU433 mask size.

enumerator `LORAWAN_CHANNELS_MASK_SIZE_EU868` = 1

Region EU868 mask size.

enumerator LORAWAN_CHANNELS_MASK_SIZE_KR920 = 1
Region KR920 mask size.

enumerator LORAWAN_CHANNELS_MASK_SIZE_IN865 = 1
Region IN865 mask size.

enumerator LORAWAN_CHANNELS_MASK_SIZE_US915 = 6
Region US915 mask size.

enumerator LORAWAN_CHANNELS_MASK_SIZE_RU864 = 1
Region RU864 mask size.

enum lorawan_datarate
LoRaWAN datarate types.
Values:

enumerator LORAWAN_DR_0 = 0
DR0 data rate.

enumerator LORAWAN_DR_1
DR1 data rate.

enumerator LORAWAN_DR_2
DR2 data rate.

enumerator LORAWAN_DR_3
DR3 data rate.

enumerator LORAWAN_DR_4
DR4 data rate.

enumerator LORAWAN_DR_5
DR5 data rate.

enumerator LORAWAN_DR_6
DR6 data rate.

enumerator LORAWAN_DR_7
DR7 data rate.

enumerator LORAWAN_DR_8
DR8 data rate.

enumerator LORAWAN_DR_9
DR9 data rate.

enumerator LORAWAN_DR_10
DR10 data rate.

enumerator LORAWAN_DR_11
DR11 data rate.

enumerator LORAWAN_DR_12
DR12 data rate.

enumerator LORAWAN_DR_13
DR13 data rate.

enumerator LORAWAN_DR_14
DR14 data rate.

enumerator LORAWAN_DR_15
DR15 data rate.

enum lorawan_region
LoRaWAN region types.
Values:

enumerator LORAWAN_REGION_AS923
Asia 923 MHz frequency band.

enumerator LORAWAN_REGION_AU915
Australia 915 MHz frequency band.

enumerator LORAWAN_REGION_CN470
China 470 MHz frequency band.

enumerator LORAWAN_REGION_CN779
China 779 MHz frequency band.

enumerator LORAWAN_REGION_EU433
Europe 433 MHz frequency band.

enumerator LORAWAN_REGION_EU868
Europe 868 MHz frequency band.

enumerator LORAWAN_REGION_KR920
South Korea 920 MHz frequency band.

enumerator LORAWAN_REGION_IN865
India 865 MHz frequency band.

enumerator LORAWAN_REGION_US915
United States 915 MHz frequency band.

enumerator LORAWAN_REGION_RU864
Russia 864 MHz frequency band.

enum `lorawan_message_type`

LoRaWAN message types.

Values:

enumerator `LORAWAN_MSG_UNCONFIRMED` = 0

Unconfirmed message.

enumerator `LORAWAN_MSG_CONFIRMED`

Confirmed message.

Functions

void `lorawan_register_battery_level_callback`(*lorawan_battery_level_cb_t* cb)

Register a battery level callback function.

Provide the LoRaWAN stack with a function to be called whenever a battery level needs to be read.

Should no callback be provided the lorawan backend will report 255.

Parameters

- `cb` – Pointer to the battery level function

void `lorawan_register_downlink_callback`(struct *lorawan_downlink_cb* *cb)

Register a callback to be run on downlink packets.

Parameters

- `cb` – Pointer to structure containing callback parameters

void `lorawan_register_dr_changed_callback`(*lorawan_dr_changed_cb_t* cb)

Register a callback to be called when the datarate changes.

The callback is called once upon successfully joining a network and again each time the datarate changes due to ADR.

Parameters

- `cb` – Pointer to datarate update callback

int `lorawan_join`(const struct *lorawan_join_config* *config)

Join the LoRaWAN network.

Join the LoRaWAN network using OTAA or AWB.

Parameters

- `config` – Configuration to be used

Returns

0 if successful, negative errno code if failure

int `lorawan_start`(void)

Start the LoRaWAN stack.

This function need to be called before joining the network.

Returns

0 if successful, negative errno code if failure

int `lorawan_send`(uint8_t port, uint8_t *data, uint8_t len, enum *lorawan_message_type* type)

Send data to the LoRaWAN network.

Send data to the connected LoRaWAN network.

Parameters

- `port` – Port to be used for sending data. Must be set if the payload is not empty.
- `data` – Data buffer to be sent
- `len` – Length of the buffer to be sent. Maximum length of this buffer is 255 bytes but the actual payload size varies with region and datarate.
- `type` – Specifies if the message shall be confirmed or unconfirmed. Must be one of *lorawan_message_type*.

Returns

0 if successful, negative errno code if failure

int `lorawan_set_class`(enum *lorawan_class* dev_class)

Set the current device class.

Change the current device class. This function may be called before or after a network connection has been established.

Parameters

- `dev_class` – New device class

Returns

0 if successful, negative errno code if failure

int `lorawan_set_conf_msg_tries`(uint8_t tries)

Set the number of tries used for transmissions.

Parameters

- `tries` – Number of tries to be used

Returns

0 if successful, negative errno code if failure

void `lorawan_enable_adr`(bool enable)

Enable Adaptive Data Rate (ADR)

Control whether adaptive data rate (ADR) is enabled. When ADR is enabled, the data rate is treated as a default data rate that will be used if the ADR algorithm has not established a data rate. ADR should normally only be enabled for devices with stable RF conditions (i.e., devices in a mostly static location).

Parameters

- `enable` – Enable or Disable adaptive data rate.

int `lorawan_set_channels_mask`(uint16_t *channels_mask, size_t channels_mask_size)

Set the channels mask.

Change the default channels mask. When mask is not changed, all the channels can be used for data transmission. Some Network Servers don't use all the channels, in this case, the channels mask must be provided.

Parameters

- `channels_mask` – Buffer with channels mask to be used.
- `channels_mask_size` – Size of channels mask buffer.

Return values

- `0` – successful
- `-EINVAL` – channels mask or channels mask size is wrong

int `lorawan_set_datarate`(enum *lorawan_datarate* dr)

Set the default data rate.

Change the default data rate.

Parameters

- `dr` – Data rate used for transmissions

Returns

0 if successful, negative errno code if failure

enum *lorawan_datarate* `lorawan_get_min_datarate`(void)

Get the minimum possible datarate.

The minimum possible datarate may change in response to a TxParamSetupReq command from the network server.

Returns

Minimum possible data rate

void `lorawan_get_payload_sizes`(uint8_t *max_next_payload_size, uint8_t *max_payload_size)

Get the current payload sizes.

Query the current payload sizes. The maximum payload size varies with datarate, while the current payload size can be less due to MAC layer commands which are inserted into uplink packets.

Parameters

- `max_next_payload_size` – Maximum payload size for the next transmission
- `max_payload_size` – Maximum payload size for this datarate

int `lorawan_set_region`(enum *lorawan_region* region)

Set the region and frequency to be used.

Control the LoRa region and frequency settings. This should be called before lorawan_start(). If you only have support for a single region selected via Kconfig, this function does not need to be called at all.

Parameters

- `region` – The region to be selected

Returns

0 if successful, negative errno otherwise

struct `lorawan_join_otaa`

#include <*lorawan.h*> LoRaWAN join parameters for over-the-Air activation (OTAA)

Note that all of the fields use LoRaWAN 1.1 terminology.

All parameters are optional if a secure element is present in which case the values stored in the secure element will be used instead.

Public Members

uint8_t *`join_eui`
Join EUI.

uint8_t *`nwk_key`
Network Key.

uint8_t *`app_key`
Application Key.

uint16_t `dev_nonce`
Device Nonce.

Starting with LoRaWAN 1.0.4 the DevNonce must be monotonically increasing for each OTAA join with the same EUI. The DevNonce should be stored in non-volatile memory by the application.

struct `lorawan_join_abp`
#include <*lorawan.h*> LoRaWAN join parameters for activation by personalization (ABP)

Public Members

uint32_t `dev_addr`
Device address on the network.

uint8_t *`app_skey`
Application session key.

uint8_t *`nwk_skey`
Network session key.

uint8_t *`app_eui`
Application EUI.

struct `lorawan_join_config`
#include <*lorawan.h*> LoRaWAN join parameters.

Public Members

struct *lorawan_join_otaa* `otaa`
OTAA join parameters.

struct *lorawan_join_abp* `abp`
ABP join parameters.

union lorawan_join_config

Join parameters.

uint8_t *dev_eui

Device EUI.

Optional if a secure element is present.

enum *lorawan_act_type* mode

Activation mode.

struct lorawan_downlink_cb

#include <*lorawan.h*> LoRaWAN downlink callback parameters.

Public Members

uint16_t port

Port to handle messages for.

- Port 0: TX packet acknowledgements
- Ports 1-255: Standard downlink port
- LW_RECV_PORT_ANY: All downlinks

void (*cb)(uint8_t port, bool data_pending, int16_t rssi, int8_t snr, uint8_t len, const uint8_t *data)

Callback function to run on downlink data.

> **Note**
>
> Callbacks are run on the system workqueue, and should therefore be as short as possible.

Param port
Port message was sent on
Param data_pending
Network server has more downlink packets pending
Param rssi
Received signal strength in dBm
Param snr
Signal to Noise ratio in dBm
Param len
Length of data received, will be 0 for ACKs
Param data
Data received, will be NULL for ACKs

sys_snode_t node

Node for callback list.

6.5 USB

USB device support

6.5.1 USB device support

- *Overview*
- *Supported USB classes*
  - *Audio*
  - *Bluetooth HCI USB transport layer*
  - *CDC ACM*
    - *Console over CDC ACM UART*
    - *CDC ACM UART as backend*
    - *POSIX default tty ECHO mitigation*
  - *DFU*
  - *USB Human Interface Devices (HID) support*
  - *Mass Storage Class*
  - *Networking*
- *Binary Device Object Store (BOS) support*
- *Implementing a non-standard USB class*
- *Interface number and endpoint address assignment*
- *Testing over USPIP in native_sim*
- *USB Vendor and Product identifiers*

Overview

The USB device stack is a hardware independent interface between USB device controller driver and USB device class drivers or customer applications. It is a port of the LPCUSB device stack and has been modified and expanded over time. It provides the following functionalities:

- Uses the *USB device controller driver API* provided by the device controller drivers to interact with the USB device controller.
- Responds to standard device requests and returns standard descriptors, essentially handling 'Chapter 9' processing, specifically the standard device requests in table 9-3 from the universal serial bus specification revision 2.0.
- Provides a programming interface to be used by USB device classes or customer applications. The APIs is described in include/zephyr/usb/usb_device.h

Note

It is planned to deprecate all APIs listed in *USB device support APIs* and the functions that depend on them between Zephyr v3.7.0 and v4.0.0, and remove them in v4.2.0. The new USB

device support, represented by the APIs in *New USB device support APIs*, will become the default in Zephyr v4.0.0.

Supported USB classes

Audio There is an experimental implementation of the Audio class. It follows specification version 1.00 (`bcdADC 0x0100`) and supports synchronous synchronisation type only. See usb-audio-headphones-microphone and usb-audio-headset samples for reference.

Bluetooth HCI USB transport layer Bluetooth HCI USB transport layer implementation uses *HCI RAW channel* to expose HCI interface to the host. It is not fully in line with the description in the Bluetooth specification and consists only of an interface with the endpoint configuration:

- HCI commands through control endpoint (host-to-device only)
- HCI events through interrupt IN endpoint
- ACL data through one bulk IN and one bulk OUT endpoints

A second interface for the voice channels has not been implemented as there is no support for this type in *Bluetooth*. It is not a big problem under Linux if HCI USB transport layer is the only interface that appears in the configuration, the btusb driver would not try to claim a second (isochronous) interface. The consequence is that if HCI USB is used in a composite configuration and is the first interface, then the Linux btusb driver will claim both the first and the next interface, preventing other composite functions from working. Because of this problem, HCI USB should not be used in a composite configuration. This problem is fixed in the implementation for new USB support.

See bluetooth-hci-usb-sample sample for reference.

CDC ACM The CDC ACM class is used as backend for different subsystems in Zephyr. However, its configuration may not be easy for the inexperienced user. Below is a description of the different use cases and some pitfalls.

The interface for CDC ACM user is *Universal Asynchronous Receiver-Transmitter (UART)* driver API. But there are two important differences in behavior to a real UART controller:

- Data transfer is only possible after the USB device stack has been initialized and started, until then any data is discarded
- If device is connected to the host, it still needs an application on the host side which requests the data
- The CDC ACM poll out implementation follows the API and blocks when the TX ring buffer is full only if the hw-flow-control property is enabled and called from a non-ISR context.

The devicetree compatible property for CDC ACM UART is `zephyr,cdc-acm-uart`. CDC ACM support is automatically selected when USB device support is enabled and a compatible node in the devicetree sources is present. If necessary, CDC ACM support can be explicitly disabled by `CONFIG_USB_CDC_ACM`. About four CDC ACM UART instances can be defined and used, limited by the maximum number of supported endpoints on the controller.

CDC ACM UART node is supposed to be child of a USB device controller node. Since the designation of the controller nodes varies from vendor to vendor, and our samples and application should be as generic as possible, the default USB device controller is usually assigned an `zephyr_udc0` node label. Often, CDC ACM UART is described in a devicetree overlay file and looks like this:

```
&zephyr_udc0 {
        cdc_acm_uart0: cdc_acm_uart0 {
                compatible = "zephyr,cdc-acm-uart";
                label = "CDC_ACM_0";
        };
};
```

Sample usb-cdc-acm has similar overlay files. And since no special properties are present, it may seem overkill to use devicetree to describe CDC ACM UART. The motivation behind using devicetree is the easy interchangeability of a real UART controller and CDC ACM UART in applications.

Console over CDC ACM UART With the CDC ACM UART node from above and `zephyr,console` property of the chosen node, we can describe that CDC ACM UART is to be used with the console. A similar overlay file is used by the usb-cdc-acm-console sample.

```
/ {
        chosen {
                zephyr,console = &cdc_acm_uart0;
        };
};

&zephyr_udc0 {
        cdc_acm_uart0: cdc_acm_uart0 {
                compatible = "zephyr,cdc-acm-uart";
                label = "CDC_ACM_0";
        };
};
```

Before the application uses the console, it is recommended to wait for the DTR signal:

```
const struct device *const dev = DEVICE_DT_GET(DT_CHOSEN(zephyr_console));
uint32_t dtr = 0;

if (usb_enable(NULL)) {
        return;
}

while (!dtr) {
        uart_line_ctrl_get(dev, UART_LINE_CTRL_DTR, &dtr);
        k_sleep(K_MSEC(100));
}

printk("nuqneH\n");
```

CDC ACM UART as backend As for the console sample, it is possible to configure CDC ACM UART as backend for other subsystems by setting *Chosen nodes* properties.

List of few Zephyr specific chosen properties which can be used to select CDC ACM UART as backend for a subsystem or application:

- `zephyr,bt-c2h-uart` used in Bluetooth, for example see bluetooth-hci-uart-sample
- `zephyr,ot-uart` used in OpenThread, for example see coprocessor
- `zephyr,shell-uart` used by shell for serial backend, for example see samples/subsys/shell/shell_module
- `zephyr,uart-mcumgr` used by smp-svr sample

POSIX default tty ECHO mitigation POSIX systems, like Linux, default to enabling ECHO on tty devices. Host side application can disable ECHO by calling `open()` on the tty device and issuing `ioctl()` (preferably via `tcsetattr()`) to disable echo if it is not desired. Unfortunately, there is an inherent race between the `open()` and `ioctl()` where the ECHO is enabled and any characters received (even if host application does not call `read()`) will be echoed back. This issue is especially visible when the CDC ACM port is used without any real UART on the other side because there is no arbitrary delay due to baud rate.

To mitigate the issue, Zephyr CDC ACM implementation arms IN endpoint with ZLP after device is configured. When the host reads the ZLP, which is pretty much the best indication that host application has opened the tty device, Zephyr will force `CONFIG_CDC_ACM_TX_DELAY_MS` millisecond delay before real payload is sent. This should allow sufficient time for first, and only first, application that opens the tty device to disable ECHO if ECHO is not desired. If ECHO is not desired at all from CDC ACM device it is best to set up udev rule to disable ECHO as soon as device is connected.

ECHO is particurarly unwanted when CDC ACM instance is used for Zephyr shell, because the control characters to set color sent back to shell are interpreted as (invalid) command and user will see garbage as a result. While minicom does disable ECHO by default, on exit with reset it will restore the termios settings to whatever was set on entry. Therefore, if minicom is the first application to open the tty device, the exit with reset will enable ECHO back and thus set up a problem for the next application (which cannot be mitigated at Zephyr side). To prevent the issue it is recommended either to leave minicom without reset or to disable ECHO before minicom is started.

DFU USB DFU class implementation is tightly coupled to *Device Firmware Upgrade* and *MCUBoot API*. This means that the target platform must support the *Flash Image* API.

See usb-dfu sample for reference.

USB Human Interface Devices (HID) support HID support abuses *Device Driver Model* simply to allow applications to use the *device_get_binding()*. Note that there is no HID device API as such, instead the interface is provided by *hid_ops*. The default instance name is `HID_n`, where n can be {0, 1, 2, ...} depending on the `CONFIG_USB_HID_DEVICE_COUNT`.

Each HID instance requires a HID report descriptor. The interface to the core and the report descriptor must be registered using *usb_hid_register_device()*.

As the USB HID specification is not only used by the USB subsystem, the USB HID API reference is split into two parts, *Human Interface Devices (HID)* and *USB HID Class API*. HID helper macros from *Human Interface Devices (HID)* should be used to compose a HID report descriptor. Macro names correspond to those used in the USB HID specification.

For the HID class interface, an IN interrupt endpoint is required for each instance, an OUT interrupt endpoint is optional. Thus, the minimum implementation requirement for *hid_ops* is to provide `int_in_ready` callback.

```
#define REPORT_ID               1
static bool configured;
static const struct device *hdev;

static void int_in_ready_cb(const struct device *dev)
{
        static uint8_t report[2] = {REPORT_ID, 0};

        if (hid_int_ep_write(hdev, report, sizeof(report), NULL)) {
                LOG_ERR("Failed to submit report");
        } else {
                report[1]++;
        }
```

(continues on next page)

(continued from previous page)

```
}

static void status_cb(enum usb_dc_status_code status, const uint8_t *param)
{
        if (status == USB_DC_RESET) {
                configured = false;
        }

        if (status == USB_DC_CONFIGURED && !configured) {
                int_in_ready_cb(hdev);
                configured = true;
        }
}

static const uint8_t hid_report_desc[] = {
        HID_USAGE_PAGE(HID_USAGE_GEN_DESKTOP),
        HID_USAGE(HID_USAGE_GEN_DESKTOP_UNDEFINED),
        HID_COLLECTION(HID_COLLECTION_APPLICATION),
        HID_LOGICAL_MIN8(0x00),
        HID_LOGICAL_MAX16(0xFF, 0x00),
        HID_REPORT_ID(REPORT_ID),
        HID_REPORT_SIZE(8),
        HID_REPORT_COUNT(1),
        HID_USAGE(HID_USAGE_GEN_DESKTOP_UNDEFINED),
        HID_INPUT(0x02),
        HID_END_COLLECTION,
};

static const struct hid_ops my_ops = {
        .int_in_ready = int_in_ready_cb,
};

int main(void)
{
        int ret;

        hdev = device_get_binding("HID_0");
        if (hdev == NULL) {
                return -ENODEV;
        }

        usb_hid_register_device(hdev, hid_report_desc, sizeof(hid_report_desc),
                                &my_ops);

        ret = usb_hid_init(hdev);
        if (ret) {
                return ret;
        }

        return usb_enable(status_cb);
}
```

If the application wishes to receive output reports via the OUT interrupt endpoint, it must enable `CONFIG_ENABLE_HID_INT_OUT_EP` and provide `int_out_ready` callback. The disadvantage of this is that Kconfig options such as `CONFIG_ENABLE_HID_INT_OUT_EP` or `CONFIG_HID_INTERRUPT_EP_MPS` apply to all instances. This design issue will be fixed in the HID class implementation for the new USB support.

See usb-hid or usb-hid-mouse sample for reference.

Mass Storage Class MSC follows Bulk-Only Transport specification and uses *Disk Access* to access and expose a RAM disk, emulated block device on a flash partition, or SD Card to the host. Only one disk instance can be exported at a time.

The disc to be used by the implementation is set by the `CONFIG_MASS_STORAGE_DISK_NAME` and should be the same as the name used by the disc access driver that the application wants to expose to the host. SD card disk drivers use options `CONFIG_MMC_VOLUME_NAME` or `CONFIG_SDMMC_VOLUME_NAME`, and flash and RAM disk drivers use node property `disk-name` to set the disk name.

For the emulated block device on a flash partition, the flash partition and flash disk to be used must be described in the devicetree. If a storage partition is already described at the board level, application devicetree overlay must also delete `storage_partition` node first. `CONFIG_MASS_STORAGE_DISK_NAME` should be the same as `disk-name` property.

```
/delete-node/ &storage_partition;

&mx25r64 {
        partitions {
                compatible = "fixed-partitions";
                #address-cells = <1>;
                #size-cells = <1>;

                storage_partition: partition@0 {
                        label = "storage";
                        reg = <0x00000000 0x00020000>;
                };
        };
};

/ {
        msc_disk0 {
                compatible = "zephyr,flash-disk";
                partition = <&storage_partition>;
                disk-name = "NAND";
                cache-size = <4096>;
        };
};
```

The `disk-property` "NAND" may be confusing, but it is simply how some file systems identifies the disc. Therefore, if the application also accesses the file system on the exposed disc, default names should be used, see usb-mass sample for reference.

Networking There are three implementations that work in a similar way, providing a virtual Ethernet connection between the remote (USB host) and Zephyr network support.

- CDC ECM class, enabled with `CONFIG_USB_DEVICE_NETWORK_ECM`
- CDC EEM class, enabled with `CONFIG_USB_DEVICE_NETWORK_EEM`
- RNDIS support, enabled with `CONFIG_USB_DEVICE_NETWORK_RNDIS`

See zperf or socket-dumb-http-server for reference. Typically, users will need to add a configuration file overlay to the build, such as samples/net/zperf/overlay-netusb.conf.

Applications using RNDIS support should enable `CONFIG_USB_DEVICE_OS_DESC` for a better user experience on a host running Microsoft Windows OS.

Binary Device Object Store (BOS) support

BOS handling can be enabled with Kconfig option `CONFIG_USB_DEVICE_BOS`. This option also has the effect of changing device descriptor `bcdUSB` to `0210`. The application should register descrip-

tors such as Capability Descriptor using *usb_bos_register_cap()*. Registered descriptors are added to the root BOS descriptor and handled by the stack.

See webusb sample for reference.

Implementing a non-standard USB class

The configuration of USB device is done in the stack layer.

The following structures and callbacks need to be defined:

- Part of USB Descriptor table
- USB Endpoint configuration table
- USB Device configuration structure
- Endpoint callbacks
- Optionally class, vendor and custom handlers

For example, for the USB loopback application:

```
struct usb_loopback_config {
        struct usb_if_descriptor if0;
        struct usb_ep_descriptor if0_out_ep;
        struct usb_ep_descriptor if0_in_ep;
} __packed;

USBD_CLASS_DESCR_DEFINE(primary, 0) struct usb_loopback_config loopback_cfg = {
        /* Interface descriptor 0 */
        .if0 = {
                .bLength = sizeof(struct usb_if_descriptor),
                .bDescriptorType = USB_DESC_INTERFACE,
                .bInterfaceNumber = 0,
                .bAlternateSetting = 0,
                .bNumEndpoints = 2,
                .bInterfaceClass = USB_BCC_VENDOR,
                .bInterfaceSubClass = 0,
                .bInterfaceProtocol = 0,
                .iInterface = 0,
        },

        /* Data Endpoint OUT */
        .if0_out_ep = {
                .bLength = sizeof(struct usb_ep_descriptor),
                .bDescriptorType = USB_DESC_ENDPOINT,
                .bEndpointAddress = LOOPBACK_OUT_EP_ADDR,
                .bmAttributes = USB_DC_EP_BULK,
                .wMaxPacketSize = sys_cpu_to_le16(CONFIG_LOOPBACK_BULK_EP_MPS),
                .bInterval = 0x00,
        },

        /* Data Endpoint IN */
        .if0_in_ep = {
                .bLength = sizeof(struct usb_ep_descriptor),
                .bDescriptorType = USB_DESC_ENDPOINT,
                .bEndpointAddress = LOOPBACK_IN_EP_ADDR,
                .bmAttributes = USB_DC_EP_BULK,
                .wMaxPacketSize = sys_cpu_to_le16(CONFIG_LOOPBACK_BULK_EP_MPS),
                .bInterval = 0x00,
        },
};
```

Endpoint configuration:

```
static struct usb_ep_cfg_data ep_cfg[] = {
        {
                .ep_cb = loopback_out_cb,
                .ep_addr = LOOPBACK_OUT_EP_ADDR,
        },
        {
                .ep_cb = loopback_in_cb,
                .ep_addr = LOOPBACK_IN_EP_ADDR,
        },
};
```

USB Device configuration structure:

```
USBD_DEFINE_CFG_DATA(loopback_config) = {
        .usb_device_description = NULL,
        .interface_config = loopback_interface_config,
        .interface_descriptor = &loopback_cfg.if0,
        .cb_usb_status = loopback_status_cb,
        .interface = {
                .class_handler = NULL,
                .custom_handler = NULL,
                .vendor_handler = loopback_vendor_handler,
        },
        .num_endpoints = ARRAY_SIZE(ep_cfg),
        .endpoint = ep_cfg,
};
```

The vendor device requests are forwarded by the USB stack core driver to the class driver through the registered vendor handler.

For the loopback class driver, `loopback_vendor_handler()` processes the vendor requests:

```
static int loopback_vendor_handler(struct usb_setup_packet *setup,
                                   int32_t *len, uint8_t **data)
{
        LOG_DBG("Class request: bRequest 0x%x bmRequestType 0x%x len %d",
                setup->bRequest, setup->bmRequestType, *len);

        if (setup->RequestType.recipient != USB_REQTYPE_RECIPIENT_DEVICE) {
                return -ENOTSUP;
        }

        if (usb_reqtype_is_to_device(setup) &&
            setup->bRequest == 0x5b) {
                LOG_DBG("Host-to-Device, data %p", *data);
                /*
                 * Copy request data in loopback_buf buffer and reuse
                 * it later in control device-to-host transfer.
                 */
                memcpy(loopback_buf, *data,
                       MIN(sizeof(loopback_buf), setup->wLength));
                return 0;
        }

        if ((usb_reqtype_is_to_host(setup)) &&
            (setup->bRequest == 0x5c)) {
                LOG_DBG("Device-to-Host, wLength %d, data %p",
                        setup->wLength, *data);
                *data = loopback_buf;
                *len = MIN(sizeof(loopback_buf), setup->wLength);
                return 0;
        }
```

(continues on next page)

(continued from previous page)

```
31
32          return -ENOTSUP;
33  }
```

The class driver waits for the `USB_DC_CONFIGURED` device status code before transmitting any data.

Interface number and endpoint address assignment

In USB terminology, a `function` is a device that provides a capability to the host, such as a HID class device that implements a keyboard. A function contains a collection of `interfaces`; at least one interface is required. An interface may contain device `endpoints`; for example, at least one input endpoint is required to implement a HID class device, and no endpoints are required to implement a USB DFU class. A USB device that combines functions is a multifunction USB device, for example, a combination of a HID class device and a CDC ACM device.

With Zephyr RTOS USB support, various combinations are possible with built-in USB classes/functions or custom user implementations. The limitation is the number of available device endpoints. Each device endpoint is uniquely addressable. The endpoint address is a combination of endpoint direction and endpoint number, a four-bit value. Endpoint number zero is used for the default control method to initialize and configure a USB device. By specification, a maximum of `15 IN` and `15 OUT` device endpoints are also available for use in functions. The actual number depends on the device controller used. Not all controllers support the maximum number of endpoints and all endpoint types. For example, a device controller might support one IN and one OUT isochronous endpoint, but only for endpoint number 8, resulting in endpoint addresses 0x88 and 0x08. Also, one controller may be able to have IN/OUT endpoints on the same endpoint number, interrupt IN endpoint 0x81 and bulk OUT endpoint 0x01, while the other may only be able to handle one endpoint per endpoint number. Information about the number of interfaces, interface associations, endpoint types, and addresses is provided to the host by the interface, interface specific, and endpoint descriptors.

Host driver for specific function, uses interface and endpoint descriptor to obtain endpoint addresses, types, and other properties. This allows function host drivers to be generic, for example, a multi-function device consisting of one or more CDC ACM and one or more CDC ECM class implementations is possible and no specific drivers are required.

Interface and endpoint descriptors of built-in USB class/function implementations in Zephyr RTOS typically have default interface numbers and endpoint addresses assigned in ascending order. During initialization, default interface numbers may be reassigned based on the number of interfaces in a given configuration. Endpoint addresses are reassigned based on controller capabilities, since certain endpoint combinations are not possible with every controller, and the number of interfaces in a given configuration. This also means that the device side class/function in the Zephyr RTOS must check the actual interface and endpoint descriptor values at runtime. This mechanism also allows as to provide generic samples and generic multifunction samples that are limited only by the resources provided by the controller, such as the number of endpoints and the size of the endpoint FIFOs.

There may be host drivers for a specific function, for example in the Linux Kernel, where the function driver does not read interface and endpoint descriptors to check interface numbers or endpoint addresses, but instead uses hardcoded values. Therefore, the host driver cannot be used in a generic way, meaning it cannot be used with different device controllers and different device configurations in combination with other functions. This may also be because the driver is designed for a specific hardware and is not intended to be used with a clone of this specific hardware. On the contrary, if the driver is generic in nature and should work with different hardware variants, then it must not use hardcoded interface numbers and endpoint addresses. It is not possible to disable endpoint reassignment in Zephyr RTOS, which may prevent you from implementing a hardware-clone firmware. Instead, if possible, the host driver implementation should be fixed to use values from the interface and endpoint descriptor.

Testing over USPIP in native_sim

A virtual USB controller implemented through USBIP might be used to test the USB device stack. Follow the general build procedure to build the USB sample for the native_sim configuration.

Run built sample with:

```
west build -t run
```

In a terminal window, run the following command to list USB devices:

```
$ usbip list -r localhost
Exportable USB devices
======================
 - 127.0.0.1
        1-1: unknown vendor : unknown product (2fe3:0100)
           : /sys/devices/pci0000:00/0000:00:01.2/usb1/1-1
           : (Defined at Interface level) (00/00/00)
           :  0 - Vendor Specific Class / unknown subclass / unknown protocol (ff/00/00)
```

In a terminal window, run the following command to attach the USB device:

```
$ sudo usbip attach -r localhost -b 1-1
```

The USB device should be connected to your Linux host, and verified with the following commands:

```
$ sudo usbip port
Imported USB devices
====================
Port 00: <Port in Use> at Full Speed(12Mbps)
       unknown vendor : unknown product (2fe3:0100)
       7-1 -> usbip://localhost:3240/1-1
           -> remote bus/dev 001/002
$ lsusb -d 2fe3:0100
Bus 007 Device 004: ID 2fe3:0100
```

USB Vendor and Product identifiers

The USB Vendor ID for the Zephyr project is `0x2FE3`. This USB Vendor ID must not be used when a vendor integrates Zephyr USB device support into its own product.

Each USB sample has its own unique Product ID. The USB maintainer, if one is assigned, or otherwise the Zephyr Technical Steering Committee, may allocate other USB Product IDs based on well-motivated and documented requests.

The following Product IDs are currently used:

Sample	PID
usb-cdc-acm	0x0001
usb-cdc-acm-composite	0x0002
Reserved (previously: usb-hid-cdc)	0x0003
usb-cdc-acm-console	0x0004
usb-dfu (Run-Time)	0x0005
usb-hid	0x0006
usb-hid-mouse	0x0007
usb-mass	0x0008
testusb-app	0x0009
webusb	0x000A
bluetooth-hci-usb-sample	0x000B
bluetooth-hci-usb-h4-sample	0x000C
wpan-usb	0x000D
uac2-explicit-feedback	0x000E
usb-dfu (DFU Mode)	0xFFFF

The USB device descriptor field `bcdDevice` (Device Release Number) represents the Zephyr kernel major and minor versions as a binary coded decimal value.

6.5.2 USB device support APIs

USB device controller driver API

The USB device controller driver API is described in include/zephyr/drivers/usb/usb_dc.h and sometimes referred to as the `usb_dc` API.

This API has some limitations by design, it does not follow *Device Driver Model* and is being replaced by *USB device controller (UDC) driver API*.

API reference

group `_usb_device_controller_api`

USB Device Controller API.

Typedefs

typedef void (*`usb_dc_ep_callback`)(uint8_t ep, enum *usb_dc_ep_cb_status_code* cb_status)

Callback function signature for the USB Endpoint status.

typedef void (*`usb_dc_status_callback`)(enum *usb_dc_status_code* cb_status, const uint8_t *param)

Callback function signature for the device.

Enums

enum `usb_dc_status_code`

USB Driver Status Codes.

Status codes reported by the registered device status callback.

Values:

enumerator USB_DC_ERROR
USB error reported by the controller.

enumerator USB_DC_RESET
USB reset.

enumerator USB_DC_CONNECTED
USB connection established, hardware enumeration is completed.

enumerator USB_DC_CONFIGURED
USB configuration done.

enumerator USB_DC_DISCONNECTED
USB connection lost.

enumerator USB_DC_SUSPEND
USB connection suspended by the HOST.

enumerator USB_DC_RESUME
USB connection resumed by the HOST.

enumerator USB_DC_INTERFACE
USB interface selected.

enumerator USB_DC_SET_HALT
Set Feature ENDPOINT_HALT received.

enumerator USB_DC_CLEAR_HALT
Clear Feature ENDPOINT_HALT received.

enumerator USB_DC_SOF
Start of Frame received.

enumerator USB_DC_UNKNOWN
Initial USB connection status.

enum usb_dc_ep_cb_status_code
USB Endpoint Callback Status Codes.

Status Codes reported by the registered endpoint callback.

Values:

enumerator USB_DC_EP_SETUP
SETUP received.

enumerator USB_DC_EP_DATA_OUT
Out transaction on this EP, data is available for read.

enumerator USB_DC_EP_DATA_IN
In transaction done on this EP.

enum usb_dc_ep_transfer_type
USB Endpoint Transfer Type.

Values:

enumerator USB_DC_EP_CONTROL = 0
Control type endpoint.

enumerator USB_DC_EP_ISOCHRONOUS
Isochronous type endpoint.

enumerator USB_DC_EP_BULK
Bulk type endpoint.

enumerator USB_DC_EP_INTERRUPT
Interrupt type endpoint

enum usb_dc_ep_synchronozation_type
USB Endpoint Synchronization Type.

Note

Valid only for Isochronous Endpoints

Values:

enumerator USB_DC_EP_NO_SYNCHRONIZATION = (0U « 2U)
No Synchronization.

enumerator USB_DC_EP_ASYNCHRONOUS = (1U « 2U)
Asynchronous.

enumerator USB_DC_EP_ADAPTIVE = (2U « 2U)
Adaptive.

enumerator USB_DC_EP_SYNCHRONOUS = (3U « 2U)
Synchronous.

Functions

int usb_dc_attach(void)
Attach USB for device connection.

Function to attach USB for device connection. Upon success, the USB PLL is enabled, and the USB device is now capable of transmitting and receiving on the USB bus and of generating interrupts.

Returns
0 on success, negative errno code on fail.

int `usb_dc_detach`(void)

Detach the USB device.

Function to detach the USB device. Upon success, the USB hardware PLL is powered down and USB communication is disabled.

Returns
0 on success, negative errno code on fail.

int `usb_dc_reset`(void)

Reset the USB device.

This function returns the USB device and firmware back to it's initial state. N.B. the USB PLL is handled by the usb_detach function

Returns
0 on success, negative errno code on fail.

int `usb_dc_set_address`(const uint8_t addr)

Set USB device address.

Parameters

- `addr` – **[in]** Device address

Returns
0 on success, negative errno code on fail.

void `usb_dc_set_status_callback`(const *usb_dc_status_callback* cb)

Set USB device controller status callback.

Function to set USB device controller status callback. The registered callback is used to report changes in the status of the device controller. The status code are described by the usb_dc_status_code enumeration.

Parameters

- `cb` – **[in]** Callback function

int `usb_dc_ep_check_cap`(const struct *usb_dc_ep_cfg_data* *const cfg)

check endpoint capabilities

Function to check capabilities of an endpoint. *usb_dc_ep_cfg_data* structure provides the endpoint configuration parameters: endpoint address, endpoint maximum packet size and endpoint type. The driver should check endpoint capabilities and return 0 if the endpoint configuration is possible.

Parameters

- `cfg` – **[in]** Endpoint config

Returns
0 on success, negative errno code on fail.

int `usb_dc_ep_configure`(const struct *usb_dc_ep_cfg_data* *const cfg)

Configure endpoint.

Function to configure an endpoint. *usb_dc_ep_cfg_data* structure provides the endpoint configuration parameters: endpoint address, endpoint maximum packet size and endpoint type.

Parameters

- `cfg` – **[in]** Endpoint config

Returns
0 on success, negative errno code on fail.

int usb_dc_ep_set_stall(const uint8_t ep)

Set stall condition for the selected endpoint.

Parameters

- ep – **[in]** Endpoint address corresponding to the one listed in the device configuration table

Returns
0 on success, negative errno code on fail.

int usb_dc_ep_clear_stall(const uint8_t ep)

Clear stall condition for the selected endpoint.

Parameters

- ep – **[in]** Endpoint address corresponding to the one listed in the device configuration table

Returns
0 on success, negative errno code on fail.

int usb_dc_ep_is_stalled(const uint8_t ep, uint8_t *const stalled)

Check if the selected endpoint is stalled.

Parameters

- ep – **[in]** Endpoint address corresponding to the one listed in the device configuration table
- stalled – **[out]** Endpoint stall status

Returns
0 on success, negative errno code on fail.

int usb_dc_ep_halt(const uint8_t ep)

Halt the selected endpoint.

Parameters

- ep – **[in]** Endpoint address corresponding to the one listed in the device configuration table

Returns
0 on success, negative errno code on fail.

int usb_dc_ep_enable(const uint8_t ep)

Enable the selected endpoint.

Function to enable the selected endpoint. Upon success interrupts are enabled for the corresponding endpoint and the endpoint is ready for transmitting/receiving data.

Parameters

- ep – **[in]** Endpoint address corresponding to the one listed in the device configuration table

Returns
0 on success, negative errno code on fail.

int usb_dc_ep_disable(const uint8_t ep)

Disable the selected endpoint.

Function to disable the selected endpoint. Upon success interrupts are disabled for the corresponding endpoint and the endpoint is no longer able for transmitting/receiving data.

Parameters

- `ep` – **[in]** Endpoint address corresponding to the one listed in the device configuration table

Returns
0 on success, negative errno code on fail.

int `usb_dc_ep_flush`(const uint8_t ep)

Flush the selected endpoint.

This function flushes the FIFOs for the selected endpoint.

Parameters

- `ep` – **[in]** Endpoint address corresponding to the one listed in the device configuration table

Returns
0 on success, negative errno code on fail.

int `usb_dc_ep_write`(const uint8_t ep, const uint8_t *const data, const uint32_t data_len, uint32_t *const ret_bytes)

Write data to the specified endpoint.

This function is called to write data to the specified endpoint. The supplied usb_ep_callback function will be called when data is transmitted out.

Parameters

- `ep` – **[in]** Endpoint address corresponding to the one listed in the device configuration table
- `data` – **[in]** Pointer to data to write
- `data_len` – **[in]** Length of the data requested to write. This may be zero for a zero length status packet.
- `ret_bytes` – **[out]** Bytes scheduled for transmission. This value may be NULL if the application expects all bytes to be written

Returns
0 on success, negative errno code on fail.

int `usb_dc_ep_read`(const uint8_t ep, uint8_t *const data, const uint32_t max_data_len, uint32_t *const read_bytes)

Read data from the specified endpoint.

This function is called by the endpoint handler function, after an OUT interrupt has been received for that EP. The application must only call this function through the supplied usb_ep_callback function. This function clears the ENDPOINT NAK, if all data in the endpoint FIFO has been read, so as to accept more data from host.

Parameters

- `ep` – **[in]** Endpoint address corresponding to the one listed in the device configuration table
- `data` – **[in]** Pointer to data buffer to write to
- `max_data_len` – **[in]** Max length of data to read
- `read_bytes` – **[out]** Number of bytes read. If data is NULL and max_data_len is 0 the number of bytes available for read should be returned.

Returns
0 on success, negative errno code on fail.

int **usb_dc_ep_set_callback**(const uint8_t ep, const *usb_dc_ep_callback* cb)

Set callback function for the specified endpoint.

Function to set callback function for notification of data received and available to application or transmit done on the selected endpoint, NULL if callback not required by application code. The callback status code is described by usb_dc_ep_cb_status_code.

Parameters

- **ep** – **[in]** Endpoint address corresponding to the one listed in the device configuration table
- **cb** – **[in]** Callback function

Returns

0 on success, negative errno code on fail.

int **usb_dc_ep_read_wait**(uint8_t ep, uint8_t *data, uint32_t max_data_len, uint32_t *read_bytes)

Read data from the specified endpoint.

This is similar to usb_dc_ep_read, the difference being that, it doesn't clear the endpoint NAKs so that the consumer is not bogged down by further upcalls till he is done with the processing of the data. The caller should reactivate ep by invoking *usb_dc_ep_read_continue()* do so.

Parameters

- **ep** – **[in]** Endpoint address corresponding to the one listed in the device configuration table
- **data** – **[in]** Pointer to data buffer to write to
- **max_data_len** – **[in]** Max length of data to read
- **read_bytes** – **[out]** Number of bytes read. If data is NULL and max_data_len is 0 the number of bytes available for read should be returned.

Returns

0 on success, negative errno code on fail.

int **usb_dc_ep_read_continue**(uint8_t ep)

Continue reading data from the endpoint.

Clear the endpoint NAK and enable the endpoint to accept more data from the host. Usually called after *usb_dc_ep_read_wait()* when the consumer is fine to accept more data. Thus these calls together act as a flow control mechanism.

Parameters

- **ep** – **[in]** Endpoint address corresponding to the one listed in the device configuration table

Returns

0 on success, negative errno code on fail.

int **usb_dc_ep_mps**(uint8_t ep)

Get endpoint max packet size.

Parameters

- **ep** – **[in]** Endpoint address corresponding to the one listed in the device configuration table

Returns

Endpoint max packet size (mps)

int **usb_dc_wakeup_request**(void)

Start the host wake up procedure.

Function to wake up the host if it's currently in sleep mode.

Returns

0 on success, negative errno code on fail.

struct **usb_dc_ep_cfg_data**

#include <*usb_dc.h*> USB Endpoint Configuration.

Structure containing the USB endpoint configuration.

Public Members

uint8_t **ep_addr**

The number associated with the EP in the device configuration structure IN EP = 0x80 | <endpoint number> OUT EP = 0x00 | <endpoint number>

uint16_t **ep_mps**

Endpoint max packet size.

enum *usb_dc_ep_transfer_type* **ep_type**

Endpoint Transfer Type.

May be Bulk, Interrupt, Control or Isochronous

USB device stack API

API reference There are two ways to transmit data, using the 'low' level read/write API or the 'high' level transfer API.

Low level API

To transmit data to the host, the class driver should call usb_write(). Upon completion the registered endpoint callback will be called. Before sending another packet the class driver should wait for the completion of the previous write. When data is received, the registered endpoint callback is called. usb_read() should be used for retrieving the received data. For CDC ACM sample driver this happens via the OUT bulk endpoint handler (cdc_acm_bulk_out) mentioned in the endpoint array (cdc_acm_ep_data).

High level API

The usb_transfer method can be used to transfer data to/from the host. The transfer API will automatically split the data transmission into one or more USB transaction(s), depending endpoint max packet size. The class driver does not have to implement endpoint callback and should set this callback to the generic usb_transfer_ep_callback.

Related code samples

802.15.4 USB
Implement a device that exposes an IEEE 802.15.4 radio over USB.

Console over USB CDC ACM
Output "Hello World!" to the console over USB CDC ACM.

USB Audio headset
Implement a USB Audio headset device with audio IN/OUT loopback.

USB Audio microphone & headphones
Implement a USB Audio microphone + headphones device with audio IN/OUT loopback.

USB CDC-ACM
Use USB CDC-ACM driver to implement a serial port echo.

USB CDC-ACM composite
Implement a composite USB device exposing two serial ports using USB CDC-ACM driver.

USB DFU (Device Firmware Upgrade)
Implement device firmware upgrade using the USB DFU class driver.

USB HID (Human Interface Device)
Use USB HID driver to enumerate as a raw HID device.

USB HID mouse
Implement a basic HID mouse device.

USB Mass Storage
Expose board's RAM or FLASH as a USB disk using USB Mass Storage driver.

USB testing application
Test USB device drivers using a loopback function.

WebUSB
Receive and echo data from a web page using WebUSB API.

group `_usb_device_core_api`

USB Device Core Layer API.

Defines

`USB_TRANS_READ`

`USB_TRANS_WRITE`

`USB_TRANS_NO_ZLP`

`USB_DEVICE_BOS_DESC_DEFINE_CAP`

Helper macro to place the BOS compatibility descriptor in the right memory section.

Typedefs

typedef void (*`usb_ep_callback`)(uint8_t ep, enum *usb_dc_ep_cb_status_code* cb_status)

Callback function signature for the USB Endpoint status.

typedef int (*`usb_request_handler`)(struct usb_setup_packet *setup, int32_t *transfer_len, uint8_t **payload_data)

Callback function signature for class specific requests.

Function which handles Class specific requests corresponding to an interface number specified in the device descriptor table. For host to device direction the 'len' and 'payload_data' contain the length of the received data and the pointer to the received data respectively. For device to host class requests, 'len' and 'payload_data' should be set

by the callback function with the length and the address of the data to be transmitted buffer respectively.

typedef void (*`usb_interface_config`)(struct usb_desc_header *head, uint8_t bInterfaceNumber)

Function for interface runtime configuration.

typedef void (*`usb_transfer_callback`)(uint8_t ep, int tsize, void *priv)

Callback function signature for transfer completion.

Functions

int `usb_set_config`(const uint8_t *usb_descriptor)

Configure USB controller.

Function to configure USB controller. Configuration parameters must be valid or an error is returned

Parameters

- `usb_descriptor` – **[in]** USB descriptor table

Returns

0 on success, negative errno code on fail

int `usb_deconfig`(void)

Deconfigure USB controller.

This function returns the USB device to it's initial state

Returns

0 on success, negative errno code on fail

int `usb_enable`(*usb_dc_status_callback* status_cb)

Enable the USB subsystem and associated hardware.

This function initializes the USB core subsystem and enables the corresponding hardware so that it can begin transmitting and receiving on the USB bus, as well as generating interrupts.

Class-specific initialization and registration must be performed by the user before invoking this, so that any data or events on the bus are processed correctly by the associated class handling code.

Parameters

- `status_cb` – **[in]** Callback registered by user to notify about USB device controller state.

Returns

0 on success, negative errno code on fail.

int `usb_disable`(void)

Disable the USB device.

Function to disable the USB device. Upon success, the specified USB interface is clock gated in hardware, it is no longer capable of generating interrupts.

Returns

0 on success, negative errno code on fail

int `usb_write`(uint8_t ep, const uint8_t *data, uint32_t data_len, uint32_t *bytes_ret)

Write data to the specified endpoint.

Function to write data to the specified endpoint. The supplied usb_ep_callback will be called when transmission is done.

Parameters

- `ep` – **[in]** Endpoint address corresponding to the one listed in the device configuration table
- `data` – **[in]** Pointer to data to write
- `data_len` – **[in]** Length of data requested to write. This may be zero for a zero length status packet.
- `bytes_ret` – **[out]** Bytes written to the EP FIFO. This value may be NULL if the application expects all bytes to be written

Returns

0 on success, negative errno code on fail

int `usb_read`(uint8_t ep, uint8_t *data, uint32_t max_data_len, uint32_t *ret_bytes)

Read data from the specified endpoint.

This function is called by the Endpoint handler function, after an OUT interrupt has been received for that EP. The application must only call this function through the supplied usb_ep_callback function.

Parameters

- `ep` – **[in]** Endpoint address corresponding to the one listed in the device configuration table
- `data` – **[in]** Pointer to data buffer to write to
- `max_data_len` – **[in]** Max length of data to read
- `ret_bytes` – **[out]** Number of bytes read. If data is NULL and max_data_len is 0 the number of bytes available for read is returned.

Returns

0 on success, negative errno code on fail

int `usb_ep_set_stall`(uint8_t ep)

Set STALL condition on the specified endpoint.

This function is called by USB device class handler code to set stall condition on endpoint.

Parameters

- `ep` – **[in]** Endpoint address corresponding to the one listed in the device configuration table

Returns

0 on success, negative errno code on fail

int `usb_ep_clear_stall`(uint8_t ep)

Clears STALL condition on the specified endpoint.

This function is called by USB device class handler code to clear stall condition on endpoint.

Parameters

- `ep` – **[in]** Endpoint address corresponding to the one listed in the device configuration table

Returns
0 on success, negative errno code on fail

int **usb_ep_read_wait**(uint8_t ep, uint8_t *data, uint32_t max_data_len, uint32_t *read_bytes)

Read data from the specified endpoint.

This is similar to usb_ep_read, the difference being that, it doesn't clear the endpoint NAKs so that the consumer is not bogged down by further upcalls till he is done with the processing of the data. The caller should reactivate ep by invoking *usb_ep_read_continue()* do so.

Parameters

- **ep** – **[in]** Endpoint address corresponding to the one listed in the device configuration table
- **data** – **[in]** pointer to data buffer to write to
- **max_data_len** – **[in]** max length of data to read
- **read_bytes** – **[out]** Number of bytes read. If data is NULL and max_data_len is 0 the number of bytes available for read should be returned.

Returns
0 on success, negative errno code on fail.

int **usb_ep_read_continue**(uint8_t ep)

Continue reading data from the endpoint.

Clear the endpoint NAK and enable the endpoint to accept more data from the host. Usually called after *usb_ep_read_wait()* when the consumer is fine to accept more data. Thus these calls together acts as flow control mechanism.

Parameters

- **ep** – **[in]** Endpoint address corresponding to the one listed in the device configuration table

Returns
0 on success, negative errno code on fail.

void **usb_transfer_ep_callback**(uint8_t ep, enum *usb_dc_ep_cb_status_code*)

Transfer management endpoint callback.

If a USB class driver wants to use high-level transfer functions, driver needs to register this callback as usb endpoint callback.

int **usb_transfer**(uint8_t ep, uint8_t *data, size_t dlen, unsigned int flags, *usb_transfer_callback* cb, void *priv)

Start a transfer.

Start a usb transfer to/from the data buffer. This function is asynchronous and can be executed in IRQ context. The provided callback will be called on transfer completion (or error) in thread context.

Parameters

- **ep** – **[in]** Endpoint address corresponding to the one listed in the device configuration table
- **data** – **[in]** Pointer to data buffer to write-to/read-from
- **dlen** – **[in]** Size of data buffer
- **flags** – **[in]** Transfer flags (USB_TRANS_READ, USB_TRANS_WRITE...)

- `cb` – **[in]** Function called on transfer completion/failure
- `priv` – **[in]** Data passed back to the transfer completion callback

Returns
0 on success, negative errno code on fail.

int `usb_transfer_sync`(uint8_t ep, uint8_t *data, size_t dlen, unsigned int flags)

Start a transfer and block-wait for completion.

Synchronous version of usb_transfer, wait for transfer completion before returning. A return value of zero can also mean that transfer was cancelled or that the endpoint is not ready. This is due to the design of transfers and usb_dc API.

Parameters

- `ep` – **[in]** Endpoint address corresponding to the one listed in the device configuration table
- `data` – **[in]** Pointer to data buffer to write-to/read-from
- `dlen` – **[in]** Size of data buffer
- `flags` – **[in]** Transfer flags

Returns
number of bytes transferred on success, negative errno code on fail.

void `usb_cancel_transfer`(uint8_t ep)

Cancel any ongoing transfer on the specified endpoint.

Parameters

- `ep` – **[in]** Endpoint address corresponding to the one listed in the device configuration table

void `usb_cancel_transfers`(void)

Cancel all ongoing transfers.

bool `usb_transfer_is_busy`(uint8_t ep)

Check that transfer is ongoing for the endpoint.

Parameters

- `ep` – **[in]** Endpoint address corresponding to the one listed in the device configuration table

Returns
true if transfer is ongoing, false otherwise.

int `usb_wakeup_request`(void)

Start the USB remote wakeup procedure.

Function to request a remote wakeup. This feature must be enabled in configuration, otherwise it will always return -ENOTSUP error.

Returns
0 on success, negative errno code on fail, i.e. when the bus is already active.

bool `usb_get_remote_wakeup_status`(void)

Get status of the USB remote wakeup feature.

Returns
true if remote wakeup has been enabled by the host, false otherwise.

void usb_bos_register_cap(void *hdr)

Register BOS capability descriptor.

This function should be used by the application to register BOS capability descriptors before the USB device stack is enabled.

Parameters

- hdr – **[in]** Pointer to BOS capability descriptor

struct usb_ep_cfg_data

#include <*usb_device.h*> USB Endpoint Configuration.

This structure contains configuration for the endpoint.

Public Members

usb_ep_callback ep_cb

Callback function for notification of data received and available to application or transmit done, NULL if callback not required by application code.

uint8_t ep_addr

The number associated with the EP in the device configuration structure IN EP = 0x80 | <endpoint number> OUT EP = 0x00 | <endpoint number>

struct usb_interface_cfg_data

#include <*usb_device.h*> USB Interface Configuration.

This structure contains USB interface configuration.

Public Members

usb_request_handler class_handler

Handler for USB Class specific Control (EP 0) communications.

usb_request_handler vendor_handler

Handler for USB Vendor specific commands.

usb_request_handler custom_handler

The custom request handler gets a first chance at handling the request before it is handed over to the 'chapter 9' request handler.

return 0 on success, -EINVAL if the request has not been handled by the custom handler and instead needs to be handled by the core USB stack. Any other error code to denote failure within the custom handler.

struct usb_cfg_data

#include <*usb_device.h*> USB device configuration.

The Application instantiates this with given parameters added using the "usb_set_config" function. Once this function is called changes to this structure will result in undefined behavior. This structure may only be updated after calls to usb_deconfig

Public Members

const uint8_t *usb_device_description

USB device description, see http://www.beyondlogic.org/usbnutshell/usb5.shtml#DeviceDescriptors.

void *interface_descriptor

Pointer to interface descriptor.

usb_interface_config interface_config

Function for interface runtime configuration.

void (*cb_usb_status)(struct *usb_cfg_data* *cfg, enum *usb_dc_status_code* cb_status, const uint8_t *param)

Callback to be notified on USB connection status change.

struct *usb_interface_cfg_data* interface

USB interface (Class) handler and storage space.

uint8_t num_endpoints

Number of individual endpoints in the device configuration.

struct *usb_ep_cfg_data* *endpoint

Pointer to an array of endpoint structs of length equal to the number of EP associated with the device description, not including control endpoints.

USB HID Class API

USB device specific part for HID support defined in include/zephyr/usb/class/usb_hid.h.

Related code samples

USB HID (Human Interface Device)
Use USB HID driver to enumerate as a raw HID device.

USB HID mouse
Implement a basic HID mouse device.

API Reference

group usb_hid_device_api

Typedefs

typedef int (*hid_cb_t)(const struct *device* *dev, struct usb_setup_packet *setup, int32_t *len, uint8_t **data)

typedef void (*`hid_int_ready_callback`)(const struct *device* *dev)

typedef void (*`hid_protocol_cb_t`)(const struct *device* *dev, uint8_t protocol)

typedef void (*`hid_idle_cb_t`)(const struct *device* *dev, uint16_t report_id)

Functions

void `usb_hid_register_device`(const struct *device* *dev, const uint8_t *desc, size_t size, const struct *hid_ops* *op)

Register HID device.

Parameters

- `dev` – **[in]** Pointer to USB HID device
- `desc` – **[in]** Pointer to HID report descriptor
- `size` – **[in]** Size of HID report descriptor
- `op` – **[in]** Pointer to USB HID device interrupt struct

int `hid_int_ep_write`(const struct *device* *dev, const uint8_t *data, uint32_t data_len, uint32_t *bytes_ret)

Write to USB HID interrupt endpoint buffer.

Parameters

- `dev` – **[in]** Pointer to USB HID device
- `data` – **[in]** Pointer to data buffer
- `data_len` – **[in]** Length of data to copy
- `bytes_ret` – **[out]** Bytes written to the EP buffer.

Returns

0 on success, negative errno code on fail.

int `hid_int_ep_read`(const struct *device* *dev, uint8_t *data, uint32_t max_data_len, uint32_t *ret_bytes)

Read from USB HID interrupt endpoint buffer.

Parameters

- `dev` – **[in]** Pointer to USB HID device
- `data` – **[in]** Pointer to data buffer
- `max_data_len` – **[in]** Max length of data to copy
- `ret_bytes` – **[out]** Number of bytes to copy. If data is NULL and max_data_len is 0 the number of bytes available in the buffer will be returned.

Returns

0 on success, negative errno code on fail.

int `usb_hid_set_proto_code`(const struct *device* *dev, uint8_t proto_code)

Set USB HID class Protocol Code.

Should be called before *usb_hid_init()*.

Parameters

- `dev` – **[in]** Pointer to USB HID device
- `proto_code` – **[in]** Protocol Code to be used for bInterfaceProtocol

Returns
0 on success, negative errno code on fail.

int `usb_hid_init`(const struct *device* *dev)
Initialize USB HID class support.

Parameters

- `dev` – **[in]** Pointer to USB HID device

Returns
0 on success, negative errno code on fail.

struct `hid_ops`
#include <*usb_hid.h*> USB HID device interface.

Binary Device Object Store (BOS) support API

API reference

group `usb_bos`
USB Binary Device Object Store support.

Enums

enum `usb_bos_capability_types`
Device capability type codes.

Values:

enumerator `USB_BOS_CAPABILITY_EXTENSION` = 0x02

enumerator `USB_BOS_CAPABILITY_PLATFORM` = 0x05

struct `usb_bos_descriptor`
#include <*bos.h*> Root BOS Descriptor.

struct `usb_bos_capability_lpm`
#include <*bos.h*> BOS USB 2.0 extension capability descriptor.

struct `usb_bos_platform_descriptor`
#include <*bos.h*> BOS platform capability descriptor.

struct `usb_bos_capability_webusb`
#include <*bos.h*> WebUSB specific part of platform capability descriptor.

struct `usb_bos_capability_msos`
#include <*bos.h*> Microsoft OS 2.0 descriptor specific part of platform capability descriptor.

New experimental USB support

6.5.3 New USB device support

Overview

USB device support consists of the USB device controller (UDC) drivers , *USB device controller (UDC) driver API*, and USB device stack, *USB device stack (next) API*. The *USB device controller (UDC) driver API* provides a generic and vendor independent interface to USB device controllers, and although, there a is clear separation between these layers, the purpose of *USB device controller (UDC) driver API* is to serve new Zephyr's USB device stack exclusively.

The new device stack supports multiple device controllers, meaning that if a SoC has multiple controllers, they can be used simultaneously. Full and high-speed device controllers are supported. It also provides support for registering multiple function or class instances to a configuration at runtime, or changing the configuration later. It has built-in support for several USB classes and provides an API to implement custom USB functions.

The new USB device support is considered experimental and will replace *USB device support*.

Built-in functions The USB device stack has built-in USB functions. Some can be used directly in the user application through a special API, such as HID or Audio class devices, while others use a general Zephyr RTOS driver API, such as MSC and CDC class implementations. The *Identification string* identifies a class or function instance (*n*) and is used as an argument to the `usbd_register_class()`.

Class or function	User API (if any)	Identification string
USB Audio 2 class	*Audio Class 2 device API*	`uac2_n`
USB CDC ACM class	*Universal Asynchronous Receiver-Transmitter (UART)*	`cdc_acm_n`
USB CDC ECM class	Ethernet device	`cdc_ecm_n`
USB Mass Storage Class (MSC)	*USB Mass Storage Class device API*	`msc_n`
USB Human Interface Devices (HID)	*HID device API*	`hid_n`
Bluetooth HCI USB transport layer	*HCI RAW channel*	`bt_hci_n`

Samples

- usb-hid-keyboard
- uac2-explicit-feedback

Samples ported to new USB device support To build a sample that supports both the old and new USB device stack, set the configuration `-DCONF_FILE=usbd_next_prj.conf` either directly or via `west`.

- bluetooth-hci-usb-sample
- usb-cdc-acm
- usb-cdc-acm-console
- usb-mass
- usb-hid-mouse
- zperf To build the sample for the new device support, set the configuration overlay file `-DDEXTRA_CONF_FILE=overlay-usbd_next_ecm.conf` and devicetree overlay file `-DDTC_OVERLAY_FILE="usbd_next_ecm.overlay` either directly or via `west`.

How to configure and enable USB device support

For the USB device support samples in the Zephyr project repository, we have a common file for instantiation, configuration and initialization, samples/subsys/usb/common/sample_usbd_init.c. The following code snippets from this file are used as examples. USB Samples Kconfig options used in the USB samples and prefixed with SAMPLE_USBD_ have default values specific to the Zephyr project and the scope is limited to the project samples. In the examples below, you will need to replace these Kconfig options and other defaults with values appropriate for your application or hardware.

The USB device stack requires a context structure to manage its properties and runtime data. The preferred way to define a device context is to use the *USBD_DEVICE_DEFINE* macro. This creates a static *usbd_context* variable with a given name. Any number of contexts may be instantiated. A USB controller device can be assigned to multiple contexts, but only one context can be initialized and used at a time. Context properties must not be directly accessed or manipulated by the application.

```
/*
 * Instantiate a context named sample_usbd using the default USB device
 * controller, the Zephyr project vendor ID, and the sample product ID.
 * Zephyr project vendor ID must not be used outside of Zephyr samples.
 */
USBD_DEVICE_DEFINE(sample_usbd,
                   DEVICE_DT_GET(DT_NODELABEL(zephyr_udc0)),
                   ZEPHYR_PROJECT_USB_VID, CONFIG_SAMPLE_USBD_PID);
```

Your USB device may have manufacturer, product, and serial number string descriptors. To instantiate these string descriptors, the application should use the appropriate *USBD_DESC_MANUFACTURER_DEFINE*, *USBD_DESC_PRODUCT_DEFINE*, and *USBD_DESC_SERIAL_NUMBER_DEFINE* macros. String descriptors also require a single instantiation of the language descriptor using the *USBD_DESC_LANG_DEFINE* macro.

```
USBD_DESC_LANG_DEFINE(sample_lang);
USBD_DESC_MANUFACTURER_DEFINE(sample_mfr, CONFIG_SAMPLE_USBD_MANUFACTURER);
USBD_DESC_PRODUCT_DEFINE(sample_product, CONFIG_SAMPLE_USBD_PRODUCT);
USBD_DESC_SERIAL_NUMBER_DEFINE(sample_sn);
```

String descriptors must be added to the device context at runtime before initializing the USB device with *usbd_add_descriptor()*.

```
err = usbd_add_descriptor(&sample_usbd, &sample_lang);
if (err) {
        LOG_ERR("Failed to initialize language descriptor (%d)", err);
        return NULL;
}

err = usbd_add_descriptor(&sample_usbd, &sample_mfr);
if (err) {
        LOG_ERR("Failed to initialize manufacturer descriptor (%d)", err);
        return NULL;
}

err = usbd_add_descriptor(&sample_usbd, &sample_product);
if (err) {
        LOG_ERR("Failed to initialize product descriptor (%d)", err);
        return NULL;
}

err = usbd_add_descriptor(&sample_usbd, &sample_sn);
if (err) {
        LOG_ERR("Failed to initialize SN descriptor (%d)", err);
```

(continues on next page)

(continued from previous page)

```
        return NULL;
}
```

USB device requires at least one configuration instance per supported speed. The application should use *USBD_CONFIGURATION_DEFINE* to instantiate a configuration. Later, USB device functions are assigned to a configuration.

```
static const uint8_t attributes = (IS_ENABLED(CONFIG_SAMPLE_USBD_SELF_POWERED) ?
                                   USB_SCD_SELF_POWERED : 0) |
                                  (IS_ENABLED(CONFIG_SAMPLE_USBD_REMOTE_WAKEUP) ?
                                   USB_SCD_REMOTE_WAKEUP : 0);

/* Full speed configuration */
USBD_CONFIGURATION_DEFINE(sample_fs_config,
                          attributes,
                          CONFIG_SAMPLE_USBD_MAX_POWER);

/* High speed configuration */
USBD_CONFIGURATION_DEFINE(sample_hs_config,
                          attributes,
                          CONFIG_SAMPLE_USBD_MAX_POWER);
```

Each configuration instance for a specific speed must be added to the device context at runtime before the USB device is initialized using *usbd_add_configuration()*. Note *USBD_SPEED_FS* and *USBD_SPEED_HS*. The first full-speed or high-speed configuration will get `bConfigurationValue` one, and then further upward.

```
err = usbd_add_configuration(&sample_usbd, USBD_SPEED_FS,
                             &sample_fs_config);
if (err) {
        LOG_ERR("Failed to add Full-Speed configuration");
        return NULL;
}
```

Although we have already done a lot, this USB device has no function. A device can have multiple configurations with different set of functions at different speeds. A function or class can be registered on a USB device before it is initialized using *usbd_register_class()*. The desired configuration is specified using *USBD_SPEED_FS* or *USBD_SPEED_HS* and the configuration number. For simple cases, *usbd_register_all_classes()* can be used to register all available instances.

```
err = usbd_register_all_classes(&sample_usbd, USBD_SPEED_FS, 1);
if (err) {
        LOG_ERR("Failed to add register classes");
        return NULL;
}
```

The last step in the preparation is to initialize the device with *usbd_init()*. After this, the configuration of the device cannot be changed. A device can be deinitialized with *usbd_shutdown()* and all instances can be reused, but the previous steps must be repeated. So it is possible to shutdown a device, register another type of configuration or function, and initialize it again. At the USB controller level, *usbd_init()* does only what is necessary to detect VBUS changes. There are controller types where the next step is only possible if a VBUS signal is present.

A function or class implementation may require its own specific configuration steps, which should be performed prior to initializing the USB device.

```
err = usbd_init(&sample_usbd);
if (err) {
        LOG_ERR("Failed to initialize device support");
        return NULL;
}
```

The final step to enable the USB device is *usbd_enable()*, after that, if the USB device is connected to a USB host controller, the host can start enumerating the device. The application can disable the USB device using *usbd_disable()*.

```
ret = usbd_enable(sample_usbd);
if (ret) {
        LOG_ERR("Failed to enable device support");
        return ret;
}
```

USB Message notifications The application can register a callback using *usbd_msg_register_cb()* to receive message notification from the USB device support subsystem. The messages are mostly about the common device state changes, and a few specific types from the USB CDC ACM implementation.

```
err = usbd_msg_register_cb(&sample_usbd, msg_cb);
if (err) {
        LOG_ERR("Failed to register message callback");
        return NULL;
}
```

The helper function *usbd_msg_type_string()* can be used to convert *usbd_msg_type* to a human readable form for logging.

If the controller supports VBUS state change detection, the battery-powered application may want to enable the USB device only when it is connected to a host. A generic application should use *usbd_can_detect_vbus()* to check for this capability.

```
static void msg_cb(struct usbd_context *const usbd_ctx,
                   const struct usbd_msg *const msg)
{
        LOG_INF("USBD message: %s", usbd_msg_type_string(msg->type));

        if (usbd_can_detect_vbus(usbd_ctx)) {
                if (msg->type == USBD_MSG_VBUS_READY) {
                        if (usbd_enable(usbd_ctx)) {
                                LOG_ERR("Failed to enable device support");
                        }
                }

                if (msg->type == USBD_MSG_VBUS_REMOVED) {
                        if (usbd_disable(usbd_ctx)) {
                                LOG_ERR("Failed to disable device support");
                        }
                }
        }
}
```

6.5.4 New USB device support APIs

USB device controller (UDC) driver API

The USB device controller driver API is described in include/zephyr/drivers/usb/udc.h and referred to as the `UDC driver` API.

UDC driver API is experimental and is subject to change without notice. It is a replacement for *USB device controller driver API*. If you wish to port an existing driver to UDC driver API, or add a new driver, please use drivers/usb/udc/udc_skeleton.c as a starting point.

API reference

group `udc_api`

New USB device controller (UDC) driver API.

Functions

static inline bool `udc_is_initialized`(const struct *device* *dev)

Checks whether the controller is initialized.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance

Returns

true if controller is initialized, false otherwise

static inline bool `udc_is_enabled`(const struct *device* *dev)

Checks whether the controller is enabled.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance

Returns

true if controller is enabled, false otherwise

static inline bool `udc_is_suspended`(const struct *device* *dev)

Checks whether the controller is suspended.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance

Returns

true if controller is suspended, false otherwise

int `udc_init`(const struct *device* *dev, udc_event_cb_t event_cb)

Initialize USB device controller.

Initialize USB device controller and control IN/OUT endpoint. After initialization controller driver should be able to detect power state of the bus and signal power state changes.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance
- `event_cb` – **[in]** Event callback from the higher layer (USB device stack)

Return values

- `-EINVAL` – on parameter error (no callback is passed)
- `-EALREADY` – already initialized

Returns

0 on success, all other values should be treated as error.

int `udc_enable`(const struct *device* *dev)

Enable USB device controller.

Enable powered USB device controller and allow host to recognize and enumerate the device.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance

Return values

- `-EPERM` – controller is not initialized
- `-EALREADY` – already enabled

Returns
0 on success, all other values should be treated as error.

int `udc_disable`(const struct *device* *dev)

Disable USB device controller.

Disable enabled USB device controller. The driver should continue to detect power state changes.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance

Return values
`-EALREADY` – already disabled

Returns
0 on success, all other values should be treated as error.

int `udc_shutdown`(const struct *device* *dev)

Poweroff USB device controller.

Shut down the controller completely to reduce energy consumption or to change the role of the controller.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance

Return values
`-EALREADY` – controller is not initialized

Returns
0 on success, all other values should be treated as error.

static inline struct udc_device_caps `udc_caps`(const struct *device* *dev)

Get USB device controller capabilities.

Obtain the capabilities of the controller such as full speed (FS), high speed (HS), and more.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance

Returns
USB device controller capabilities.

enum udc_bus_speed `udc_device_speed`(const struct *device* *dev)

Get actual USB device speed.

The function should be called after the reset event to determine the actual bus speed.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance

Returns
USB device controller capabilities.

static inline int `udc_set_address`(const struct *device* *dev, const uint8_t addr)

Set USB device address.

Set address of enabled USB device.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance
- `addr` – **[in]** USB device address

Return values

`-EPERM` – controller is not enabled (or not initialized)

Returns

0 on success, all other values should be treated as error.

static inline int `udc_test_mode`(const struct *device* *dev, const uint8_t mode, const bool dryrun)

Enable Test Mode.

For compliance testing, high-speed controllers must support test modes. A particular test is enabled by a SetFeature(TEST_MODE) request. To disable a test mode, device needs to be power cycled.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance
- `mode` – **[in]** Test mode
- `dryrun` – **[in]** Verify that a particular mode can be enabled, but do not enable test mode

Return values

`-ENOTSUP` – Test mode is not supported

Returns

0 on success, all other values should be treated as error.

static inline int `udc_host_wakeup`(const struct *device* *dev)

Initiate host wakeup procedure.

Initiate host wakeup. Only possible when the bus is suspended.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance

Return values

`-EPERM` – controller is not enabled (or not initialized)

Returns

0 on success, all other values should be treated as error.

int `udc_ep_try_config`(const struct *device* *dev, const uint8_t ep, const uint8_t attributes, uint16_t *const mps, const uint8_t interval)

Try an endpoint configuration.

Try an endpoint configuration based on endpoint descriptor. This function may modify wMaxPacketSize descriptor fields of the endpoint. All properties of the descriptor, such as direction, and transfer type, should be set correctly. If wMaxPacketSize value is zero, it will be updated to maximum buffer size of the endpoint.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance
- `ep` – **[in]** Endpoint address (same as bEndpointAddress)
- `attributes` – **[in]** Endpoint attributes (same as bmAttributes)
- `mps` – **[in]** Maximum packet size (same as wMaxPacketSize)
- `interval` – **[in]** Polling interval (same as bInterval)

Return values

- `-EINVAL` – on wrong parameter
- `-ENOTSUP` – endpoint configuration not supported
- `-ENODEV` – no endpoints available

Returns
0 on success, all other values should be treated as error.

int `udc_ep_enable`(const struct *device* *dev, const uint8_t ep, const uint8_t attributes, const uint16_t mps, const uint8_t interval)

Configure and enable endpoint.

Configure and make an endpoint ready for use. Valid for all endpoints except control IN/OUT.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance
- `ep` – **[in]** Endpoint address (same as bEndpointAddress)
- `attributes` – **[in]** Endpoint attributes (same as bmAttributes)
- `mps` – **[in]** Maximum packet size (same as wMaxPacketSize)
- `interval` – **[in]** Polling interval (same as bInterval)

Return values

- `-EINVAL` – on wrong parameter (control IN/OUT endpoint)
- `-EPERM` – controller is not initialized
- `-ENODEV` – endpoint configuration not found
- `-EALREADY` – endpoint is already enabled

Returns
0 on success, all other values should be treated as error.

int `udc_ep_disable`(const struct *device* *dev, const uint8_t ep)

Disable endpoint.

Valid for all endpoints except control IN/OUT.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance
- `ep` – **[in]** Endpoint address

Return values

- `-EINVAL` – on wrong parameter (control IN/OUT endpoint)
- `-ENODEV` – endpoint configuration not found
- `-EALREADY` – endpoint is already disabled
- `-EPERM` – controller is not initialized

Returns
0 on success, all other values should be treated as error.

int `udc_ep_set_halt`(const struct *device* *dev, const uint8_t ep)

Halt endpoint.

Valid for all endpoints.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance
- `ep` – **[in]** Endpoint address

Return values

- `-ENODEV` – endpoint configuration not found
- `-ENOTSUP` – not supported (e.g. isochronous endpoint)
- `-EPERM` – controller is not enabled

Returns
0 on success, all other values should be treated as error.

int `udc_ep_clear_halt`(const struct *device* *dev, const uint8_t ep)

Clear endpoint halt.

Valid for all endpoints.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance
- `ep` – **[in]** Endpoint address

Return values

- `-ENODEV` – endpoint configuration not found
- `-ENOTSUP` – not supported (e.g. isochronous endpoint)
- `-EPERM` – controller is not enabled

Returns
0 on success, all other values should be treated as error.

int `udc_ep_enqueue`(const struct *device* *dev, struct *net_buf* *const buf)

Queue USB device controller request.

Add request to the queue. If the queue is empty, the request buffer can be claimed by the controller immediately.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance
- `buf` – **[in]** Pointer to UDC request buffer

Return values

- `-ENODEV` – endpoint configuration not found
- `-EACCES` – endpoint is not enabled (TBD)
- `-EBUSY` – request can not be queued
- `-EPERM` – controller is not initialized

Returns
0 on success, all other values should be treated as error.

int `udc_ep_dequeue`(const struct *device* *dev, const uint8_t ep)

Remove all USB device controller requests from endpoint queue.

UDC_EVT_EP_REQUEST event will be generated when the driver releases claimed buffer, no new requests will be claimed, all requests in the queue will passed as chained list of the event variable buf. The endpoint queue is empty after that.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance

- `ep` – **[in]** Endpoint address

Return values

- `-ENODEV` – endpoint configuration not found
- `-EACCES` – endpoint is not disabled
- `-EPERM` – controller is not initialized

Returns
0 on success, all other values should be treated as error.

struct *net_buf* *`udc_ep_buf_alloc`(const struct *device* *dev, const uint8_t ep, const size_t size)

Allocate UDC request buffer.

Allocate a new buffer from common request buffer pool.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance
- `ep` – **[in]** Endpoint address
- `size` – **[in]** Size of the request buffer

Returns
pointer to allocated request or NULL on error.

int `udc_ep_buf_free`(const struct *device* *dev, struct *net_buf* *const buf)

Free UDC request buffer.

Put the buffer back into the request buffer pool.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance
- `buf` – **[in]** Pointer to UDC request buffer

Returns
0 on success, all other values should be treated as error.

static inline void `udc_ep_buf_set_zlp`(struct *net_buf* *const buf)

Set ZLP flag in requests metadata.

The controller should send a ZLP at the end of the transfer.

Parameters

- `buf` – **[in]** Pointer to UDC request buffer

static inline struct udc_buf_info *`udc_get_buf_info`(const struct *net_buf* *const buf)

Get requests metadata.

Parameters

- `buf` – **[in]** Pointer to UDC request buffer

Returns
pointer to metadata structure.

USB device stack (next) API

New USB device stack API is experimental and is subject to change without notice.

Related code samples

Console over USB CDC ACM
Output ”Hello World!” to the console over USB CDC ACM.

USB Audio asynchronous explicit feedback sample
USB Audio 2 explicit feedback sample playing audio on I2S.

USB CDC-ACM
Use USB CDC-ACM driver to implement a serial port echo.

USB HID keyboard
Implement a basic HID keyboard device.

USB Mass Storage
Expose board’s RAM or FLASH as a USB disk using USB Mass Storage driver.

USB shell
Use shell commands to interact with USB device stack.

API reference

group `usbd_api`

New USB device stack core API.

Defines

`USB_BSTRING_LENGTH`(s)

`USB_STRING_DESCRIPTOR_LENGTH`(s)

`USBD_NUMOF_INTERFACES_MAX`

`USBD_CCTX_REGISTERED`

USB Class instance registered flag.

`USBD_DEVICE_DEFINE`(device_name, udc_dev, vid, pid)

Define USB device context structure.

Macro defines a USB device structure needed by the stack to manage its properties and runtime data. The `vid` and `pid` parameters can also be changed using *usbd_device_set_vid()* and *usbd_device_set_pid()*.

Example of use:

```
USBD_DEVICE_DEFINE(sample_usbd,
                   DEVICE_DT_GET(DT_NODELABEL(zephyr_udc0)),
                   YOUR_VID, YOUR_PID);
```

Parameters

- `device_name` – USB device context name
- `udc_dev` – Pointer to UDC device structure
- `vid` – Vendor ID
- `pid` – Product ID

USBD_CONFIGURATION_DEFINE(name, attrib, power)

Define USB device configuration.

USB device requires at least one configuration instance per supported speed. attrib is a combination of USB_SCD_SELF_POWERED or USB_SCD_REMOTE_WAKEUP, depending on which characteristic the USB device should have in this configuration.

Parameters

- name – Configuration name
- attrib – Configuration characteristics. Attributes can also be updated with *usbd_config_attrib_rwup()* and *usbd_config_attrib_self()*
- power – bMaxPower value in 2 mA units. This value can also be set with *usbd_config_maxpower()*

USBD_DESC_LANG_DEFINE(name)

Create a string descriptor node and language string descriptor.

This macro defines a descriptor node and a string descriptor that, when added to the device context, is automatically used as the language string descriptor zero. Both descriptor node and descriptor are defined with static-storage-class specifier. Default and currently only supported language ID is 0x0409 English (United States). If string descriptors are used, it is necessary to add this descriptor as the first one to the USB device context.

Parameters

- name – Language string descriptor node identifier.

USBD_DESC_STRING_DEFINE(d_name, d_string, d_utype)

Create a string descriptor.

This macro defines a descriptor node and a string descriptor. The string literal passed to the macro should be in the ASCII7 format. It is converted to UTF16LE format on the host request.

Parameters

- d_name – Internal string descriptor node identifier name
- d_string – ASCII7 encoded string literal
- d_utype – String descriptor usage type

USBD_DESC_MANUFACTURER_DEFINE(d_name, d_string)

Create a string descriptor node and manufacturer string descriptor.

This macro defines a descriptor node and a string descriptor that, when added to the device context, is automatically used as the manufacturer string descriptor. Both descriptor node and descriptor are defined with static-storage-class specifier.

Parameters

- d_name – String descriptor node identifier.
- d_string – ASCII7 encoded manufacturer string literal

USBD_DESC_PRODUCT_DEFINE(d_name, d_string)

Create a string descriptor node and product string descriptor.

This macro defines a descriptor node and a string descriptor that, when added to the device context, is automatically used as the product string descriptor. Both descriptor node and descriptor are defined with static-storage-class specifier.

Parameters

- d_name – String descriptor node identifier.

- **d_string** – ASCII7 encoded product string literal

USBD_DESC_SERIAL_NUMBER_DEFINE(d_name)

Create a string descriptor node and serial number string descriptor.

This macro defines a descriptor node that, when added to the device context, is automatically used as the serial number string descriptor. A valid serial number is generated from HWID (HWINFO= whenever this string descriptor is requested.

Parameters

- **d_name** – String descriptor node identifier.

USBD_DESC_BOS_DEFINE(name, len, subset)

Define BOS Device Capability descriptor node.

The application defines a BOS capability descriptor node for descriptors such as USB 2.0 Extension Descriptor.

Parameters

- **name** – Descriptor node identifier
- **len** – Device Capability descriptor length
- **subset** – Pointer to a Device Capability descriptor

USBD_DEFINE_CLASS(class_name, class_api, class_priv, class_v_reqs)

Define USB device support class data.

Macro defines class (function) data, as well as corresponding node structures used internally by the stack.

Parameters

- **class_name** – Class name
- **class_api** – Pointer to struct *usbd_class_api*
- **class_priv** – Class private data
- **class_v_reqs** – Pointer to struct *usbd_cctx_vendor_req*

VENDOR_REQ_DEFINE(_reqs, _len)

Helper to declare request table of *usbd_cctx_vendor_req*.

Parameters

- **_reqs** – Pointer to the vendor request field
- **_len** – Number of supported vendor requests

USBD_VENDOR_REQ(_reqs...)

Helper to declare supported vendor requests.

Parameters

- **_reqs** – Variable number of vendor requests

Typedefs

typedef void (***usbd_msg_cb_t**)(struct *usbd_context* *const ctx, const struct *usbd_msg* *const msg)

Callback type definition for USB device message delivery.

The implementation uses the system workqueue, and a callback provided and registered by the application. The application callback is called in the context of the system

workqueue. Notification messages are stored in a queue and delivered to the callback in sequence.

Param ctx
[in] Pointer to USB device support context

Param msg
[in] Pointer to USB device message

Enums

enum `usbd_ch9_state`

USB device support middle layer runtime state.

Part of USB device states without suspended and powered states, as it is better to track them separately.

Values:

enumerator `USBD_STATE_DEFAULT` = 0

enumerator `USBD_STATE_ADDRESS`

enumerator `USBD_STATE_CONFIGURED`

enum `usbd_speed`

USB device speed.

Values:

enumerator `USBD_SPEED_FS`

Device supports or is connected to a full speed bus.

enumerator `USBD_SPEED_HS`

Device supports or is connected to a high speed bus

enumerator `USBD_SPEED_SS`

Device supports or is connected to a super speed bus.

Functions

static inline struct *usbd_context* *`usbd_class_get_ctx`(const struct *usbd_class_data* *const c_data)

Get the USB device runtime context under which the class is registered.

The class implementation must use this function and not access the members of the struct directly.

Parameters

- `c_data` – **[in]** Pointer to USB device class data

Returns
Pointer to USB device runtime context

static inline void *usbd_class_get_private(const struct *usbd_class_data* *const c_data)

Get class implementation private data.

The class implementation must use this function and not access the members of the struct directly.

Parameters

- **c_data** – **[in]** Pointer to USB device class data

Returns

Pointer to class implementation private data

int usbd_add_descriptor(struct *usbd_context* *uds_ctx, struct *usbd_desc_node* *dn)

Add common USB descriptor.

Add common descriptor like string or BOS Device Capability.

Parameters

- **uds_ctx** – **[in]** Pointer to USB device support context
- **dn** – **[in]** Pointer to USB descriptor node

Returns

0 on success, other values on fail.

uint8_t usbd_str_desc_get_idx(const struct *usbd_desc_node* *const desc_nd)

Get USB string descriptor index from descriptor node.

Parameters

- **desc_nd** – **[in]** Pointer to USB descriptor node

Returns

Descriptor index, 0 if descriptor is not part of any device

void usbd_remove_descriptor(struct *usbd_desc_node* *const desc_nd)

Remove USB string descriptor.

Remove linked USB string descriptor from any list.

Parameters

- **desc_nd** – **[in]** Pointer to USB descriptor node

int usbd_add_configuration(struct *usbd_context* *uds_ctx, const enum *usbd_speed* speed, struct *usbd_config_node* *cd)

Add a USB device configuration.

Parameters

- **uds_ctx** – **[in]** Pointer to USB device support context
- **speed** – **[in]** Speed at which this configuration operates
- **cd** – **[in]** Pointer to USB configuration node

Returns

0 on success, other values on fail.

int usbd_register_class(struct *usbd_context* *uds_ctx, const char *name, const enum *usbd_speed* speed, uint8_t cfg)

Register an USB class instance.

An USB class implementation can have one or more instances. To identify the instances we use device drivers API. Device names have a prefix derived from the name of the class, for example CDC_ACM for CDC ACM class instance, and can also be easily identified in the shell. Class instance can only be registered when the USB device stack is

disabled. Registered instances are initialized at initialization of the USB device stack, and the interface descriptors of each instance are adapted to the whole context.

Parameters

- `uds_ctx` – **[in]** Pointer to USB device support context
- `name` – **[in]** Class instance name
- `speed` – **[in]** Configuration speed
- `cfg` – **[in]** Configuration value (bConfigurationValue)

Returns
0 on success, other values on fail.

int `usbd_register_all_classes`(struct *usbd_context* *uds_ctx, const enum *usbd_speed* speed, uint8_t cfg)

Register all available USB class instances.

Register all available instances. Like usbd_register_class, but does not take the instance name and instead registers all available instances.

Note

This cannot be combined. If your application calls usbd_register_class for any device, configuration number, or instance, either usbd_register_class or this function will fail.

Parameters

- `uds_ctx` – **[in]** Pointer to USB device support context
- `speed` – **[in]** Configuration speed
- `cfg` – **[in]** Configuration value (bConfigurationValue)

Returns
0 on success, other values on fail.

int `usbd_unregister_class`(struct *usbd_context* *uds_ctx, const char *name, const enum *usbd_speed* speed, uint8_t cfg)

Unregister an USB class instance.

USB class instance will be removed and will not appear on the next start of the stack. Instance can only be unregistered when the USB device stack is disabled.

Parameters

- `uds_ctx` – **[in]** Pointer to USB device support context
- `name` – **[in]** Class instance name
- `speed` – **[in]** Configuration speed
- `cfg` – **[in]** Configuration value (bConfigurationValue)

Returns
0 on success, other values on fail.

int `usbd_unregister_all_classes`(struct *usbd_context* *uds_ctx, const enum *usbd_speed* speed, uint8_t cfg)

Unregister all available USB class instances.

Unregister all available instances. Like usbd_unregister_class, but does not take the instance name and instead unregisters all available instances.

Parameters

- `uds_ctx` – **[in]** Pointer to USB device support context
- `speed` – **[in]** Configuration speed
- `cfg` – **[in]** Configuration value (bConfigurationValue)

Returns
0 on success, other values on fail.

int `usbd_msg_register_cb`(struct *usbd_context* *const uds_ctx, const *usbd_msg_cb_t* cb)

Register USB notification message callback.

Parameters

- `uds_ctx` – **[in]** Pointer to USB device support context
- `cb` – **[in]** Pointer to message callback function

Returns
0 on success, other values on fail.

int `usbd_init`(struct *usbd_context* *uds_ctx)

Initialize USB device.

Initialize USB device descriptors and configuration, initialize USB device controller. Class instances should be registered before they are involved. However, the stack should also initialize without registered instances, even if the host would complain about missing interfaces.

Parameters

- `uds_ctx` – **[in]** Pointer to USB device support context

Returns
0 on success, other values on fail.

int `usbd_enable`(struct *usbd_context* *uds_ctx)

Enable the USB device support and registered class instances.

This function enables the USB device support.

Parameters

- `uds_ctx` – **[in]** Pointer to USB device support context

Returns
0 on success, other values on fail.

int `usbd_disable`(struct *usbd_context* *uds_ctx)

Disable the USB device support.

This function disables the USB device support.

Parameters

- `uds_ctx` – **[in]** Pointer to USB device support context

Returns
0 on success, other values on fail.

int `usbd_shutdown`(struct *usbd_context* *const uds_ctx)

Shutdown the USB device support.

This function completely disables the USB device support.

Parameters

- `uds_ctx` – **[in]** Pointer to USB device support context

Returns
0 on success, other values on fail.

int **usbd_ep_set_halt**(struct *usbd_context* *uds_ctx, uint8_t ep)
Halt endpoint.

Parameters
- **uds_ctx** – **[in]** Pointer to USB device support context
- **ep** – **[in]** Endpoint address

Returns
0 on success, or error from *udc_ep_set_halt()*

int **usbd_ep_clear_halt**(struct *usbd_context* *uds_ctx, uint8_t ep)
Clear endpoint halt.

Parameters
- **uds_ctx** – **[in]** Pointer to USB device support context
- **ep** – **[in]** Endpoint address

Returns
0 on success, or error from *udc_ep_clear_halt()*

bool **usbd_ep_is_halted**(struct *usbd_context* *uds_ctx, uint8_t ep)
Checks whether the endpoint is halted.

Parameters
- **uds_ctx** – **[in]** Pointer to USB device support context
- **ep** – **[in]** Endpoint address

Returns
true if endpoint is halted, false otherwise

struct *net_buf* ***usbd_ep_buf_alloc**(const struct *usbd_class_data* *const c_data, const uint8_t ep, const size_t size)
Allocate buffer for USB device request.

Allocate a new buffer from controller's driver buffer pool.

Parameters
- **c_data** – **[in]** Pointer to USB device class data
- **ep** – **[in]** Endpoint address
- **size** – **[in]** Size of the request buffer

Returns
pointer to allocated request or NULL on error.

int **usbd_ep_ctrl_enqueue**(struct *usbd_context* *const uds_ctx, struct *net_buf* *const buf)
Queue USB device control request.

Add control request to the queue.

Parameters
- **uds_ctx** – **[in]** Pointer to USB device support context
- **buf** – **[in]** Pointer to UDC request buffer

Returns
0 on success, all other values should be treated as error.

int **usbd_ep_enqueue**(const struct *usbd_class_data* *const c_data, struct *net_buf* *const buf)

Queue USB device request.

Add request to the queue.

Parameters

- **c_data** – **[in]** Pointer to USB device class data
- **buf** – **[in]** Pointer to UDC request buffer

Returns
0 on success, or error from *udc_ep_enqueue()*

int **usbd_ep_dequeue**(struct *usbd_context* *uds_ctx, const uint8_t ep)

Remove all USB device controller requests from endpoint queue.

Parameters

- **uds_ctx** – **[in]** Pointer to USB device support context
- **ep** – **[in]** Endpoint address

Returns
0 on success, or error from *udc_ep_dequeue()*

int **usbd_ep_buf_free**(struct *usbd_context* *uds_ctx, struct *net_buf* *buf)

Free USB device request buffer.

Put the buffer back into the request buffer pool.

Parameters

- **uds_ctx** – **[in]** Pointer to USB device support context
- **buf** – **[in]** Pointer to UDC request buffer

Returns
0 on success, all other values should be treated as error.

bool **usbd_is_suspended**(struct *usbd_context* *uds_ctx)

Checks whether the USB device controller is suspended.

Parameters

- **uds_ctx** – **[in]** Pointer to USB device support context

Returns
true if endpoint is halted, false otherwise

int **usbd_wakeup_request**(struct *usbd_context* *uds_ctx)

Initiate the USB remote wakeup (TBD)

Returns
0 on success, other values on fail.

enum *usbd_speed* **usbd_bus_speed**(const struct *usbd_context* *const uds_ctx)

Get actual device speed.

Parameters

- **uds_ctx** – **[in]** Pointer to a device context

Returns
Actual device speed

enum *usbd_speed* **usbd_caps_speed**(const struct *usbd_context* *const uds_ctx)

Get highest speed supported by the controller.

Parameters

- **uds_ctx** – **[in]** Pointer to a device context

Returns

Highest supported speed

int **usbd_device_set_bcd**(struct *usbd_context* *const uds_ctx, const enum *usbd_speed* speed, const uint16_t bcd)

Set USB device descriptor value bcdUSB.

Parameters

- **uds_ctx** – **[in]** Pointer to USB device support context
- **speed** – **[in]** Speed for which the bcdUSB should be set
- **bcd** – **[in]** bcdUSB value

Returns

0 on success, other values on fail.

int **usbd_device_set_vid**(struct *usbd_context* *const uds_ctx, const uint16_t vid)

Set USB device descriptor value idVendor.

Parameters

- **uds_ctx** – **[in]** Pointer to USB device support context
- **vid** – **[in]** idVendor value

Returns

0 on success, other values on fail.

int **usbd_device_set_pid**(struct *usbd_context* *const uds_ctx, const uint16_t pid)

Set USB device descriptor value idProduct.

Parameters

- **uds_ctx** – **[in]** Pointer to USB device support context
- **pid** – **[in]** idProduct value

Returns

0 on success, other values on fail.

int **usbd_device_set_code_triple**(struct *usbd_context* *const uds_ctx, const enum *usbd_speed* speed, const uint8_t base_class, const uint8_t subclass, const uint8_t protocol)

Set USB device descriptor code triple Base Class, SubClass, and Protocol.

Parameters

- **uds_ctx** – **[in]** Pointer to USB device support context
- **speed** – **[in]** Speed for which the code triple should be set
- **base_class** – **[in]** bDeviceClass value
- **subclass** – **[in]** bDeviceSubClass value
- **protocol** – **[in]** bDeviceProtocol value

Returns

0 on success, other values on fail.

int **usbd_config_attrib_rwup**(struct *usbd_context* *const uds_ctx, const enum *usbd_speed* speed, const uint8_t cfg, const bool enable)

Setup USB device configuration attribute Remote Wakeup.

Parameters

- **uds_ctx** – **[in]** Pointer to USB device support context
- **speed** – **[in]** Configuration speed
- **cfg** – **[in]** Configuration number
- **enable** – **[in]** Sets attribute if true, clears it otherwise

Returns
0 on success, other values on fail.

int **usbd_config_attrib_self**(struct *usbd_context* *const uds_ctx, const enum *usbd_speed* speed, const uint8_t cfg, const bool enable)

Setup USB device configuration attribute Self-powered.

Parameters

- **uds_ctx** – **[in]** Pointer to USB device support context
- **speed** – **[in]** Configuration speed
- **cfg** – **[in]** Configuration number
- **enable** – **[in]** Sets attribute if true, clears it otherwise

Returns
0 on success, other values on fail.

int **usbd_config_maxpower**(struct *usbd_context* *const uds_ctx, const enum *usbd_speed* speed, const uint8_t cfg, const uint8_t power)

Setup USB device configuration power consumption.

Parameters

- **uds_ctx** – **[in]** Pointer to USB device support context
- **speed** – **[in]** Configuration speed
- **cfg** – **[in]** Configuration number
- **power** – **[in]** Maximum power consumption value (bMaxPower)

Returns
0 on success, other values on fail.

bool **usbd_can_detect_vbus**(struct *usbd_context* *const uds_ctx)

Check that the controller can detect the VBUS state change.

This can be used in a generic application to explicitly handle the VBUS detected event after *usbd_init()*. For example, to call *usbd_enable()* after a short delay to give the PMIC time to detect the bus, or to handle cases where *usbd_enable()* can only be called after a VBUS detected event.

Parameters

- **uds_ctx** – **[in]** Pointer to USB device support context

Returns
true if controller can detect VBUS state change, false otherwise

struct **usbd_str_desc_data**

#include <usbd.h> Used internally to keep descriptors in order.

USBD string descriptor data

Public Members

uint8_t **idx**

Descriptor index, required for string descriptors.

enum usbd_str_desc_utype **utype**

Descriptor usage type (not bDescriptorType)

unsigned int **ascii7**

The string descriptor is in ASCII7 format.

unsigned int **use_hwinfo**

Device stack obtains SerialNumber using the HWINFO API.

struct **usbd_bos_desc_data**

#include <usbd.h> USBD BOS Device Capability descriptor data.

Public Members

enum usbd_bos_desc_utype **utype**

Descriptor usage type (not bDescriptorType)

struct **usbd_desc_node**

#include <usbd.h> Descriptor node.

Descriptor node is used to manage descriptors that are not directly part of a structure, such as string or BOS capability descriptors.

Public Members

sys_dnode_t **node**

slist node struct

const void *const **ptr**

Opaque pointer to a descriptor payload.

uint8_t **bLength**

Descriptor size in bytes.

uint8_t **bDescriptorType**

Descriptor type.

struct **usbd_config_node**

#include <usbd.h> Device configuration node.

Configuration node is used to manage device configurations, at least one configuration is required. It does not have an index, instead bConfigurationValue of the descriptor is used for identification.

Public Members

sys_snode_t **node**
slist node struct

void ***desc**
Pointer to configuration descriptor.

sys_slist_t **class_list**
List of registered classes (functions)

struct **usbd_ch9_data**
#include <*usbd.h*> USB device support middle layer runtime data.

Public Members

struct usb_setup_packet **setup**
Setup packet, up-to-date for the respective control request.

int **ctrl_type**
Control type, internally used for stage verification.

enum *usbd_ch9_state* **state**
Protocol state of the USB device stack.

uint32_t **ep_halt**
Halted endpoints bitmap.

uint8_t **configuration**
USB device stack selected configuration.

bool **post_status**
Post status stage work required, e.g.
set new device address

uint8_t **alternate**[16U]
Array to track interfaces alternate settings.

struct **usbd_status**
#include <*usbd.h*> USB device support status.

Public Members

unsigned int **initialized**
USB device support is initialized.

unsigned int **enabled**
: USB device support is enabled.

unsigned int **suspended**
: USB device is suspended.

unsigned int **rwup**
: USB remote wake-up feature is enabled.

enum *usbd_speed* **speed**
: USB device speed.

struct **usbd_context**
: *#include <usbd.h>* USB device support runtime context.

  Main structure that organizes all descriptors, configuration, and interfaces. An UDC device must be assigned to this structure.

Public Members

const char ***name**
: Name of the USB device.

struct *k_mutex* **mutex**
: Access mutex.

const struct *device* ***dev**
: Pointer to UDC device.

usbd_msg_cb_t **msg_cb**
: Notification message recipient callback.

struct *usbd_ch9_data* **ch9_data**
: Middle layer runtime data.

sys_dlist_t **descriptors**
: slist to manage descriptors like string, BOS

sys_slist_t **fs_configs**
: slist to manage Full-Speed device configurations

sys_slist_t **hs_configs**
: slist to manage High-Speed device configurations

struct *usbd_status* **status**
: Status of the USB device support.

void ***fs_desc**
: Pointer to Full-Speed device descriptor.

void *hs_desc
Pointer to High-Speed device descriptor.

struct usbd_cctx_vendor_req
#include <*usbd.h*> Vendor Requests Table.

Public Members

const uint8_t *reqs
Array of vendor requests supported by the class.

uint8_t len
Length of the array.

struct usbd_class_api
#include <*usbd.h*> USB device support class instance API.

Public Members

void (*feature_halt)(struct *usbd_class_data* *const c_data, uint8_t ep, bool halted)
Feature halt state update handler.

void (*update)(struct *usbd_class_data* *const c_data, uint8_t iface, uint8_t alternate)
Configuration update handler.

int (*control_to_dev)(struct *usbd_class_data* *const c_data, const struct usb_setup_packet *const setup, const struct *net_buf* *const buf)
USB control request handler to device.

int (*control_to_host)(struct *usbd_class_data* *const c_data, const struct usb_setup_packet *const setup, struct *net_buf* *const buf)
USB control request handler to host.

int (*request)(struct *usbd_class_data* *const c_data, struct *net_buf* *buf, int err)
Endpoint request completion event handler.

void (*suspended)(struct *usbd_class_data* *const c_data)
USB power management handler suspended.

void (*resumed)(struct *usbd_class_data* *const c_data)
USB power management handler resumed.

void (*sof)(struct *usbd_class_data* *const c_data)
Start of Frame.

void (*enable)(struct *usbd_class_data* *const c_data)
Class associated configuration is selected.

void (*disable)(struct *usbd_class_data* *const c_data)
Class associated configuration is disabled.

int (*init)(struct *usbd_class_data* *const c_data)
Initialization of the class implementation.

void (*shutdown)(struct *usbd_class_data* *const c_data)
Shutdown of the class implementation.

void *(*get_desc)(struct *usbd_class_data* *const c_data, const enum *usbd_speed* speed)
Get function descriptor based on speed parameter.

struct usbd_class_data
#include <*usbd.h*> USB device support class data.

Public Members

const char *name
Name of the USB device class instance.

struct *usbd_context* *uds_ctx
Pointer to USB device stack context structure.

const struct *usbd_class_api* *api
Pointer to device support class API.

const struct *usbd_cctx_vendor_req* *v_reqs
Supported vendor request table, can be NULL.

void *priv
Pointer to private data.

group usbd_msg_api

Enums

enum usbd_msg_type
USB device support message types.
The first set of message types map to event types from the UDC driver API.
Values:

enumerator USBD_MSG_VBUS_READY
VBUS ready message (optional)

enumerator USBD_MSG_VBUS_REMOVED
VBUS removed message (optional)

enumerator USBD_MSG_RESUME
Device resume message.

enumerator USBD_MSG_SUSPEND
Device suspended message.

enumerator USBD_MSG_RESET
Bus reset detected.

enumerator USBD_MSG_UDC_ERROR
Non-correctable UDC error message

enumerator USBD_MSG_STACK_ERROR
Unrecoverable device stack error message

enumerator USBD_MSG_CDC_ACM_LINE_CODING
CDC ACM Line Coding update.

enumerator USBD_MSG_CDC_ACM_CONTROL_LINE_STATE
CDC ACM Line State update.

enumerator USBD_MSG_MAX_NUMBER
Maximum number of message types.

Functions

static inline const char *usbd_msg_type_string(const enum *usbd_msg_type* type)
Returns the message type as a constant string.

Parameters

- type – **[in]** USBD message type

Returns
Message type as a constant string

struct usbd_msg
#include <*usbd_msg.h*> USB device message.

Public Members

enum *usbd_msg_type* type
Message type.

union usbd_msg
Message status, value or data.

HID device API

HID device specific API defined in include/zephyr/usb/class/usbd_hid.h.

Related code samples

USB HID keyboard
Implement a basic HID keyboard device.

API Reference

group `usbd_hid_device`

USBD HID Device API.

Enums

HID report types Report types used in Get/Set Report requests.

Values:

enumerator `HID_REPORT_TYPE_INPUT` = 1

enumerator `HID_REPORT_TYPE_OUTPUT`

enumerator `HID_REPORT_TYPE_FEATURE`

Functions

int `hid_device_register`(const struct *device* *dev, const uint8_t *const rdesc, const uint16_t rsize, const struct *hid_device_ops* *const ops)

Register HID device report descriptor and user callbacks.

The device must register report descriptor and user callbacks before USB device support is initialized and enabled.

Parameters

- `dev` – **[in]** Pointer to HID device
- `rdesc` – **[in]** Pointer to HID report descriptor
- `rsize` – **[in]** Size of HID report descriptor
- `ops` – **[in]** Pointer to HID device callbacks

int `hid_device_submit_report`(const struct *device* *dev, const uint16_t size, const uint8_t *const report)

Submit new input report.

Submit a new input report to be sent via the interrupt IN pipe. If sync is true, the functions will block until the report is sent. If the device does not provide input_report_done() callback, *hid_device_submit_report()* will be processed synchronously.

Parameters

- `dev` – **[in]** Pointer to HID device
- `size` – **[in]** Size of the input report

- `report` – **[in]** Input report buffer. Report buffer must be aligned.

Returns
0 on success, negative errno code on fail.

struct `hid_device_ops`

#include <usbd_hid.h> HID device user callbacks.

Each device depends on a user part that handles feature, input, and output report processing according to the device functionality described by the report descriptor. Which callbacks must be implemented depends on the device functionality. The USB device part of the HID device, cannot interpret device specific report descriptor and only handles USB specific parts, transfers and validation of requests, all reports are opaque to it. Callbacks are called from the USB device stack thread and must not block.

Public Members

void (*`iface_ready`)(const struct *device* *dev, const bool ready)

The interface ready callback is called with the ready argument set to true when the corresponding interface is part of the active configuration and the device can e.g.

begin submitting input reports, and with the argument set to false when the interface is no longer active. This callback is optional.

int (*`get_report`)(const struct *device* *dev, const uint8_t type, const uint8_t id, const uint16_t len, uint8_t *const buf)

This callback is called for the HID Get Report request to get a feature, input, or output report, which is specified by the argument type.

If there is no report ID in the report descriptor, the id argument is zero. The callback implementation must check the arguments, such as whether the report type is supported and the report length, and return a negative value to indicate an unsupported type or an error, or return the length of the report written to the buffer.

int (*`set_report`)(const struct *device* *dev, const uint8_t type, const uint8_t id, const uint16_t len, const uint8_t *const buf)

This callback is called for the HID Set Report request to set a feature, input, or output report, which is specified by the argument type.

If there is no report ID in the report descriptor, the id argument is zero. The callback implementation must check the arguments, such as whether the report type is supported, and return a nonzero value to indicate an unsupported type or an error.

void (*`set_idle`)(const struct *device* *dev, const uint8_t id, const uint32_t duration)

Notification to limit input report frequency.

The device should mute an input report submission until a new event occurs or until the time specified by the duration value has elapsed. If a report ID is used in the report descriptor, the device must store the duration and handle the specified report accordingly. Duration time resolution is in milliseconds.

uint32_t (*`get_idle`)(const struct *device* *dev, const uint8_t id)

If a report ID is used in the report descriptor, the device must implement this callback and return the duration for the specified report ID.

Duration time resolution is in milliseconds.

void (*`set_protocol`)(const struct *device* *dev, const uint8_t proto)

Notification that the host has changed the protocol from Boot Protocol(0) to Report Protocol(1) or vice versa.

void (*`input_report_done`)(const struct *device* *dev)

Notification that input report submitted with *hid_device_submit_report()* has been sent.

If the device does not use the callback, *hid_device_submit_report()* will be processed synchronously.

void (*`output_report`)(const struct *device* *dev, const uint16_t len, const uint8_t *const buf)

New output report callback.

Callback will only be called for reports received through the optional interrupt OUT pipe. If there is no interrupt OUT pipe, output reports will be received using *set_report()*. If a report ID is used in the report descriptor, the host places the ID in the buffer first, followed by the report data.

void (*`sof`)(const struct *device* *dev)

Optional Start of Frame (SoF) event callback.

There will always be software and hardware dependent jitter and latency. This should be used very carefully, it should not block and the execution time should be quite short.

Audio Class 2 device API

USB Audio Class 2 device specific API defined in include/zephyr/usb/class/usbd_uac2.h.

Related code samples

USB Audio asynchronous explicit feedback sample
USB Audio 2 explicit feedback sample playing audio on I2S.

API Reference

group `uac2_device`

USB Audio Class 2 device API.

Defines

`UAC2_ENTITY_ID`(node)

Get entity ID.

Parameters

- `node` – node identifier

Functions

void **usbd_uac2_set_ops**(const struct *device* *dev, const struct *uac2_ops* *ops, void *user_data)

Register USB Audio 2 application callbacks.

Parameters

- **dev** – USB Audio 2 device instance
- **ops** – USB Audio 2 callback structure
- **user_data** – Opaque user data to pass to ops callbacks

int **usbd_uac2_send**(const struct *device* *dev, uint8_t terminal, void *data, uint16_t size)

Send audio data to output terminal.

Data buffer must be sufficiently aligned and otherwise suitable for use by UDC driver.

Parameters

- **dev** – USB Audio 2 device
- **terminal** – Output Terminal ID linked to AudioStreaming interface
- **data** – Buffer containing outgoing data
- **size** – Number of bytes to send

Returns

0 on success, negative value on error

struct **uac2_ops**

#include <usbd_uac2.h> USB Audio 2 application event handlers.

Public Members

void (***sof_cb**)(const struct *device* *dev, void *user_data)

Start of Frame callback.

Notifies application about SOF event on the bus.

Param dev
USB Audio 2 device
Param user_data
Opaque user data pointer

void (***terminal_update_cb**)(const struct *device* *dev, uint8_t terminal, bool enabled, bool microframes, void *user_data)

Terminal update callback.

Notifies application that host has enabled or disabled a terminal.

Param dev
USB Audio 2 device
Param terminal
Terminal ID linked to AudioStreaming interface
Param enabled
True if host enabled terminal, False otherwise
Param microframes
True if USB connection speed uses microframes
Param user_data
Opaque user data pointer

void *(*`get_recv_buf`)(const struct *device* *dev, uint8_t terminal, uint16_t size, void *user_data)

Get receive buffer address.

USB stack calls this function to obtain receive buffer address for AudioStreaming interface. The buffer is owned by USB stack until *data_recv_cb* callback is called. The buffer must be sufficiently aligned and otherwise suitable for use by UDC driver.

Param dev
USB Audio 2 device

Param terminal
Input Terminal ID linked to AudioStreaming interface

Param size
Maximum number of bytes USB stack will write to buffer.

Param user_data
Opaque user data pointer

void (*`data_recv_cb`)(const struct *device* *dev, uint8_t terminal, void *buf, uint16_t size, void *user_data)

Data received.

This function releases buffer obtained in *get_recv_buf* after USB has written data to the buffer and/or no longer needs it.

Param dev
USB Audio 2 device

Param terminal
Input Terminal ID linked to AudioStreaming interface

Param buf
Buffer previously obtained via *get_recv_buf*

Param size
Number of bytes written to buffer

Param user_data
Opaque user data pointer

void (*`buf_release_cb`)(const struct *device* *dev, uint8_t terminal, void *buf, void *user_data)

Transmit buffer release callback.

This function releases buffer provided in *usbd_uac2_send* when the class no longer needs it.

Param dev
USB Audio 2 device

Param terminal
Output Terminal ID linked to AudioStreaming interface

Param buf
Buffer previously provided via *usbd_uac2_send*

Param user_data
Opaque user data pointer

uint32_t (*`feedback_cb`)(const struct *device* *dev, uint8_t terminal, void *user_data)

Get Explicit Feedback value.

Explicit feedback value format depends terminal connection speed. If device is High-Speed capable, it must use Q16.16 format if and only if the *terminal_update_cb* was called with microframes parameter set to true. On Full-Speed only devices, or if High-Speed capable device is operating at Full-Speed (microframes was false), the format is Q10.14 stored on 24 least significant bits (i.e. 8 most significant bits are ignored).

Param dev
USB Audio 2 device
Param terminal
Input Terminal ID whose feedback should be returned
Param user_data
Opaque user data pointer

USB Mass Storage Class device API

USB Mass Storage Class device API defined in include/zephyr/usb/class/usbd_msc.h.

Related code samples

USB Mass Storage
Expose board's RAM or FLASH as a USB disk using USB Mass Storage driver.

API Reference

group `usbd_msc_device`
USB Mass Storage Class device API.

Defines

`USBD_DEFINE_MSC_LUN`(disk_name, t10_vendor, t10_product, t10_revision)
Define USB Mass Storage Class logical unit.

Use this macro to create Logical Unit mapping in USB MSC for selected disk. Up to `CONFIG_USBD_MSC_LUNS_PER_INSTANCE` disks can be registered on single USB MSC instance. Currently only one USB MSC instance is supported.

Parameters

- `disk_name` – Disk name as used in *Disk Access Interface*
- `t10_vendor` – T10 Vendor Indetification
- `t10_product` – T10 Product Identification
- `t10_revision` – T10 Product Revision Level

6.5.5 USB host support APIs

USB host controller (UHC) driver API

The USB host controller driver API is described in include/zephyr/drivers/usb/uhc.h and referred to as the `UHC driver` API.

UHC driver API is experimental and is subject to change without notice.

Driver API reference

group `uhc_api`
USB host controller (UHC) driver API.

Defines

UHC_STATUS_INITIALIZED

Controller is initialized by *uhc_init()*

UHC_STATUS_ENABLED

Controller is enabled and all API functions are available.

Typedefs

typedef int (*uhc_event_cb_t)(const struct *device* *dev, const struct *uhc_event* *const event)

Callback to submit UHC event to higher layer.

At the higher level, the event is to be inserted into a message queue.

Param dev
[in] Pointer to device struct of the driver instance

Param event
[in] Point to event structure

Return
0 on success, all other values should be treated as error.

Enums

enum uhc_control_stage

USB control transfer stage.

Values:

enumerator UHC_CONTROL_STAGE_SETUP = 0

enumerator UHC_CONTROL_STAGE_DATA

enumerator UHC_CONTROL_STAGE_STATUS

enum uhc_event_type

USB host controller event types.

Values:

enumerator UHC_EVT_DEV_CONNECTED_LS

Low speed device connected.

enumerator UHC_EVT_DEV_CONNECTED_FS

Full speed device connected.

enumerator UHC_EVT_DEV_CONNECTED_HS

High speed device connected.

enumerator UHC_EVT_DEV_REMOVED
Device (peripheral) removed.

enumerator UHC_EVT_RESETED
Bus reset operation finished.

enumerator UHC_EVT_SUSPENDED
Bus suspend operation finished.

enumerator UHC_EVT_RESUMED
Bus resume operation finished.

enumerator UHC_EVT_RWUP
Remote wakeup signal.

enumerator UHC_EVT_EP_REQUEST
Endpoint request result event.

enumerator UHC_EVT_ERROR
Non-correctable error event, requires attention from higher levels or application.

Functions

static inline bool uhc_is_initialized(const struct *device* *dev)
Checks whether the controller is initialized.

Parameters

- dev – **[in]** Pointer to device struct of the driver instance

Returns
true if controller is initialized, false otherwise

static inline bool uhc_is_enabled(const struct *device* *dev)
Checks whether the controller is enabled.

Parameters

- dev – **[in]** Pointer to device struct of the driver instance

Returns
true if controller is enabled, false otherwise

static inline int uhc_bus_reset(const struct *device* *dev)
Reset USB bus.

Perform USB bus reset, controller may emit UHC_EVT_RESETED at the end of reset signaling.

Parameters

- dev – **[in]** Pointer to device struct of the driver instance

Return values
-EBUSY – if the controller is already performing a bus operation

Returns
0 on success, all other values should be treated as error.

static inline int **uhc_sof_enable**(const struct *device* *dev)

Enable Start of Frame generator.

Enable SOF generator.

Parameters

- **dev** – **[in]** Pointer to device struct of the driver instance

Return values

-EALREADY – if already enabled

Returns

0 on success, all other values should be treated as error.

static inline int **uhc_bus_suspend**(const struct *device* *dev)

Suspend USB bus.

Disable SOF generator and emit UHC_EVT_SUSPENDED event when USB bus is suspended.

Parameters

- **dev** – **[in]** Pointer to device struct of the driver instance

Return values

-EALREADY – if already suspended

Returns

0 on success, all other values should be treated as error.

static inline int **uhc_bus_resume**(const struct *device* *dev)

Resume USB bus.

Signal resume for at least 20ms, emit UHC_EVT_RESUMED at the end of USB bus resume signaling. The SoF generator should subsequently start within 3ms.

Parameters

- **dev** – **[in]** Pointer to device struct of the driver instance

Return values

-EBUSY – if the controller is already performing a bus operation

Returns

0 on success, all other values should be treated as error.

struct *uhc_transfer* ***uhc_xfer_alloc**(const struct *device* *dev, const uint8_t addr, const uint8_t ep, const uint8_t attrib, const uint16_t mps, const uint16_t timeout, void *const udev, void *const cb)

Allocate UHC transfer.

Allocate a new transfer from common transfer pool. Transfer has no buffer after allocation, but can be allocated and added from different pools.

Parameters

- **dev** – **[in]** Pointer to device struct of the driver instance
- **addr** – **[in]** Device (peripheral) address
- **ep** – **[in]** Endpoint address
- **attrib** – **[in]** Endpoint attributes
- **mps** – **[in]** Maximum packet size of the endpoint
- **timeout** – **[in]** Timeout in number of frames
- **udev** – **[in]** Opaque pointer to USB device

- `cb` – **[in]** Transfer completion callback

Returns
pointer to allocated transfer or NULL on error.

struct *uhc_transfer* *`uhc_xfer_alloc_with_buf`(const struct *device* *dev, const uint8_t addr, const uint8_t ep, const uint8_t attrib, const uint16_t mps, const uint16_t timeout, void *const udev, void *const cb, size_t size)

Allocate UHC transfer with buffer.

Allocate a new transfer from common transfer pool with buffer.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance
- `addr` – **[in]** Device (peripheral) address
- `ep` – **[in]** Endpoint address
- `attrib` – **[in]** Endpoint attributes
- `mps` – **[in]** Maximum packet size of the endpoint
- `timeout` – **[in]** Timeout in number of frames
- `udev` – **[in]** Opaque pointer to USB device
- `cb` – **[in]** Transfer completion callback
- `size` – **[in]** Size of the buffer

Returns
pointer to allocated transfer or NULL on error.

int `uhc_xfer_free`(const struct *device* *dev, struct *uhc_transfer* *const xfer)

Free UHC transfer and any buffers.

Free any buffers and put the transfer back into the transfer pool.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance
- `xfer` – **[in]** Pointer to UHC transfer

Returns
0 on success, all other values should be treated as error.

int `uhc_xfer_buf_add`(const struct *device* *dev, struct *uhc_transfer* *const xfer, struct *net_buf* *buf)

Add UHC transfer buffer.

Add a previously allocated buffer to the transfer.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance
- `xfer` – **[in]** Pointer to UHC transfer
- `buf` – **[in]** Pointer to UHC request buffer

Returns
pointer to allocated request or NULL on error.

struct *net_buf* *uhc_xfer_buf_alloc(const struct *device* *dev, const size_t size)

Allocate UHC transfer buffer.

Allocate a new buffer from common request buffer pool and assign it to the transfer if the xfer parameter is not NULL.

Parameters

- dev – **[in]** Pointer to device struct of the driver instance
- size – **[in]** Size of the request buffer

Returns

pointer to allocated request or NULL on error.

void uhc_xfer_buf_free(const struct *device* *dev, struct *net_buf* *const buf)

Free UHC request buffer.

Put the buffer back into the request buffer pool.

Parameters

- dev – **[in]** Pointer to device struct of the driver instance
- buf – **[in]** Pointer to UHC request buffer

int uhc_ep_enqueue(const struct *device* *dev, struct *uhc_transfer* *const xfer)

Queue USB host controller transfer.

Add transfer to the queue. If the queue is empty, the transfer can be claimed by the controller immediately.

Parameters

- dev – **[in]** Pointer to device struct of the driver instance
- xfer – **[in]** Pointer to UHC transfer

Return values

-EPERM – controller is not initialized

Returns

0 on success, all other values should be treated as error.

int uhc_ep_dequeue(const struct *device* *dev, struct *uhc_transfer* *const xfer)

Remove a USB host controller transfers from queue.

Not implemented yet.

Parameters

- dev – **[in]** Pointer to device struct of the driver instance
- xfer – **[in]** Pointer to UHC transfer

Return values

-EPERM – controller is not initialized

Returns

0 on success, all other values should be treated as error.

int uhc_init(const struct *device* *dev, *uhc_event_cb_t* event_cb)

Initialize USB host controller.

Initialize USB host controller.

Parameters

- dev – **[in]** Pointer to device struct of the driver instance
- event_cb – **[in]** Event callback from the higher layer (USB host stack)

Return values

- `-EINVAL` – on parameter error (no callback is passed)
- `-EALREADY` – already initialized

Returns
0 on success, all other values should be treated as error.

int `uhc_enable`(const struct *device* *dev)

Enable USB host controller.

Enable powered USB host controller and allow host stack to recognize and enumerate devices.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance

Return values

- `-EPERM` – controller is not initialized
- `-EALREADY` – already enabled

Returns
0 on success, all other values should be treated as error.

int `uhc_disable`(const struct *device* *dev)

Disable USB host controller.

Disable enabled USB host controller.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance

Return values
`-EALREADY` – already disabled

Returns
0 on success, all other values should be treated as error.

int `uhc_shutdown`(const struct *device* *dev)

Poweroff USB host controller.

Shut down the controller completely to reduce energy consumption or to change the role of the controller.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance

Return values
`-EALREADY` – controller is already uninitialized

Returns
0 on success, all other values should be treated as error.

static inline struct *uhc_device_caps* `uhc_caps`(const struct *device* *dev)

Get USB host controller capabilities.

Obtain the capabilities of the controller such as high speed (HS), and more.

Parameters

- `dev` – **[in]** Pointer to device struct of the driver instance

Returns
USB host controller capabilities.

struct **uhc_transfer**

#include <uhc.h> UHC endpoint buffer info.

This structure is mandatory for all UHC request. It contains the meta data about the request and FIFOs to store *net_buf* structures for each request.

The members of this structure should not be used directly by a higher layer (host stack).

Public Members

sys_dnode_t **node**

dlist node

uint8_t **setup_pkt**[8]

Control transfer setup packet.

struct *net_buf* ***buf**

Transfer data buffer.

uint8_t **addr**

Device (peripheral) address.

uint8_t **ep**

Endpoint to which request is associated.

uint8_t **attrib**

Endpoint attributes (TBD)

uint16_t **mps**

Maximum packet size.

uint16_t **timeout**

Timeout in number of frames.

unsigned int **queued**

Flag marks request buffer is queued.

unsigned int **stage**

Control stage status, up to the driver to use it or not.

void ***udev**

Pointer to USB device (opaque for the UHC)

void ***cb**

Pointer to transfer completion callback (opaque for the UHC)

int **err**

Transfer result, 0 on success, other values on error.

struct `uhc_event`

#include <*uhc.h*> USB host controller event.

Common structure for all events that originate from the UHC driver and are passed to higher layer using message queue and a callback (uhc_event_cb_t) provided by higher layer during controller initialization (uhc_init).

Public Members

sys_snode_t `node`

slist node for the message queue

enum *uhc_event_type* `type`

Event type.

int `status`

Event status value, if any.

struct *uhc_transfer* *`xfer`

Pointer to request used only for UHC_EVT_EP_REQUEST.

const struct *device* *`dev`

Pointer to controller's device struct.

struct `uhc_device_caps`

#include <*uhc.h*> USB host controller capabilities.

This structure is mainly intended for the USB host stack.

Public Members

uint32_t `hs`

USB high speed capable controller.

struct `uhc_data`

#include <*uhc.h*> Common UHC driver data structure.

Mandatory structure for each UHC controller driver. To be implemented as device's private data (device->data).

Public Members

struct *uhc_device_caps* `caps`

Controller capabilities.

struct *k_mutex* `mutex`

Driver access mutex.

sys_dlist_t **ctrl_xfers**
dlist for control transfers

sys_dlist_t **bulk_xfers**
dlist for bulk transfers

uhc_event_cb_t **event_cb**
Callback to submit an UHC event to upper layer.

atomic_t **status**
USB host controller status.

void ***priv**
Driver private data.

USB Power Delivery support

6.5.6 USB-C device stack

The USB-C device stack is a hardware independent interface between a Type-C Port Controller (TCPC) and customer applications. It is a port of the Google ChromeOS Type-C Port Manager (TCPM) stack. It provides the following functionalities:

- Uses the APIs provided by the Type-C Port Controller drivers to interact with the Type-C Port Controller.
- Provides a programming interface that's used by a customer applications. The APIs is described in include/zephyr/usb_c/usbc.h

Currently the device stack supports implementation of Sink only and Source only devices. Dual Role Power (DRP) devices are not yet supported.

List of samples for different purposes.

Implementing a Sink Type-C and Power Delivery USB-C device

The configuration of a USB-C Device is done in the stack layer and devicetree.

The following devicetree, structures and callbacks need to be defined:

- Devicetree usb-c-connector node referencing a TCPC
- Devicetree vbus node referencing a VBUS measurement device
- User defined structure that encapsulates application specific data
- Policy callbacks

For example, for the Sample USB-C Sink application:

Each Physical Type-C port is represented in the devicetree by a usb-c-connector compatible node:

```
port1: usbc-port@1 {
        compatible = "usb-c-connector";
        reg = <1>;
        tcpc = <&ucpd1>;
        vbus = <&vbus1>;
        power-role = "sink";
        sink-pdos = <PDO_FIXED(5000, 100, 0)>;
};
```

VBUS is measured by a device that's referenced in the devicetree by a usb-c-vbus-adc compatible node:

```
        vbus1: vbus {
                compatible = "zephyr,usb-c-vbus-adc";
                io-channels = <&adc2 8>;
                output-ohms = <49900>;
                full-ohms = <(330000 + 49900)>;
        };
```

A user defined structure is defined and later registered with the subsystem and can be accessed from callback through an API:

```
/**
 * @brief A structure that encapsulates Port data.
 */
static struct port0_data_t {
        /** Sink Capabilities */
        uint32_t snk_caps[DT_PROP_LEN(USBC_PORT0_NODE, sink_pdos)];
        /** Number of Sink Capabilities */
        int snk_cap_cnt;
        /** Source Capabilities */
        uint32_t src_caps[PDO_MAX_DATA_OBJECTS];
        /** Number of Source Capabilities */
        int src_cap_cnt;
        /* Power Supply Ready flag */
        atomic_t ps_ready;
} port0_data = {
        .snk_caps = {DT_FOREACH_PROP_ELEM(USBC_PORT0_NODE, sink_pdos, SINK_PDO)},
        .snk_cap_cnt = DT_PROP_LEN(USBC_PORT0_NODE, sink_pdos),
        .src_caps = {0},
        .src_cap_cnt = 0,
        .ps_ready = 0
};

```

These callbacks are used by the subsystem to set or get application specific data:

```
static int port0_policy_cb_get_snk_cap(const struct device *dev,
                                       uint32_t **pdos,
                                       int *num_pdos)
{
        struct port0_data_t *dpm_data = usbc_get_dpm_data(dev);

        *pdos = dpm_data->snk_caps;
        *num_pdos = dpm_data->snk_cap_cnt;

        return 0;
}

static void port0_policy_cb_set_src_cap(const struct device *dev,
                                        const uint32_t *pdos,
                                        const int num_pdos)
{
        struct port0_data_t *dpm_data;
        int num;
        int i;

        dpm_data = usbc_get_dpm_data(dev);

        num = num_pdos;
        if (num > PDO_MAX_DATA_OBJECTS) {
                num = PDO_MAX_DATA_OBJECTS;
```

(continues on next page)

(continued from previous page)

```
        }

        for (i = 0; i < num; i++) {
                dpm_data->src_caps[i] = *(pdos + i);
        }

        dpm_data->src_cap_cnt = num;
}

static uint32_t port0_policy_cb_get_rdo(const struct device *dev)
{
        struct port0_data_t *dpm_data = usbc_get_dpm_data(dev);

        return build_rdo(dpm_data);
}
```

This callback is used by the subsystem to query if a certain action can be taken:

```
bool port0_policy_check(const struct device *dev,
                        const enum usbc_policy_check_t policy_check)
{
        switch (policy_check) {
        case CHECK_POWER_ROLE_SWAP:
                /* Reject power role swaps */
                return false;
        case CHECK_DATA_ROLE_SWAP_TO_DFP:
                /* Reject data role swap to DFP */
                return false;
        case CHECK_DATA_ROLE_SWAP_TO_UFP:
                /* Accept data role swap to UFP */
                return true;
        case CHECK_SNK_AT_DEFAULT_LEVEL:
                /* This device is always at the default power level */
                return true;
        default:
                /* Reject all other policy checks */
                return false;

        }
}
```

This callback is used by the subsystem to notify the application of an event:

```
static void port0_notify(const struct device *dev,
                         const enum usbc_policy_notify_t policy_notify)
{
        struct port0_data_t *dpm_data = usbc_get_dpm_data(dev);

        switch (policy_notify) {
        case PROTOCOL_ERROR:
                break;
        case MSG_DISCARDED:
                break;
        case MSG_ACCEPT_RECEIVED:
                break;
        case MSG_REJECTED_RECEIVED:
                break;
        case MSG_NOT_SUPPORTED_RECEIVED:
                break;
        case TRANSITION_PS:
                atomic_set_bit(&dpm_data->ps_ready, 0);
```

(continues on next page)

(continued from previous page)

```
                break;
        case PD_CONNECTED:
                break;
        case NOT_PD_CONNECTED:
                break;
        case POWER_CHANGE_0A0:
                LOG_INF("PWR 0A");
                break;
        case POWER_CHANGE_DEF:
                LOG_INF("PWR DEF");
                break;
        case POWER_CHANGE_1A5:
                LOG_INF("PWR 1A5");
                break;
        case POWER_CHANGE_3A0:
                LOG_INF("PWR 3A0");
                break;
        case DATA_ROLE_IS_UFP:
                break;
        case DATA_ROLE_IS_DFP:
                break;
        case PORT_PARTNER_NOT_RESPONSIVE:
                LOG_INF("Port Partner not PD Capable");
                break;
        case SNK_TRANSITION_TO_DEFAULT:
                break;
        case HARD_RESET_RECEIVED:
                break;
        case SENDER_RESPONSE_TIMEOUT:
                break;
        case SOURCE_CAPABILITIES_RECEIVED:
                break;
        }
}
```

Registering the callbacks:

```
        /* Register USB-C Callbacks */

        /* Register Policy Check callback */
        usbc_set_policy_cb_check(usbc_port0, port0_policy_check);
        /* Register Policy Notify callback */
        usbc_set_policy_cb_notify(usbc_port0, port0_notify);
        /* Register Policy Get Sink Capabilities callback */
        usbc_set_policy_cb_get_snk_cap(usbc_port0, port0_policy_cb_get_snk_cap);
        /* Register Policy Set Source Capabilities callback */
        usbc_set_policy_cb_set_src_cap(usbc_port0, port0_policy_cb_set_src_cap);
        /* Register Policy Get Request Data Object callback */
        usbc_set_policy_cb_get_rdo(usbc_port0, port0_policy_cb_get_rdo);
```

Register the user defined structure:

```
        /* Set Application port data object. This object is passed to the policy callbacks␣
↪*/
        port0_data.ps_ready = ATOMIC_INIT(0);
        usbc_set_dpm_data(usbc_port0, &port0_data);
```

Start the USB-C subsystem:

```
        /* Start the USB-C Subsystem */
        usbc_start(usbc_port0);
```

Implementing a Source Type-C and Power Delivery USB-C device

The configuration of a USB-C Device is done in the stack layer and devicetree.

Define the following devicetree, structures and callbacks:

- Devicetree `usb-c-connector` node referencing a TCPC
- Devicetree `vbus` node referencing a VBUS measurement device
- User defined structure that encapsulates application specific data
- Policy callbacks

For example, for the Sample USB-C Source application:

Each Physical Type-C port is represented in the devicetree by a `usb-c-connector` compatible node:

```
                port1: usbc-port@1 {
                        compatible = "usb-c-connector";
                        reg = <1>;
                        tcpc = <&ucpd1>;
                        vbus = <&vbus1>;
                        power-role = "source";
                        typec-power-opmode = "3.0A";
                        source-pdos = <PDO_FIXED(5000, 100, 0) PDO_FIXED(9000, 100, 0) PDO_
↪FIXED(15000, 100, 0)>;
                };
```

VBUS is measured by a device that's referenced in the devicetree by a `usb-c-vbus-adc` compatible node:

```
        vbus1: vbus {
                compatible = "zephyr,usb-c-vbus-adc";
                io-channels = <&adc1 9>;
                output-ohms = <49900>;
                full-ohms = <(330000 + 49900)>;

                /* Pin B13 is used to control VBUS Discharge for Port1 */
                discharge-gpios = <&gpiob 13 GPIO_ACTIVE_HIGH>;
        };
```

A user defined structure is defined and later registered with the subsystem and can be accessed from callback through an API:

```
/**
 * @brief A structure that encapsulates Port data.
 */
static struct port0_data_t {
        /** Source Capabilities */
        uint32_t src_caps[DT_PROP_LEN(USBC_PORT0_NODE, source_pdos)];
        /** Number of Source Capabilities */
        int src_cap_cnt;
        /** CC Rp value */
        int rp;
        /** Sink Request RDO */
        union pd_rdo sink_request;
        /** Requested Object Pos */
        int obj_pos;
        /** VCONN CC line*/
        enum tc_cc_polarity vconn_pol;
        /** True if power supply is ready */
        bool ps_ready;
```

(continues on next page)

(continued from previous page)

```
        /** True if power supply should transition to a new level */
        bool ps_tran_start;
        /** Log Sink Requested RDO to console */
        atomic_t show_sink_request;
} port0_data = {
        .rp = DT_ENUM_IDX(USBC_PORT0_NODE, typec_power_opmode),
        .src_caps = {DT_FOREACH_PROP_ELEM(USBC_PORT0_NODE, source_pdos, SOURCE_PDO)},
        .src_cap_cnt = DT_PROP_LEN(USBC_PORT0_NODE, source_pdos),
};

```

These callbacks are used by the subsystem to set or get application specific data:

```
/**
 * @brief PE calls this function when it needs to set the Rp on CC
 */
int port0_policy_cb_get_src_rp(const struct device *dev,
                               enum tc_rp_value *rp)
{
        struct port0_data_t *dpm_data = usbc_get_dpm_data(dev);

        *rp = dpm_data->rp;

        return 0;
}

/**
 * @brief PE calls this function to Enable (5V) or Disable (0V) the
 *          Power Supply
 */
int port0_policy_cb_src_en(const struct device *dev, bool en)
{
        source_ctrl_set(en ? SOURCE_5V : SOURCE_0V);

        return 0;
}

/**
 * @brief PE calls this function to Enable or Disable VCONN
 */
int port0_policy_cb_vconn_en(const struct device *dev, enum tc_cc_polarity pol, bool en)
{
        struct port0_data_t *dpm_data = usbc_get_dpm_data(dev);

        dpm_data->vconn_pol = pol;

        if (en == false) {
                /* Disable VCONN on CC1 and CC2 */
                vconn_ctrl_set(VCONN_OFF);
        } else if (pol == TC_POLARITY_CC1) {
                /* set VCONN on CC1 */
                vconn_ctrl_set(VCONN1_ON);
        } else {
                /* set VCONN on CC2 */
                vconn_ctrl_set(VCONN2_ON);
        }

        return 0;
}

/**
```

(continues on next page)

(continued from previous page)

```
 * @brief PE calls this function to get the Source Caps that will be sent
 *          to the Sink
 */
int port0_policy_cb_get_src_caps(const struct device *dev,
                         const uint32_t **pdos, uint32_t *num_pdos)
{
        struct port0_data_t *dpm_data = usbc_get_dpm_data(dev);

        *pdos = dpm_data->src_caps;
        *num_pdos = dpm_data->src_cap_cnt;

        return 0;
}

/**
 * @brief PE calls this function to verify that a Sink's request if valid
 */
static enum usbc_snk_req_reply_t port0_policy_cb_check_sink_request(const struct device *dev,
                                        const uint32_t request_msg)
{
        struct port0_data_t *dpm_data = usbc_get_dpm_data(dev);
        union pd_fixed_supply_pdo_source pdo;
        uint32_t obj_pos;
        uint32_t op_current;

        dpm_data->sink_request.raw_value = request_msg;
        obj_pos = dpm_data->sink_request.fixed.object_pos;
        op_current =
                PD_CONVERT_FIXED_PDO_CURRENT_TO_MA(dpm_data->sink_request.fixed.operating_current);

        if (obj_pos == 0 || obj_pos > dpm_data->src_cap_cnt) {
                return SNK_REQUEST_REJECT;
        }

        pdo.raw_value = dpm_data->src_caps[obj_pos - 1];

        if (dpm_data->sink_request.fixed.operating_current > pdo.max_current) {
                return SNK_REQUEST_REJECT;
        }

        dpm_data->obj_pos = obj_pos;

        atomic_set_bit(&port0_data.show_sink_request, 0);

        /*
         * Clear PS ready. This will be set to true after PS is ready after
         * it transitions to the new level.
         */
        port0_data.ps_ready = false;

        return SNK_REQUEST_VALID;
}

/**
 * @brief PE calls this function to check if the Power Supply is at the requested
 *          level
 */
static bool port0_policy_cb_is_ps_ready(const struct device *dev)
{
```

(continues on next page)

(continued from previous page)

```
        struct port0_data_t *dpm_data = usbc_get_dpm_data(dev);


        /* Return true to inform that the Power Supply is ready */
        return dpm_data->ps_ready;
}

/**
 * @brief PE calls this function to check if the Present Contract is still
 *          valid
 */
static bool port0_policy_cb_present_contract_is_valid(const struct device *dev,
                                      const uint32_t present_contract)
{
        struct port0_data_t *dpm_data = usbc_get_dpm_data(dev);
        union pd_fixed_supply_pdo_source pdo;
        union pd_rdo request;
        uint32_t obj_pos;
        uint32_t op_current;

        request.raw_value = present_contract;
        obj_pos = request.fixed.object_pos;
        op_current = PD_CONVERT_FIXED_PDO_CURRENT_TO_MA(request.fixed.operating_current);

        if (obj_pos == 0 || obj_pos > dpm_data->src_cap_cnt) {
                return false;
        }

        pdo.raw_value = dpm_data->src_caps[obj_pos - 1];

        if (request.fixed.operating_current > pdo.max_current) {
                return false;
        }

        return true;
}

```

This callback is used by the subsystem to query if a certain action can be taken:

```
bool port0_policy_check(const struct device *dev,
                        const enum usbc_policy_check_t policy_check)
{
        struct port0_data_t *dpm_data = usbc_get_dpm_data(dev);

        switch (policy_check) {
        case CHECK_POWER_ROLE_SWAP:
                /* Reject power role swaps */
                return false;
        case CHECK_DATA_ROLE_SWAP_TO_DFP:
                /* Accept data role swap to DFP */
                return true;
        case CHECK_DATA_ROLE_SWAP_TO_UFP:
                /* Reject data role swap to UFP */
                return false;
        case CHECK_SRC_PS_AT_DEFAULT_LEVEL:
                /*
                 * This check is sent from the PE_SRC_Transition_to_default
                 * state and requires the following:
                 *        1: Vconn should be turned ON
                 *        2: Return TRUE when Power Supply is at default level
```

(continues on next page)

(continued from previous page)

```
                 */

                /* Power on VCONN */
                vconn_ctrl_set(dpm_data->vconn_pol);

                /* PS should be at default level after receiving a Hard Reset */
                return true;
        default:
                /* Reject all other policy checks */
                return false;

        }
}
```

This callback is used by the subsystem to notify the application of an event:

```
static void port0_notify(const struct device *dev,
                         const enum usbc_policy_notify_t policy_notify)
{
        struct port0_data_t *dpm_data = usbc_get_dpm_data(dev);

        switch (policy_notify) {
        case PROTOCOL_ERROR:
                break;
        case MSG_DISCARDED:
                break;
        case MSG_ACCEPT_RECEIVED:
                break;
        case MSG_REJECTED_RECEIVED:
                break;
        case MSG_NOT_SUPPORTED_RECEIVED:
                break;
        case TRANSITION_PS:
                dpm_data->ps_tran_start = true;
                break;
        case PD_CONNECTED:
                break;
        case NOT_PD_CONNECTED:
                break;
        case DATA_ROLE_IS_UFP:
                break;
        case DATA_ROLE_IS_DFP:
                break;
        case PORT_PARTNER_NOT_RESPONSIVE:
                LOG_INF("Port Partner not PD Capable");
                break;
        case HARD_RESET_RECEIVED:
                /*
                 * This notification is sent from the PE_SRC_Transition_to_default
                 * state and requires the following:
                 *        1: Vconn should be turned OFF
                 *        2: Reset of the local hardware
                 */

                /* Power off VCONN */
                vconn_ctrl_set(VCONN_OFF);
                /* Transition PS to Default level */
                source_ctrl_set(SOURCE_5V);
                break;
        default:
        }
}
```

Registering the callbacks:

```
        /* Register USB-C Callbacks */

        /* Register Policy Check callback */
        usbc_set_policy_cb_check(usbc_port0, port0_policy_check);
        /* Register Policy Notify callback */
        usbc_set_policy_cb_notify(usbc_port0, port0_notify);
        /* Register Policy callback to set the Rp on CC lines */
        usbc_set_policy_cb_get_src_rp(usbc_port0, port0_policy_cb_get_src_rp);
        /* Register Policy callback to enable or disable power supply */
        usbc_set_policy_cb_src_en(usbc_port0, port0_policy_cb_src_en);
        /* Register Policy callback to enable or disable vconn */
        usbc_set_vconn_control_cb(usbc_port0, port0_policy_cb_vconn_en);
        /* Register Policy callback to send the source caps to the sink */
        usbc_set_policy_cb_get_src_caps(usbc_port0, port0_policy_cb_get_src_caps);
        /* Register Policy callback to check if the sink request is valid */
        usbc_set_policy_cb_check_sink_request(usbc_port0, port0_policy_cb_check_sink_
↪request);
        /* Register Policy callback to check if the power supply is ready */
        usbc_set_policy_cb_is_ps_ready(usbc_port0, port0_policy_cb_is_ps_ready);
        /* Register Policy callback to check if Present Contract is still valid */
        usbc_set_policy_cb_present_contract_is_valid(usbc_port0,
                                port0_policy_cb_present_contract_is_valid);

```

Register the user defined structure:

```
        /* Set Application port data object. This object is passed to the policy callbacks␣
↪*/
        usbc_set_dpm_data(usbc_port0, &port0_data);
```

Start the USB-C subsystem:

```
        /* Start the USB-C Subsystem */
        usbc_start(usbc_port0);
```

API reference

Related code samples

Basic USB-C Sink
Implement a USB-C Power Delivery application in the form of a USB-C Sink.

Basic USB-C Source
Implement a USB-C Power Delivery application in the form of a USB-C Source.

group `_usbc_device_api`

USB-C Device APIs.

Since
3.3

Version
0.1.0

Defines

FIXED_5V_100MA_RDO

This Request Data Object (RDO) value can be returned from the policy_cb_get_rdo if 5V@100mA with the following options are sufficient for the Sink to operate.

The RDO is configured as follows: Maximum operating current 100mA Operating current 100mA Unchunked Extended Messages Not Supported No USB Suspend Not USB Communications Capable No capability mismatch Don't giveback Object position 1 (5V PDO)

Enums

enum usbc_policy_request_t

Device Policy Manager requests.

Values:

enumerator REQUEST_NOP

No request.

enumerator REQUEST_TC_DISABLED

Request Type-C layer to transition to Disabled State.

enumerator REQUEST_TC_ERROR_RECOVERY

Request Type-C layer to transition to Error Recovery State.

enumerator REQUEST_TC_END

End of Type-C requests.

enumerator REQUEST_PE_DR_SWAP

Request Policy Engine layer to perform a Data Role Swap.

enumerator REQUEST_PE_HARD_RESET_SEND

Request Policy Engine layer to send a hard reset.

enumerator REQUEST_PE_SOFT_RESET_SEND

Request Policy Engine layer to send a soft reset.

enumerator REQUEST_PE_GET_SRC_CAPS

Request Policy Engine layer to get Source Capabilities from port partner.

enumerator REQUEST_GET_SNK_CAPS

Request Policy Engine to get Sink Capabilities from port partner.

enumerator REQUEST_PE_GOTO_MIN

Request Policy Engine to request the port partner to source minimum power.

enum usbc_policy_notify_t

Device Policy Manager notifications.

Values:

enumerator MSG_ACCEPT_RECEIVED

Power Delivery Accept message was received.

enumerator MSG_REJECTED_RECEIVED

Power Delivery Reject message was received.

enumerator MSG_DISCARDED

Power Delivery discarded the message being transmitted.

enumerator MSG_NOT_SUPPORTED_RECEIVED

Power Delivery Not Supported message was received.

enumerator DATA_ROLE_IS_UFP

Data Role has been set to Upstream Facing Port (UFP)

enumerator DATA_ROLE_IS_DFP

Data Role has been set to Downstream Facing Port (DFP)

enumerator PD_CONNECTED

A PD Explicit Contract is in place.

enumerator NOT_PD_CONNECTED

No PD Explicit Contract is in place.

enumerator TRANSITION_PS

Transition the Power Supply.

enumerator PORT_PARTNER_NOT_RESPONSIVE

Port partner is not responsive.

enumerator PROTOCOL_ERROR

Protocol Error occurred.

enumerator SNK_TRANSITION_TO_DEFAULT

Transition the Sink to default.

enumerator HARD_RESET_RECEIVED

Hard Reset Received.

enumerator POWER_CHANGE_0A0

Sink SubPower state at 0V.

enumerator POWER_CHANGE_DEF

Sink SubPower state a 5V / 500mA.

enumerator POWER_CHANGE_1A5
Sink SubPower state a 5V / 1.5A.

enumerator POWER_CHANGE_3A0
Sink SubPower state a 5V / 3A.

enumerator SENDER_RESPONSE_TIMEOUT
Sender Response Timeout.

enumerator SOURCE_CAPABILITIES_RECEIVED
Source Capabilities Received.

enum usbc_policy_check_t
Device Policy Manager checks.
Values:

enumerator CHECK_POWER_ROLE_SWAP
Check if Power Role Swap is allowed.

enumerator CHECK_DATA_ROLE_SWAP_TO_DFP
Check if Data Role Swap to DFP is allowed.

enumerator CHECK_DATA_ROLE_SWAP_TO_UFP
Check if Data Role Swap to UFP is allowed.

enumerator CHECK_SNK_AT_DEFAULT_LEVEL
Check if Sink is at default level.

enumerator CHECK_VCONN_CONTROL
Check if should control VCONN.

enumerator CHECK_SRC_PS_AT_DEFAULT_LEVEL
Check if Source Power Supply is at default level.

enum usbc_policy_wait_t
Device Policy Manager Wait message notifications.
Values:

enumerator WAIT_SINK_REQUEST
The port partner is unable to meet the sink request at this time.

enumerator WAIT_POWER_ROLE_SWAP
The port partner is unable to do a Power Role Swap at this time.

enumerator WAIT_DATA_ROLE_SWAP
The port partner is unable to do a Data Role Swap at this time.

enumerator WAIT_VCONN_SWAP
The port partner is unable to do a VCONN Swap at this time.

enum usbc_snk_req_reply_t
Device Policy Manager's response to a Sink Request.

Values:

enumerator SNK_REQUEST_VALID
The sink port partner's request can be met.

enumerator SNK_REQUEST_REJECT
The sink port partner's request can not be met.

enumerator SNK_REQUEST_WAIT
The sink port partner's request can be met at a later time.

Functions

int usbc_start(const struct *device* *dev)
Start the USB-C Subsystem.

Parameters

- dev – Runtime device structure

Return values
0 – on success

int usbc_suspend(const struct *device* *dev)
Suspend the USB-C Subsystem.

Parameters

- dev – Runtime device structure

Return values
0 – on success

int usbc_request(const struct *device* *dev, const enum *usbc_policy_request_t* req)
Make a request of the USB-C Subsystem.

Parameters

- dev – Runtime device structure
- req – request

Return values
0 – on success

void usbc_bypass_next_sleep(const struct *device* *dev)

void usbc_set_dpm_data(const struct *device* *dev, void *dpm_data)
Set pointer to Device Policy Manager (DPM) data.

Parameters

- dev – Runtime device structure
- dpm_data – pointer to dpm data

void *usbc_get_dpm_data(const struct *device* *dev)

Get pointer to Device Policy Manager (DPM) data.

Parameters

- dev – Runtime device structure

Return values

- pointer – to dpm data that was set with usbc_set_dpm_data
- NULL – if dpm data was not set

void usbc_set_vconn_control_cb(const struct *device* *dev, const *tcpc_vconn_control_cb_t* cb)

Set the callback used to set VCONN control.

Parameters

- dev – Runtime device structure
- cb – VCONN control callback

void usbc_set_vconn_discharge_cb(const struct *device* *dev, const *tcpc_vconn_discharge_cb_t* cb)

Set the callback used to discharge VCONN.

Parameters

- dev – Runtime device structure
- cb – VCONN discharge callback

void usbc_set_policy_cb_check(const struct *device* *dev, const *policy_cb_check_t* cb)

Set the callback used to check a policy.

Parameters

- dev – Runtime device structure
- cb – callback

void usbc_set_policy_cb_notify(const struct *device* *dev, const *policy_cb_notify_t* cb)

Set the callback used to notify Device Policy Manager of a policy change.

Parameters

- dev – Runtime device structure
- cb – callback

void usbc_set_policy_cb_wait_notify(const struct *device* *dev, const *policy_cb_wait_notify_t* cb)

Set the callback used to notify Device Policy Manager of WAIT message reception.

Parameters

- dev – Runtime device structure
- cb – callback

void usbc_set_policy_cb_get_snk_cap(const struct *device* *dev, const *policy_cb_get_snk_cap_t* cb)

Set the callback used to get the Sink Capabilities.

Parameters

- dev – Runtime device structure
- cb – callback

void **usbc_set_policy_cb_set_src_cap**(const struct *device* *dev, const *policy_cb_set_src_cap_t* cb)

Set the callback used to store the received Port Partner's Source Capabilities.

Parameters

- **dev** – Runtime device structure
- **cb** – callback

void **usbc_set_policy_cb_get_rdo**(const struct *device* *dev, const *policy_cb_get_rdo_t* cb)

Set the callback used to get the Request Data Object (RDO)

Parameters

- **dev** – Runtime device structure
- **cb** – callback

void **usbc_set_policy_cb_is_snk_at_default**(const struct *device* *dev, const *policy_cb_is_snk_at_default_t* cb)

Set the callback used to check if the sink power supply is at the default level.

Parameters

- **dev** – Runtime device structure
- **cb** – callback

void **usbc_set_policy_cb_get_src_rp**(const struct *device* *dev, const *policy_cb_get_src_rp_t* cb)

Set the callback used to get the Rp value that should be placed on the CC lines.

Parameters

- **dev** – USB-C Connector Instance
- **cb** – callback

void **usbc_set_policy_cb_src_en**(const struct *device* *dev, const *policy_cb_src_en_t* cb)

Set the callback used to enable VBUS.

Parameters

- **dev** – USB-C Connector Instance
- **cb** – callback

void **usbc_set_policy_cb_get_src_caps**(const struct *device* *dev, const *policy_cb_get_src_caps_t* cb)

Set the callback used to get the Source Capabilities from the Device Policy Manager.

Parameters

- **dev** – USB-C Connector Instance
- **cb** – callback

void **usbc_set_policy_cb_check_sink_request**(const struct *device* *dev, const *policy_cb_check_sink_request_t* cb)

Set the callback used to check if Sink request is valid.

Parameters

- **dev** – USB-C Connector Instance
- **cb** – callback

void **usbc_set_policy_cb_is_ps_ready**(const struct *device* *dev, const *policy_cb_is_ps_ready_t* cb)

Set the callback used to check if Source Power Supply is ready.

Parameters

- **dev** – USB-C Connector Instance
- **cb** – callback

void **usbc_set_policy_cb_present_contract_is_valid**(const struct *device* *dev, const *policy_cb_present_contract_is_valid_t* cb)

Set the callback to check if present Contract is still valid.

Parameters

- **dev** – USB-C Connector Instance
- **cb** – callback

void **usbc_set_policy_cb_change_src_caps**(const struct *device* *dev, const *policy_cb_change_src_caps_t* cb)

Set the callback used to request that a different set of Source Caps be sent to the Sink.

Parameters

- **dev** – USB-C Connector Instance
- **cb** – callback

void **usbc_set_policy_cb_set_port_partner_snk_cap**(const struct *device* *dev, const *policy_cb_set_port_partner_snk_cap_t* cb)

Set the callback used to store the Capabilities received from a Sink Port Partner.

Parameters

- **dev** – USB-C Connector Instance
- **cb** – callback

SINK callback reference

group **sink_callbacks**

Typedefs

typedef int (***policy_cb_get_snk_cap_t**)(const struct *device* *dev, uint32_t **pdos, int *num_pdos)

Callback type used to get the Sink Capabilities.

Param dev
USB-C Connector Instance

Param pdos
pointer where pdos are stored

Param num_pdos
pointer where number of pdos is stored

Return
0 on success

typedef void (*`policy_cb_set_src_cap_t`)(const struct *device* *dev, const uint32_t *pdos, const int num_pdos)

Callback type used to report the received Port Partner's Source Capabilities.

Param dev
USB-C Connector Instance

Param pdos
pointer to the partner's source pdos

Param num_pdos
number of source pdos

typedef bool (*`policy_cb_check_t`)(const struct *device* *dev, const enum *usbc_policy_check_t* policy_check)

Callback type used to check a policy.

Param dev
USB-C Connector Instance

Param policy_check
policy to check

Return
true if policy is currently allowed by the device policy manager

typedef bool (*`policy_cb_wait_notify_t`)(const struct *device* *dev, const enum *usbc_policy_wait_t* wait_notify)

Callback type used to notify Device Policy Manager of WAIT message reception.

Param dev
USB-C Connector Instance

Param wait_notify
wait notification

Return
return true if the PE should wait and resend the message

typedef void (*`policy_cb_notify_t`)(const struct *device* *dev, const enum *usbc_policy_notify_t* policy_notify)

Callback type used to notify Device Policy Manager of a policy change.

Param dev
USB-C Connector Instance

Param policy_notify
policy notification

typedef uint32_t (*`policy_cb_get_rdo_t`)(const struct *device* *dev)

Callback type used to get the Request Data Object (RDO)

Param dev
USB-C Connector Instance

Return
RDO

typedef bool (*`policy_cb_is_snk_at_default_t`)(const struct *device* *dev)

Callback type used to check if the sink power supply is at the default level.

Param dev
USB-C Connector Instance

Return
true if power supply is at default level

SOURCE callback reference

group `source_callbacks`

Typedefs

typedef int (*`policy_cb_get_src_caps_t`)(const struct *device* *dev, const uint32_t **pdos, uint32_t *num_pdos)

Callback type used to get the Source Capabilities from the Device Policy Manager.

Param dev
USB-C Connector Instance

Param pdos
pointer to source capability pdos

Param num_pdos
pointer to number of source capability pdos

Return
0 on success

typedef enum *usbc_snk_req_reply_t* (*`policy_cb_check_sink_request_t`)(const struct *device* *dev, const uint32_t request_msg)

Callback type used to check if Sink request is valid.

Param dev
USB-C Connector Instance

Param request_msg
request message to check

Return
sink request reply

typedef bool (*`policy_cb_is_ps_ready_t`)(const struct *device* *dev)

Callback type used to check if Source Power Supply is ready.

Param dev
USB-C Connector Instance

Return
true if power supply is ready, else false

typedef bool (*`policy_cb_present_contract_is_valid_t`)(const struct *device* *dev, const uint32_t present_contract)

Callback type used to check if present Contract is still valid.

Param dev
USB-C Connector Instance

Param present_contract
present contract

Return
true if present contract is still valid

typedef bool (*`policy_cb_change_src_caps_t`)(const struct *device* *dev)

Callback type used to request that a different set of Source Caps be sent to the Sink.

Param dev
USB-C Connector Instance

Return
true if a different set of Source Caps is available

typedef void (*`policy_cb_set_port_partner_snk_cap_t`)(const struct *device* *dev, const uint32_t *pdos, const int num_pdos)

Callback type used to report the Capabilities received from a Sink Port Partner.

Param dev
USB-C Connector Instance

Param pdos
pointer to sink cap pdos

Param num_pdos
number of sink cap pdos

typedef int (*`policy_cb_get_src_rp_t`)(const struct *device* *dev, enum *tc_rp_value* *rp)

Callback type used to get the Rp value that should be placed on the CC lines.

Param dev
USB-C Connector Instance

Param rp
rp value

Return
0 on success

typedef int (*`policy_cb_src_en_t`)(const struct *device* *dev, bool en)

Callback type used to enable VBUS.

Param dev
USB-C Connector Instance

Param en
true if VBUS should be enabled, else false to disable it

Return
0 on success

Common sections related to USB support

6.5.7 Human Interface Devices (HID)

Common USB HID part that can be used outside of USB support, defined in header file include/zephyr/usb/class/hid.h.

HID types reference

group usb_hid_definitions

hid.h API

USB HID types and values

USB_HID_VERSION

HID Specification release v1.11.

USB_DESC_HID

USB HID Class HID descriptor type.

USB_DESC_HID_REPORT

USB HID Class Report descriptor type.

USB_DESC_HID_PHYSICAL

USB HID Class physical descriptor type.

USB_HID_GET_REPORT

USB HID Class GetReport bRequest value.

USB_HID_GET_IDLE

USB HID Class GetIdle bRequest value.

USB_HID_GET_PROTOCOL

USB HID Class GetProtocol bRequest value.

USB_HID_SET_REPORT

USB HID Class SetReport bRequest value.

USB_HID_SET_IDLE

USB HID Class SetIdle bRequest value.

USB_HID_SET_PROTOCOL

USB HID Class SetProtocol bRequest value.

HID_BOOT_IFACE_CODE_NONE

USB HID Boot Interface Protocol (bInterfaceProtocol) Code None.

HID_BOOT_IFACE_CODE_KEYBOARD

USB HID Boot Interface Protocol (bInterfaceProtocol) Code Keyboard.

HID_BOOT_IFACE_CODE_MOUSE

USB HID Boot Interface Protocol (bInterfaceProtocol) Code Mouse.

HID_PROTOCOL_BOOT

USB HID Class Boot protocol code.

HID_PROTOCOL_REPORT
: USB HID Class Report protocol code.

HID_ITEM_TYPE_MAIN
: HID Main item type.

HID_ITEM_TYPE_GLOBAL
: HID Global item type.

HID_ITEM_TYPE_LOCAL
: HID Local item type.

HID_ITEM_TAG_INPUT
: HID Input item tag.

HID_ITEM_TAG_OUTPUT
: HID Output item tag.

HID_ITEM_TAG_COLLECTION
: HID Collection item tag.

HID_ITEM_TAG_FEATURE
: HID Feature item tag.

HID_ITEM_TAG_COLLECTION_END
: HID End Collection item tag.

HID_ITEM_TAG_USAGE_PAGE
: HID Usage Page item tag.

HID_ITEM_TAG_LOGICAL_MIN
: HID Logical Minimum item tag.

HID_ITEM_TAG_LOGICAL_MAX
: HID Logical Maximum item tag.

HID_ITEM_TAG_PHYSICAL_MIN
: HID Physical Minimum item tag.

HID_ITEM_TAG_PHYSICAL_MAX
: HID Physical Maximum item tag.

HID_ITEM_TAG_UNIT_EXPONENT
: HID Unit Exponent item tag.

HID_ITEM_TAG_UNIT
: HID Unit item tag.

HID_ITEM_TAG_REPORT_SIZE
: HID Report Size item tag.

HID_ITEM_TAG_REPORT_ID
: HID Report ID item tag.

HID_ITEM_TAG_REPORT_COUNT
: HID Report count item tag.

HID_ITEM_TAG_USAGE
: HID Usage item tag.

HID_ITEM_TAG_USAGE_MIN
: HID Usage Minimum item tag.

HID_ITEM_TAG_USAGE_MAX
: HID Usage Maximum item tag.

HID_COLLECTION_PHYSICAL
: Physical collection type.

HID_COLLECTION_APPLICATION
: Application collection type.

HID_COLLECTION_LOGICAL
: Logical collection type.

HID_COLLECTION_REPORT
: Report collection type.

HID_COLLECTION_NAMED_ARRAY
: Named Array collection type.

HID_COLLECTION_USAGE_SWITCH
: Usage Switch collection type.

HID_COLLECTION_MODIFIER
: Modifier collection type.

HID_USAGE_GEN_DESKTOP
: HID Generic Desktop Controls Usage page.

HID_USAGE_GEN_KEYBOARD
: HID Keyboard Usage page.

HID_USAGE_GEN_LEDS
: HID LEDs Usage page.

HID_USAGE_GEN_BUTTON
: HID Button Usage page.

HID_USAGE_GEN_DESKTOP_UNDEFINED
: HID Generic Desktop Undefined Usage ID.

HID_USAGE_GEN_DESKTOP_POINTER
: HID Generic Desktop Pointer Usage ID.

HID_USAGE_GEN_DESKTOP_MOUSE
: HID Generic Desktop Mouse Usage ID.

HID_USAGE_GEN_DESKTOP_JOYSTICK
: HID Generic Desktop Joystick Usage ID.

HID_USAGE_GEN_DESKTOP_GAMEPAD
: HID Generic Desktop Gamepad Usage ID.

HID_USAGE_GEN_DESKTOP_KEYBOARD
: HID Generic Desktop Keyboard Usage ID.

HID_USAGE_GEN_DESKTOP_KEYPAD
: HID Generic Desktop Keypad Usage ID.

HID_USAGE_GEN_DESKTOP_X
: HID Generic Desktop X Usage ID.

HID_USAGE_GEN_DESKTOP_Y
: HID Generic Desktop Y Usage ID.

HID_USAGE_GEN_DESKTOP_WHEEL
: HID Generic Desktop Wheel Usage ID.

HID items reference

group usb_hid_items

Defines

HID_ITEM(bTag, bType, bSize)
: Define HID short item.

  Parameters

  - bTag – Item tag
  - bType – Item type
  - bSize – Item data size

Returns
HID Input item

`HID_INPUT(a)`

Define HID Input item with the data length of one byte.

For usage examples, see *HID_MOUSE_REPORT_DESC()*, *HID_KEYBOARD_REPORT_DESC()*

Parameters

- **a** – Input item data

Returns
HID Input item

`HID_OUTPUT(a)`

Define HID Output item with the data length of one byte.

For usage examples, see *HID_KEYBOARD_REPORT_DESC()*

Parameters

- **a** – Output item data

Returns
HID Output item

`HID_FEATURE(a)`

Define HID Feature item with the data length of one byte.

Parameters

- **a** – Feature item data

Returns
HID Feature item

`HID_COLLECTION(a)`

Define HID Collection item with the data length of one byte.

For usage examples, see *HID_MOUSE_REPORT_DESC()*, *HID_KEYBOARD_REPORT_DESC()*

Parameters

- **a** – Collection item data

Returns
HID Collection item

`HID_END_COLLECTION`

Define HID End Collection (non-data) item.

For usage examples, see *HID_MOUSE_REPORT_DESC()*, *HID_KEYBOARD_REPORT_DESC()*

Returns
HID End Collection item

`HID_USAGE_PAGE(page)`

Define HID Usage Page item.

For usage examples, see *HID_MOUSE_REPORT_DESC()*, *HID_KEYBOARD_REPORT_DESC()*

Parameters

- **page** – Usage Page

Returns
HID Usage Page item

`HID_LOGICAL_MIN8`(a)

Define HID Logical Minimum item with the data length of one byte.

For usage examples, see *HID_MOUSE_REPORT_DESC()*, *HID_KEYBOARD_REPORT_DESC()*

Parameters

- `a` – Minimum value in logical units

Returns
HID Logical Minimum item

`HID_LOGICAL_MAX8`(a)

Define HID Logical Maximum item with the data length of one byte.

For usage examples, see *HID_MOUSE_REPORT_DESC()*, *HID_KEYBOARD_REPORT_DESC()*

Parameters

- `a` – Maximum value in logical units

Returns
HID Logical Maximum item

`HID_LOGICAL_MIN16`(a, b)

Define HID Logical Minimum item with the data length of two bytes.

Parameters

- `a` – Minimum value lower byte
- `b` – Minimum value higher byte

Returns
HID Logical Minimum item

`HID_LOGICAL_MAX16`(a, b)

Define HID Logical Maximum item with the data length of two bytes.

Parameters

- `a` – Minimum value lower byte
- `b` – Minimum value higher byte

Returns
HID Logical Maximum item

`HID_LOGICAL_MIN32`(a, b, c, d)

Define HID Logical Minimum item with the data length of four bytes.

Parameters

- `a` – Minimum value lower byte
- `b` – Minimum value low middle byte
- `c` – Minimum value high middle byte
- `d` – Minimum value higher byte

Returns
HID Logical Minimum item

HID_LOGICAL_MAX32(a, b, c, d)

Define HID Logical Maximum item with the data length of four bytes.

Parameters

- a – Minimum value lower byte
- b – Minimum value low middle byte
- c – Minimum value high middle byte
- d – Minimum value higher byte

Returns
HID Logical Maximum item

HID_REPORT_SIZE(size)

Define HID Report Size item with the data length of one byte.

For usage examples, see *HID_MOUSE_REPORT_DESC()*, *HID_KEYBOARD_REPORT_DESC()*

Parameters

- size – Report field size in bits

Returns
HID Report Size item

HID_REPORT_ID(id)

Define HID Report ID item with the data length of one byte.

Parameters

- id – Report ID

Returns
HID Report ID item

HID_REPORT_COUNT(count)

Define HID Report Count item with the data length of one byte.

For usage examples, see *HID_MOUSE_REPORT_DESC()*, *HID_KEYBOARD_REPORT_DESC()*

Parameters

- count – Number of data fields included in the report

Returns
HID Report Count item

HID_USAGE(idx)

Define HID Usage Index item with the data length of one byte.

For usage examples, see *HID_MOUSE_REPORT_DESC()*, *HID_KEYBOARD_REPORT_DESC()*

Parameters

- idx – Number of data fields included in the report

Returns
HID Usage Index item

HID_USAGE_MIN8(a)

Define HID Usage Minimum item with the data length of one byte.

For usage examples, see *HID_MOUSE_REPORT_DESC()*, *HID_KEYBOARD_REPORT_DESC()*

Parameters

- a – Starting Usage

Returns
HID Usage Minimum item

HID_USAGE_MAX8(a)

Define HID Usage Maximum item with the data length of one byte.

For usage examples, see *HID_MOUSE_REPORT_DESC()*, *HID_KEYBOARD_REPORT_DESC()*

Parameters

- a – Ending Usage

Returns
HID Usage Maximum item

HID_USAGE_MIN16(a, b)

Define HID Usage Minimum item with the data length of two bytes.

For usage examples, see *HID_MOUSE_REPORT_DESC()*, *HID_KEYBOARD_REPORT_DESC()*

Parameters

- a – Starting Usage lower byte
- b – Starting Usage higher byte

Returns
HID Usage Minimum item

HID_USAGE_MAX16(a, b)

Define HID Usage Maximum item with the data length of two bytes.

For usage examples, see *HID_MOUSE_REPORT_DESC()*, *HID_KEYBOARD_REPORT_DESC()*

Parameters

- a – Ending Usage lower byte
- b – Ending Usage higher byte

Returns
HID Usage Maximum item

HID Mouse and Keyboard report descriptors

The pre-defined Mouse and Keyboard report descriptors can be used by a HID device implementation or simply as examples.

group usb_hid_mk_report_desc

Defines

HID_MOUSE_REPORT_DESC(bcnt)

Simple HID mouse report descriptor for n button mouse.

Parameters

- bcnt – Button count. Allowed values from 1 to 8.

HID_KEYBOARD_REPORT_DESC()
Simple HID keyboard report descriptor.

Enums

enum hid_kbd_code
HID keyboard button codes.
Values:

enumerator HID_KEY_A = 4

enumerator HID_KEY_B = 5

enumerator HID_KEY_C = 6

enumerator HID_KEY_D = 7

enumerator HID_KEY_E = 8

enumerator HID_KEY_F = 9

enumerator HID_KEY_G = 10

enumerator HID_KEY_H = 11

enumerator HID_KEY_I = 12

enumerator HID_KEY_J = 13

enumerator HID_KEY_K = 14

enumerator HID_KEY_L = 15

enumerator HID_KEY_M = 16

enumerator HID_KEY_N = 17

enumerator HID_KEY_O = 18

enumerator HID_KEY_P = 19

enumerator HID_KEY_Q = 20

enumerator HID_KEY_R = 21

enumerator HID_KEY_S = 22

enumerator HID_KEY_T = 23

enumerator HID_KEY_U = 24

enumerator HID_KEY_V = 25

enumerator HID_KEY_W = 26

enumerator HID_KEY_X = 27

enumerator HID_KEY_Y = 28

enumerator HID_KEY_Z = 29

enumerator HID_KEY_1 = 30

enumerator HID_KEY_2 = 31

enumerator HID_KEY_3 = 32

enumerator HID_KEY_4 = 33

enumerator HID_KEY_5 = 34

enumerator HID_KEY_6 = 35

enumerator HID_KEY_7 = 36

enumerator HID_KEY_8 = 37

enumerator HID_KEY_9 = 38

enumerator HID_KEY_0 = 39

enumerator HID_KEY_ENTER = 40

enumerator HID_KEY_ESC = 41

enumerator HID_KEY_BACKSPACE = 42

enumerator HID_KEY_TAB = 43

enumerator HID_KEY_SPACE = 44

enumerator HID_KEY_MINUS = 45

enumerator HID_KEY_EQUAL = 46

enumerator HID_KEY_LEFTBRACE = 47

enumerator HID_KEY_RIGHTBRACE = 48

enumerator HID_KEY_BACKSLASH = 49

enumerator HID_KEY_HASH = 50

enumerator HID_KEY_SEMICOLON = 51

enumerator HID_KEY_APOSTROPHE = 52

enumerator HID_KEY_GRAVE = 53

enumerator HID_KEY_COMMA = 54

enumerator HID_KEY_DOT = 55

enumerator HID_KEY_SLASH = 56

enumerator HID_KEY_CAPSLOCK = 57

enumerator HID_KEY_F1 = 58

enumerator HID_KEY_F2 = 59

enumerator HID_KEY_F3 = 60

enumerator HID_KEY_F4 = 61

enumerator HID_KEY_F5 = 62

enumerator HID_KEY_F6 = 63

enumerator HID_KEY_F7 = 64

enumerator HID_KEY_F8 = 65

enumerator HID_KEY_F9 = 66

enumerator HID_KEY_F10 = 67

enumerator HID_KEY_F11 = 68

enumerator HID_KEY_F12 = 69

enumerator HID_KEY_SYSRQ = 70

enumerator HID_KEY_SCROLLLOCK = 71

enumerator HID_KEY_PAUSE = 72

enumerator HID_KEY_INSERT = 73

enumerator HID_KEY_HOME = 74

enumerator HID_KEY_PAGEUP = 75

enumerator HID_KEY_DELETE = 76

enumerator HID_KEY_END = 77

enumerator HID_KEY_PAGEDOWN = 78

enumerator HID_KEY_RIGHT = 79

enumerator HID_KEY_LEFT = 80

enumerator HID_KEY_DOWN = 81

enumerator HID_KEY_UP = 82

enumerator HID_KEY_NUMLOCK = 83

enumerator HID_KEY_KPSLASH = 84

enumerator HID_KEY_KPASTERISK = 85

enumerator HID_KEY_KPMINUS = 86

enumerator HID_KEY_KPPLUS = 87

enumerator HID_KEY_KPENTER = 88

enumerator HID_KEY_KP_1 = 89

enumerator HID_KEY_KP_2 = 90

enumerator HID_KEY_KP_3 = 91

enumerator HID_KEY_KP_4 = 92

enumerator HID_KEY_KP_5 = 93

enumerator HID_KEY_KP_6 = 94

enumerator HID_KEY_KP_7 = 95

enumerator HID_KEY_KP_8 = 96

enumerator HID_KEY_KP_9 = 97

enumerator HID_KEY_KP_0 = 98

enum hid_kbd_modifier

HID keyboard modifiers.

Values:

enumerator HID_KBD_MODIFIER_NONE = 0x00

enumerator HID_KBD_MODIFIER_LEFT_CTRL = 0x01

enumerator HID_KBD_MODIFIER_LEFT_SHIFT = 0x02

enumerator HID_KBD_MODIFIER_LEFT_ALT = 0x04

enumerator HID_KBD_MODIFIER_LEFT_UI = 0x08

enumerator HID_KBD_MODIFIER_RIGHT_CTRL = 0x10

enumerator HID_KBD_MODIFIER_RIGHT_SHIFT = 0x20

enumerator HID_KBD_MODIFIER_RIGHT_ALT = 0x40

enumerator HID_KBD_MODIFIER_RIGHT_UI = 0x80

enum hid_kbd_led

HID keyboard LEDs.

Values:

enumerator HID_KBD_LED_NUM_LOCK = 0x01

enumerator HID_KBD_LED_CAPS_LOCK = 0x02

enumerator HID_KBD_LED_SCROLL_LOCK = 0x04

enumerator HID_KBD_LED_COMPOSE = 0x08

enumerator HID_KBD_LED_KANA = 0x10

Chapter 7

Hardware Support

7.1 Architecture-related Guides

7.1.1 Zephyr support status on ARC processors

Overview

This page describes current state of Zephyr for ARC processors and some future plans. Please note that

- plans are given without exact deadlines
- software features require corresponding hardware to be present and configured the proper way
- not all the features can be enabled at the same time

Support status

Legend: **Y** - yes, supported; **N** - no, not supported; **WIP** - Work In Progress; **TBD** - to be decided

	Processor families				
	EM	**HS3x/4x**	**VPX**	**HS5x**	**HS6x**
Port status	up-streamed	up-streamed	up-streamed	up-streamed	up-streamed
Features					
Closely coupled memories (ICCM, DCCM)[1]	Y	Y	Y	TBD	TBD
Execution with caches - Instruction/Data, L1/L2 caches	Y	Y	Y	Y	Y
Hardware-assisted unaligned memory access	Y[2]	Y	Y	Y	Y
Regular interrupts with multiple priority levels, direct interrupts	Y	Y	Y	Y	Y
Fast interrupts, separate register banks for fast interrupts	Y	Y	TBD	N	N
Hardware floating point unit (FPU)	Y	Y	TBD[6]	TBD	TBD
Symmetric multiprocessing (SMP) support, switch-based	N/A	Y	TBD	Y	Y
Hardware-assisted stack checking	Y	Y	Y	N	N
Hardware-assisted atomic operations	N/A	Y	Y	Y	Y
DSP ISA	Y	N[3]	TBD[6]	TBD	TBD
DSP AGU/XY extensions	Y	N[3]	N/A	TBD	TBD
Userspace	Y	Y	N	TBD	TBD
Memory protection unit (MPU)	Y	Y	TBD	N	N
Memory management unit (MMU)	N/A	N	TBD	N	N
SecureShield	Y	N/A	N/A	N/A	N/A
Single-thread kernel support[5]	Y	Y	Y	Y	Y
Toolchains					
GNU (open source GCC-based)	Y	Y	N	Y	Y
MetaWare (proprietary Clang-based)	Y	Y	Y	Y	Y
Simulators					
QEMU (open source)[4]	Y	Y	N	Y	Y
nSIM (proprietary, provided by MetaWare Development Tools)	Y	Y	Y	Y	Y

Notes

7.1.2 Arm Cortex-M Developer Guide

Overview

This page contains detailed information about the status of the Arm Cortex-M architecture porting in the Zephyr RTOS and describes key aspects when developing Zephyr applications for Arm Cortex-M-based platforms.

[6] currently only ARC VPX scalar port is supported. The support of VPX vector pipeline, VCCM, STU is not included in this port, and require additional development and / or other runtime integration.
[1] usage of CCMs is limited on SMP systems
[2] except the systems with secure features (SecureShield) due to HW limitation
[3] We only support save/restore ACCL/ACCH registers in task's context. Rest of DSP/AGU registers save/restore isn't implemented but kernel itself does not use these registers. This allows single task per core to use DSP/AGU safely.
[5] Single-thread kernel is support only for single core targets
[4] QEMU doesn't support all the ARC processor's HW features. For the detailed info please check the ARC QEMU documentation

Key supported features

The table below summarizes the status of key OS features in the different Arm Cortex-M implementation variants.

		Processor families							
Architecture variant		Arm v6-M		Arm v7-M			Arm v8-M		Arm v8.1-M
		M0/M	**M0+**	**M3**	**M4**	**M7**	**M23**	**M33**	**M55**
OS Features									
Programmable fault IRQ priorities		Y	N	Y	Y	Y	N	Y	Y
Single-thread kernel support		Y	Y	Y	Y	Y	Y	Y	Y
Thread local storage support		Y	Y	Y	Y	Y	Y	Y	Y
Interrupt handling									
	Regular interrupts	Y	Y	Y	Y	Y	Y	Y	Y
	Dynamic interrupts	Y	Y	Y	Y	Y	Y	Y	Y
	Direct interrupts	Y	Y	Y	Y	Y	Y	Y	Y
	Zero Latency interrupts	N	N	Y	Y	Y	Y	Y	Y
CPU idling		Y	Y	Y	Y	Y	Y	Y	Y
Native system timer (SysTick)		N[1]	Y	Y	Y	Y	Y	Y	Y
Memory protection									
	User mode	N	Y	Y	Y	Y	Y	Y	Y
	HW stack protection (MPU)	N	N	Y	Y	Y	Y	Y	Y
	HW-assisted stack limit checking	N	N	N	N	N	Y[2]	Y	Y
HW-assisted null-pointer dereference detection		N	N	Y	Y	Y	Y	Y	Y
HW-assisted atomic operations		N	N	Y	Y	Y	N	Y	Y
Support for non-cacheable regions		N	N	Y	Y	Y	N	Y	Y
Execute SRAM functions		N	N	Y	Y	Y	N	Y	Y
Floating Point Services		N	N	N	Y	Y	N	Y	Y
DSP ISA		N	N	N	Y	Y	N	Y	Y
Trusted-Execution									
	Native TrustZone-M support	N	N	N	N	N	Y	Y	Y
	TF-M integration	N	N	N	N	N	N	Y	N
Code relocation		Y	Y	Y	Y	Y	Y	Y	Y
SW-based vector table relaying		Y	Y	Y	Y	Y	Y	Y	Y
HW-assisted timing functions		N	N	Y	Y	Y	N	Y	Y

Notes

OS features

Threads

Thread stack alignment Each Zephyr thread is defined with its own stack memory. By default, Cortex-M enforces a double word thread stack alignment, see CONFIG_STACK_ALIGN_DOUBLE_WORD. If MPU-based HW-assisted stack overflow detection (CONFIG_MPU_STACK_GUARD) is enabled, thread stacks need to be aligned with a larger value, reflected by CONFIG_ARM_MPU_REGION_MIN_ALIGN_AND_SIZE. In Arm v6-M and Arm v7-M architecture variants, thread stacks are additionally required to align with a value equal to their size, in applications that need to support user mode (CONFIG_USERSPACE). The thread stack sizes in that case need to be a power of two. This is all reflected by CONFIG_MPU_REQUIRES_POWER_OF_TWO_ALIGNMENT, that is enforced in Arm v6-M and Arm v7-M builds with user mode support.

Stack pointers While executing in thread mode the processor is using the Process Stack Pointer (PSP). The processor uses the Main Stack Pointer (MSP) while executing in handler mode, that is, while servicing exceptions and HW interrupts. Using PSP in thread mode *facilitates thread stack pointer manipulation* during thread context switching, without affecting the current execution context flow in handler mode.

In Arm Cortex-M builds a single interrupt stack memory is shared among exceptions and interrupts. The size of the interrupt stack needs to be selected taking into consideration nested interrupts, each pushing an additional stack frame. Developers can modify the interrupt stack size using CONFIG_ISR_STACK_SIZE.

The interrupt stack is also used during early boot so the kernel can initialize the main thread's stack before switching to the main thread.

Thread context switching In Arm Cortex-M builds, the PendSV exception is used in order to trigger a context switch to a different thread. PendSV exception is always present in Cortex-M implementations. PendSV is configured with the lowest possible interrupt priority level, in all Cortex-M variants. The main reasons for that design are

- to utilize the tail chaining feature of Cortex-M processors, and thus limit the number of context switch operations that occur.
- to not impact the interrupt latency observed by HW interrupts.

As a result, context switch in Cortex-M is non-atomic, i.e. it may be *preempted* by HW interrupts, however, a context-switch operation must be completed before a new thread context-switch may start.

Typically a thread context-switch will perform the following operations

- When switching-out the current thread, the processor stores
  - the callee-saved registers (R4 - R11) in the thread's container for callee-saved registers, which is located in kernel memory
  - the thread's current operation *mode*
    - user or privileged execution mode
    - presence of an active floating point context

[1] SysTick is optional in Cortex-M1
[2] Stack limit checking only in Secure builds in Cortex-M23

        * the EXC_RETURN value of the current handler context (PendSV)
    - the floating point callee-saved registers (S16 - S31) in the thread's container for FP callee-saved registers, if the current thread has an active FP context
    - the PSP of the current thread which points to the beginning of the current thread's exception stack frame. The latter contains the caller-saved context and the return address of the switched-out thread.
- When switching-in a new thread the processor
    - restores the new thread's callee-saved registers from the thread's container for callee-saved registers
    - restores the new thread's operation *mode*
    - restores the FP callee-saved registers if the switched-in thread had an active FP context before being switched-out
    - re-programs the dynamic MPU regions to allow a user thread access its stack and application memories, and/or programs a stack-overflow MPU guard at the bottom of the thread's privileged stack
    - restores the PSP for the incoming thread and re-programs the stack pointer limit register (if applicable, see `CONFIG_BUILTIN_STACK_GUARD`)
    - optionally does a stack limit checking for the switched-in thread, if sentinel-based stack limit checking is enabled (see `CONFIG_STACK_SENTINEL`).

PendSV exception return sequence restores the new thread's caller-saved registers and the return address, as part of unstacking the exception stack frame.

The implementation of the context-switch mechanism is present in `arch/arm/core/cortex_m/swap_helper.S`.

Stack limit checking (Arm v8-M) Armv8-M and Armv8.1-M variants support stack limit checking using the MSPLIM and PSPLIM core registers. The feature is enabled when `CONFIG_BUILTIN_STACK_GUARD` is set. When stack limit checking is enabled, both the thread's privileged or user stack, as well as the interrupt stack are guarded by PSPLIM and MSPLIM registers, respectively. MSPLIM is configured *once* during kernel boot, while PSLIM is re-programmed during every thread context-switch or during system calls, when the thread switches from using its default stack to using its privileged stack, and vice versa. PSPLIM re-programming

- has a relatively low runtime overhead (programming is done with MSR instructions)
- does not impact interrupt latency
- does not require any memory areas to be reserved for stack guards
- does not make use of MPU regions

It is, therefore, considered as a lightweight but very efficient stack overflow detection mechanism in Cortex-M applications.

Stack overflows trigger the dedicated UsageFault exception provided by Arm v8-M.

Interrupt handling features This section describes certain aspects around exception and interrupt handling in Arm Cortex-M.

Interrupt priority levels The number of available (configurable) interrupt priority levels is determined by the number of implemented interrupt priority bits in NVIC; this needs to be described for each Cortex-M platform using DeviceTree:

```
&nvic {
        arm,num-irq-priority-bits = <#priority-bits>;
};
```

Reserved priority levels A number of interrupt priority levels are reserved for the OS.

By design, system fault exceptions have the highest priority level. In *Baseline* Cortex-M, this is actually enforced by hardware, as HardFault is the only available processor fault exception, and its priority is higher than any configurable exception priority.

In *Mainline* Cortex-M, the available fault exceptions (e.g. MemManage-Fault, UsageFault, etc.) are assigned the highest *configurable* priority level. (`CONFIG_CPU_CORTEX_M_HAS_PROGRAMMABLE_FAULT_PRIOS` signifies explicitly that the Cortex-M implementation supports configurable fault priorities.)

This priority level is never shared with HW interrupts (an exception to this rule is described below). As a result, processor faults occurring in regular ISRs will be handled by the corresponding fault handler and will not escalate to a HardFault, *similar to processor faults occurring in thread mode*.

SVC exception is normally configured with the highest configurable priority level (an exception to this rule will be described below). SVCs are used by the Zephyr kernel to dispatch system calls, trigger runtime system errors (e.g. Kernel oops or panic), or implement IRQ offloading.

In Baseline Cortex-M the priority level of SVC may be shared with other exceptions or HW interrupts that are also given the highest configurable priority level (As a result of this, kernel runtime errors during interrupt handling will escalate to HardFault. Additional logic in the fault handling routines ensures that such runtime errors are detected successfully).

In Mainline Cortex-M, however, the SVC priority level is *reserved*, thus normally it is only shared with the fault exceptions of configurable priority. This simplifies the fault handling routines in Mainline Cortex-M architecture, since runtime kernel errors are serviced by the SVC handler (i.e no HardFault escalation, even if the kernel errors occur in ISR context).

HW interrupts in Mainline Cortex-M builds are allocated a priority level lower than the SVC.

One exception to the above rules is when Zephyr applications support Zero Latency Interrupts (ZLIs). Such interrupts are designed to have a priority level higher than any HW or system interrupt. If the ZLI feature is enabled in Mainline Cortex-M builds (see `CONFIG_ZERO_LATENCY_IRQS`), then

- ZLIs are assigned the highest configurable priority level
- SVCs are assigned the second highest configurable priority level
- Regular HW interrupts are assigned priority levels lower than SVC.

The priority level configuration in Cortex-M is implemented in `include/zephyr/arch/arm/cortex_m/exception.h`.

Locking and unlocking IRQs In Baseline Cortex-M locking interrupts is implemented using the PRIMASK register.

```
arch_irq_lock()
```

will set the PRIMASK register to 1, eventually, masking all IRQs with configurable priority. While this fulfils the OS requirement of locking interrupts, the consequence is that kernel runtime errors (triggering SVCs) will escalate to HardFault.

In Mainline Cortex-M locking interrupts is implemented using the BASEPRI register (Mainline Cortex-M builds select `CONFIG_CPU_CORTEX_M_HAS_BASEPRI` to signify that BASEPRI register is im-

plemented.). By modifying BASEPRI (or BASEPRI_MAX) arch_irq_lock() masks all system and HW interrupts with the exception of

- SVCs
- processor faults
- ZLIs

This allows zero latency interrupts to be triggered inside OS critical sections. Additionally, this allows system (processor and kernel) faults to be handled by Zephyr in *exactly the same way*, regardless of whether IRQs have been locked or not when the error occurs. It also allows for system calls to be dispatched while IRQs are locked.

Note

Mainline Cortex-M fault handling is designed and configured in a way that all processor and kernel faults are handled by the corresponding exception handlers and never result in HardFault escalation. In other words, a HardFault may only occur in Zephyr applications that have modified the default fault handling configurations. The main reason for this design was to reserve the HardFault exception for handling exceptional error conditions in safety critical applications.

Dynamic direct interrupts Cortex-M builds support the installation of direct interrupt service routines during runtime. Direct interrupts are designed for performance-critical interrupt handling and do not go through all of the common Zephyr interrupt handling code.

Direct dynamic interrupts are enabled via switching on `CONFIG_DYNAMIC_DIRECT_INTERRUPTS`.

Note that enabling direct dynamic interrupts requires enabling support for dynamic interrupts in the kernel, as well (see `CONFIG_DYNAMIC_INTERRUPTS`).

Zero Latency interrupts As described above, in Mainline Cortex-M applications, the Zephyr kernel reserves the highest configurable interrupt priority level for its own use (SVC). SVCs will not be masked by interrupt locking. Zero-latency interrupt can be used to set up an interrupt at the highest interrupt priority which will not be blocked by interrupt locking. To use the ZLI feature `CONFIG_ZERO_LATENCY_IRQS` needs to be enabled.

Zero latency IRQs have minimal interrupt latency, as they will always preempt regular HW or system interrupts.

Note, however, that since ZLI ISRs will run at a priority level higher than the kernel exceptions they **cannot use** any kernel functionality. Additionally, since the ZLI interrupt priority level is equal to processor fault priority level, faults occurring in ZLI ISRs will escalate to HardFault and will not be handled in the same way as regular processor faults. Developers need to be aware of this limitation.

CPU Idling The Cortex-M architecture port implements both k_cpu_idle() and k_cpu_atomic_idle(). The implementation is present in `arch/arm/core/cortex_m/cpu_idle.c`.

In both implementations, the processor will attempt to put the core to low power mode. In k_cpu_idle() the processor ends up executing WFI (Wait For Interrupt) instruction, while in k_cpu_atomic_idle() the processor will execute a WFE (Wait For Event) instruction.

When using the CPU idling API in Cortex-M it is important to note the following:

- Both k_cpu_idle() and k_cpu_atomic_idle() are *assumed* to be invoked with interrupts locked. This is taken care of by the kernel if the APIs are called by the idle thread.

- After waking up from low power mode, both functions will *restore* interrupts unconditionally, that is, regardless of the interrupt lock status before the CPU idle API was called.

The Zephyr CPU Idling mechanism is detailed in *CPU Idling*.

Memory protection features This section describes certain aspects around memory protection features in Arm Cortex-M applications.

User mode system calls User mode is supported in Cortex-M platforms that implement the standard (Arm) MPU or a similar core peripheral logic for memory access policy configuration and control, such as the NXP MPU for Kinetis platforms. (Currently, `CONFIG_ARCH_HAS_USERSPACE` is selected if `CONFIG_ARM_MPU` is enabled by the user in the board default Kconfig settings).

A thread performs a system call by triggering a (synchronous) SVC exception, where

- up to 5 arguments are placed on registers R1 - R5
- system call ID is placed on register R6.

The SVC Handler will branch to the system call preparation logic, which will perform the following operations

- switch the thread's PSP to point to the beginning of the thread's privileged stack area, optionally reprogramming the PSPLIM if stack limit checking is enabled
- modify CONTROL register to switch to privileged mode
- modify the return address in the SVC exception stack frame, so that after exception return the system call dispatcher is executed (in thread privileged mode)

Once the system call execution is completed the system call dispatcher will restore the user's original PSP and PSPLIM and switch the CONTROL register back to unprivileged mode before returning back to the caller of the system call.

System calls execute in thread mode and can be preempted by interrupts at any time. A thread may also be context-switched-out while doing a system call; the system call will resume as soon as the thread is switched-in again.

The system call dispatcher executes at SVC priority, therefore it cannot be preempted by HW interrupts (with the exception of ZLIs), which may observe some additional interrupt latency if they occur during a system call preparation.

MPU-assisted stack overflow detection Cortex-M platforms with MPU may enable `CONFIG_MPU_STACK_GUARD` to enable the MPU-based stack overflow detection mechanism. The following points need to be considered when enabling the MPU stack guards

- stack overflows are triggering processor faults as soon as they occur
- the mechanism is essential for detecting stack overflows in supervisor threads, or user threads in privileged mode; stack overflows in threads in user mode will always be detected regardless of `CONFIG_MPU_STACK_GUARD` being set.
- stack overflows are always detected, however, the mechanism does not guarantee that no memory corruption occurs when supervisor threads overflow their stack memory
- `CONFIG_MPU_STACK_GUARD` will normally reserve one MPU region for programming the stack guard (in certain Arm v8-M configurations with `CONFIG_MPU_GAP_FILLING` enabled 2 MPU regions are required to implement the guard feature)
- MPU guards are re-programmed at every context-switch, adding a small overhead to the thread swap routine. Compared, however, to the `CONFIG_BUILTIN_STACK_GUARD` feature, no re-programming occurs during system calls.

- When CONFIG_HW_STACK_PROTECTION is enabled on Arm v8-M platforms the native stack limit checking mechanism is used by default instead of the MPU-based stack overflow detection mechanism; users may override this setting by manually enabling CONFIG_MPU_STACK_GUARD in these scenarios.

Memory map and MPU considerations

Fixed MPU regions By default, when CONFIG_ARM_MPU is enabled a set of *fixed* MPU regions are programmed during system boot.

- One MPU region programs the entire flash area as read-execute. User can override this setting by enabling CONFIG_MPU_ALLOW_FLASH_WRITE, which programs the flash with RWX permissions. If CONFIG_USERSPACE is enabled unprivileged access on the entire flash area is allowed.
- One MPU region programs the entire SRAM area with privileged-only RW permissions. That is, an MPU region is utilized to disallow execute permissions on SRAM. (An exception to this setting is when CONFIG_MPU_GAP_FILLING is disabled (Arm v8-M only); in that case no SRAM MPU programming is done so the access is determined by the default Arm memory map policies, allowing for privileged-only RWX permissions on SRAM).
- All the memory regions defined in the devicetree with the property zephyr,memory-attr defining the MPU permissions for the memory region. See the next section for more details.

The above MPU regions are defined in arch/arm/core/mpu/arm_mpu_regions.c. Alternative MPU configurations are allowed by enabling CONFIG_CPU_HAS_CUSTOM_FIXED_SOC_MPU_REGIONS. When enabled, this option signifies that the Cortex-M SoC will define and configure its own fixed MPU regions in the SoC definition.

Fixed MPU regions defined in devicetree When the property zephyr,memory-attr is present in a memory node, a new MPU region will be allocated and programmed during system boot. When used with the zephyr,memory-region devicetree compatible, it will result in a linker section being generated associated to that MPU region.

For example, to define a new non-cacheable memory region in devicetree:

```
sram_no_cache: memory@20300000 {
     compatible = "zephyr,memory-region", "mmio-sram";
     reg = <0x20300000 0x100000>;
     zephyr,memory-region = "SRAM_NO_CACHE";
     zephyr,memory-attr = <( DT_MEM_ARM(ATTR_MPU_RAM_NOCACHE) )>;
};
```

This will automatically create a new MPU entry in with the correct name, base, size and attributes gathered directly from the devicetree.

Static MPU regions Additional *static* MPU regions may be programmed once during system boot. These regions are required to enable certain features

- a RX region to allow execution from SRAM, when CONFIG_ARCH_HAS_RAMFUNC_SUPPORT is enabled and users have defined functions to execute from SRAM.
- a RX region for relocating text sections to SRAM, when CONFIG_CODE_DATA_RELOCATION_SRAM is enabled
- a no-cache region to allow for a none-cacheable SRAM area, when CONFIG_NOCACHE_MEMORY is enabled
- a possibly unprivileged RW region for GCOV code coverage accounting area, when CONFIG_COVERAGE_GCOV is enabled

- a no-access region to implement null pointer dereference detection, when CONFIG_NULL_POINTER_EXCEPTION_DETECTION_MPU is enabled

The boundaries of these static MPU regions are derived from symbols exposed by the linker, in include/linker/linker-defs.h.

Dynamic MPU regions Certain thread-specific MPU regions may be re-programmed dynamically, at each thread context switch:

- an unprivileged RW region for the current thread's stack area (for user threads)
- a read-only region for the MPU stack guard
- unprivileged RW regions for the partitions of the current thread's application memory domain.

Considerations The number of available MPU regions for a Cortex-M platform is a limited resource. Most platforms have 8 MPU regions, while some Cortex-M33 or Cortex-M7 platforms may have up to 16 MPU regions. Therefore there is a relatively strict limitation on how many fixed, static and dynamic MPU regions may be programmed simultaneously. For platforms with 8 available MPU regions it might not be possible to enable all the aforementioned features that require MPU region programming. In most practical applications, however, only a certain set of features is required and 8 MPU regions are, in many cases, sufficient.

In Arm v8-M processors the MPU architecture does not allow programmed MPU regions to overlap. CONFIG_MPU_GAP_FILLING controls whether the fixed MPU region covering the entire SRAM is programmed. When it does, a full SRAM area partitioning is required, in order to program the static and the dynamic MPU regions. This increases the total number of required MPU regions. When CONFIG_MPU_GAP_FILLING is not enabled the fixed MPU region covering the entire SRAM is not programmed, thus, the static and dynamic regions are simply programmed on top of the always-existing background region (full-SRAM partitioning is not required). Note, however, that the background SRAM region allows execution from SRAM, so when CONFIG_MPU_GAP_FILLING is not set Zephyr is not protected against attacks that attempt to execute malicious code from SRAM.

Floating point Services Both unshared and shared FP registers mode are supported in Cortex-M (see *Floating Point Services* for more details).

When FPU support is enabled in the build (CONFIG_FPU is enabled), the sharing FP registers mode (CONFIG_FPU_SHARING) is enabled by default. This is done as some compiler configurations may activate a floating point context by generating FP instructions for any thread, regardless of whether floating point calculations are performed, and that context must be preserved when switching such threads in and out.

The developers can still disable the FP sharing mode in their application projects, and switch to Unshared FP registers mode, if it is guaranteed that the image code does not generate FP instructions outside the single thread context that is allowed (and supposed) to do so.

Under FPU sharing mode, the callee-saved FPU registers are saved and restored in context-switch, if the corresponding threads have an active FP context. This adds some runtime overhead on the swap routine. In addition to the runtime overhead, the sharing FPU mode

- requires additional memory for each thread to save the callee-saved FP registers
- requires additional stack memory for each thread, to stack the caller-saved FP registers, upon exception entry, if an FP context is active. Note, however, that since lazy stacking is enabled, there is no runtime overhead of FP context stacking in regular interrupts (FP state preservation is only activated in the swap routine in PendSV interrupt).

Misc

Chain-loadable images Cortex-M applications may either be standalone images or chain-loadable, for instance, by a bootloader. Application images chain-loadable by bootloaders (or other applications) normally occupy a specific area in the flash denoted as their *code partition*. `CONFIG_USE_DT_CODE_PARTITION` will ensure that a Zephyr chain-loadable image will be linked into its code partition, specified in DeviceTree.

HW initialization at boot In order to boot properly, chain-loaded applications may require that the core Arm hardware registers and peripherals are initialized in their reset values. Enabling `CONFIG_INIT_ARCH_HW_AT_BOOT` Zephyr to force the initialization of the internal Cortex-M architectural state during boot to the reset values as specified by the corresponding Arm architecture manual.

Software vector relaying In Cortex-M platforms that implement the VTOR register (see `CONFIG_CPU_CORTEX_M_HAS_VTOR`), chain-loadable images relocate the Cortex-M vector table by updating the VTOR register with the offset of the image vector table.

Baseline Cortex-M platforms without VTOR register might not be able to relocate their vector table which remains at a fixed location. Therefore, a chain-loadable image will require an alternative way to route HW interrupts and system exceptions to its own vector table; this is achieved with software vector relaying.

When a bootloader image enables `CONFIG_SW_VECTOR_RELAY` it is able to relay exceptions and interrupts based on a vector table pointer that is set by the chain-loadable application. The latter sets the `CONFIG_SW_VECTOR_RELAY_CLIENT` option to instruct the boot sequence to set the vector table pointer in SRAM so that the bootloader can forward the exceptions and interrupts to the chain-loadable image's software vector table.

While this feature is intended for processors without VTOR register, it may also be used in Mainline Cortex-M platforms.

Code relocation Cortex-M support the code relocation feature. When `CONFIG_CODE_DATA_RELOCATION_SRAM` is selected, Zephyr will relocate .text, data and .bss sections from the specified files and place it in SRAM. It is possible to relocate only parts of the code sections into SRAM, without relocating the whole image text and data sections. More details on the code relocation feature can be found in *Code And Data Relocation*.

Linking Cortex-M applications

Most Cortex-M platforms make use of the default Cortex-M GCC linker script in `include/zephyr/arch/arm/cortex_m/scripts/linker.ld`, although it is possible for platforms to use a custom linker script as well.

CMSIS

Cortex-M CMSIS headers are hosted in a standalone module repository: zephyrproject-rtos/cmsis.

`CONFIG_CPU_CORTEX_M` selects `CONFIG_HAS_CMSIS_CORE` to signify that CMSIS headers are available for all supported Cortex-M variants.

Testing

A list of unit tests for the Cortex-M porting and miscellaneous features is present in `tests/arch/arm/`. The tests suites are continuously extended and new test suites are added, in an effort to increase the coverage of the Cortex-M architecture support in Zephyr.

QEMU

We use QEMU to verify the implemented features of the Cortex-M architecture port in Zephyr. Adequate coverage is achieved by defining and utilizing a list of QEMU targets, each with a specific architecture variant and Arm peripheral support list.

The table below lists the QEMU platform targets defined in Zephyr along with the corresponding Cortex-M implementation variant and the peripherals these targets emulate.

	QEMU target				
Architecture variant	Arm v6-M	Arm v7-M		Arm v8-M	Arm v8.1-M
	qemu_cortex_m	**qemu_cortex_m**	**mps2_an38**	**mps2_an52**	**mps3_an547**
Emulated features					
NVIC	Y	Y	Y	Y	Y
BASEPRI	N	Y	Y	Y	Y
SysTick	N	Y	Y	Y	Y
MPU	N	N	Y	Y	Y
FPU	N	N	N	Y	N
SPLIM	N	N	N	Y	Y
TrustZone-M	N	N	N	Y	N

Maintainers & Collaborators

The status of the Arm Cortex-M architecture port in Zephyr is: *maintained*. The updated list of maintainers and collaborators for Cortex-M can be found in `MAINTAINERS.yml`.

7.1.3 Zephyr support status on RISC-V processors

Overview

This page describes current state of Zephyr for RISC-V processors. Currently, there's support for some boards, as well as Qemu support and support for some FPGA implementations such as neorv32 and litex_vexriscv.

Zephyr support includes PMP, *user mode*, several ISA extensions as well as *semihosting*.

User mode and PMP support

When the platform has Physical Memory Protection (PMP) support, enabling it on Zephyr allows user space support and stack protection to be selected.

ISA extensions

It's possible to set in Zephyr which ISA extensions (RV32/64I(E)MAFD(G)QC) are available on a given platform, by setting the appropriate *RISCV_ISA_** kconfig. Look at `arch/riscv/Kconfig.isa` for more information.

Note that Zephyr SDK toolchain support may not be defined for all combinations.

SMP support

SMP is supported on RISC-V, but currently only on Qemu platforms. In order to test the SMP support, one can use *qemu_riscv32_smp* or *qemu_riscv64_smp* boards.

7.1.4 Semihosting Guide

Overview

Semihosting is a mechanism that enables code running on ARM and RISC-V targets to communicate and use the Input/Output facilities on a host computer that is running a debugger or emulator.

More complete documentation on the available functionality is available at the ARM Github documentation.

The RISC-V functionality borrows from the ARM definitions, as described at the RISC-V Github documentation.

File Operations

Semihosting enables files on the host computer to be opened, read, and modified by an application. This can be useful when attempting to validate the behaviour of code across datasets that are larger than what can fit into ROM of an emulated platform. File paths can be either absolute, or relative to the directory of the running process.

```
const char *path = "./data.bin";
long file_len, bytes_read, fd;
uint8_t buffer[16];

/* Open the data file for reading */
fd = semihost_open(path, SEMIHOST_OPEN_RB);
if (fd < 0) {
   return -ENOENT;
}
/* Read all data from the file */
file_len = semihost_flen(fd);
while(file_len > 0) {
   bytes_read = semihost_read(fd, buffer, MIN(file_len, sizeof(buffer)));
   if (bytes_read < 0) {
      break;
   }
   /* Process read data */
   do_data_processing(buffer, bytes_read);
   /* Update remaining length */
   file_len -= bytes_read;
}
/* Close the file */
semihost_close(fd);
```

7.1.5 Additional Functionality

Additional functionality is available by running semihosting instructions directly with *semihost_exec()* with one of the instructions defined in *semihost_instr*. For complete documentation on the required arguments and return codes, see the ARM Github documentation.

API Reference

group **semihost**

Enums

enum **semihost_instr**

Semihosting instructions.

Values:

enumerator **SEMIHOST_OPEN** = 0x01

Open a file or stream on the host system.

enumerator **SEMIHOST_ISTTY** = 0x09

Check whether a file is associated with a stream/terminal.

enumerator **SEMIHOST_WRITE** = 0x05

Write to a file or stream.

enumerator **SEMIHOST_READ** = 0x06

Read from a file at the current cursor position.

enumerator **SEMIHOST_CLOSE** = 0x02

Closes a file on the host which has been opened by SEMIHOST_OPEN.

enumerator **SEMIHOST_FLEN** = 0x0C

Get the length of a file.

enumerator **SEMIHOST_SEEK** = 0x0A

Set the file cursor to a given position in a file.

enumerator **SEMIHOST_TMPNAM** = 0x0D

Get a temporary absolute file path to create a temporary file.

enumerator **SEMIHOST_REMOVE** = 0x0E

Remove a file on the host system.

Possibly insecure!

enumerator **SEMIHOST_RENAME** = 0x0F

Rename a file on the host system.

Possibly insecure!

enumerator SEMIHOST_WRITEC = 0x03
Write one character to the debug terminal.

enumerator SEMIHOST_WRITE0 = 0x04
Write a NULL terminated string to the debug terminal.

enumerator SEMIHOST_READC = 0x07
Read one character from the debug terminal.

enumerator SEMIHOST_CLOCK = 0x10

enumerator SEMIHOST_ELAPSED = 0x30

enumerator SEMIHOST_TICKFREQ = 0x31

enumerator SEMIHOST_TIME = 0x11

enumerator SEMIHOST_ERRNO = 0x13
Retrieve the errno variable from semihosting operations.

enumerator SEMIHOST_GET_CMDLINE = 0x15
Get commandline parameters for the application to run with.

enumerator SEMIHOST_HEAPINFO = 0x16

enumerator SEMIHOST_ISERROR = 0x08

enumerator SEMIHOST_SYSTEM = 0x12

enum semihost_open_mode
Modes to open a file with.

Behaviour corresponds to equivalent fopen strings. i.e. SEMIHOST_OPEN_RB_PLUS == "rb+"

Values:

enumerator SEMIHOST_OPEN_R = 0

enumerator SEMIHOST_OPEN_RB = 1

enumerator SEMIHOST_OPEN_R_PLUS = 2

enumerator SEMIHOST_OPEN_RB_PLUS = 3

enumerator SEMIHOST_OPEN_W = 4

enumerator SEMIHOST_OPEN_WB = 5

enumerator SEMIHOST_OPEN_W_PLUS = 6

enumerator SEMIHOST_OPEN_WB_PLUS = 7

enumerator SEMIHOST_OPEN_A = 8

enumerator SEMIHOST_OPEN_AB = 9

enumerator SEMIHOST_OPEN_A_PLUS = 10

enumerator SEMIHOST_OPEN_AB_PLUS = 11

Functions

long **semihost_exec**(enum *semihost_instr* instr, void *args)

Manually execute a semihosting instruction.

Parameters

- **instr** – instruction code to run
- **args** – instruction specific arguments

Returns

integer return code of instruction

char **semihost_poll_in**(void)

Read a byte from the console.

Returns

char byte read from the console.

void **semihost_poll_out**(char c)

Write a byte to the console.

Parameters

- **c** – byte to write to console

long **semihost_open**(const char *path, long mode)

Open a file on the host system.

Parameters

- **path** – file path to open. Can be absolute or relative to current directory of the running process.
- **mode** – value from *semihost_open_mode*.

Return values

- **handle** – positive handle on success.
- **-1** – on failure.

long **semihost_close**(long fd)

Close a file.

Parameters

- **fd** – handle returned by *semihost_open*.

Return values

- `0` – on success.
- `-1` – on failure.

long `semihost_flen`(long fd)

Query the size of a file.

Parameters

- `fd` – handle returned by *semihost_open*.

Return values

- `positive` – file size on success.
- `-1` – on failure.

long `semihost_seek`(long fd, long offset)

Seeks to an absolute position in a file.

Parameters

- `fd` – handle returned by *semihost_open*.
- `offset` – offset from the start of the file in bytes.

Return values

- `0` – on success.
- `-errno` – negative error code on failure.

long `semihost_read`(long fd, void *buf, long len)

Read the contents of a file into a buffer.

Parameters

- `fd` – handle returned by *semihost_open*.
- `buf` – buffer to read data into.
- `len` – number of bytes to read.

Return values

- `read` – number of bytes read on success.
- `-errno` – negative error code on failure.

long `semihost_write`(long fd, const void *buf, long len)

Write the contents of a buffer into a file.

Parameters

- `fd` – handle returned by *semihost_open*.
- `buf` – buffer to write data from.
- `len` – number of bytes to write.

Return values

- `0` – on success.
- `-errno` – negative error code on failure.

7.1.6 x86 Developer Guide

Overview

This page contains information on certain aspects when developing for x86-based platforms.

Virtual Memory

During very early boot, page tables are loaded so technically the kernel is executing in virtual address space. By default, physical and virtual memory are identity mapped and thus giving the appearance of execution taking place in physical address space. The physical address space is marked by kconfig `CONFIG_SRAM_BASE_ADDRESS` and `CONFIG_SRAM_SIZE` while the virtual address space is marked by `CONFIG_KERNEL_VM_BASE` and `CONFIG_KERNEL_VM_SIZE`. Note that `CONFIG_SRAM_OFFSET` controls where the Zephyr kernel is being placed in the memory, and its counterpart `CONFIG_KERNEL_VM_OFFSET`.

Separate Virtual Address Space from Physical Address Space On 32-bit x86, it is possible to have separate physical and virtual address space. Code and data are linked in virtual address space, but are still loaded in physical memory. However, during boot, code and data must be available and also addressable in physical address space before `vm_enter` inside `arch/x86/core/ia32/crt0.S`. After `vm_enter`, code execution is done via virtual addresses and data can be referred via their virtual addresses. This is possible as the page table generation script (`arch/x86/gen_mmu.py`) identity maps the physical addresses at the page directory level, in addition to mapping virtual addresses to the physical memory. Later in the boot process, the entries for identity mapping at the page directory level are cleared in `z_x86_mmu_init()`, effectively removing the identity mapping of physical memory. This unmapping must be done for userspace isolation or else they would be able to access restricted memory via physical addresses. Since the identity mapping is done at the page directory level, there is no need to allocate additional space for the page table. However, additional space may still be required for additional page directory table.

There are restrictions on where virtual address space can be:

- Physical and virtual address spaces must be disjoint. This is required as the entries in page directory table will be cleared. If they are not disjoint, it would clear the entries needed for virtual addresses.
  - If `CONFIG_X86_PAE` is enabled (`=y`), each address space must reside in their own 1GB region, due to each entry of PDP (Page Directory Pointer) covers 1GB of memory. For example:
    - Assuming `CONFIG_SRAM_OFFSET` and `CONFIG_KERNEL_VM_OFFSET` are both `0x0`.
    - `CONFIG_SRAM_BASE_ADDRESS == 0x00000000` and `CONFIG_KERNEL_VM_BASE = 0x40000000` is valid, while
    - `CONFIG_SRAM_BASE_ADDRESS == 0x00000000` and `CONFIG_KERNEL_VM_BASE = 0x20000000` is not.
  - If `CONFIG_X86_PAE` is disabled (`=n`), each address space must reside in their own 4MB region, due to each entry of PD (Page Directory) covers 4MB of memory.
  - Both `CONFIG_SRAM_BASE_ADDRESS` and `CONFIG_KERNEL_VM_BASE` must also align with the starting addresses of targeted regions.

Specifying Additional Memory Mappings at Build Time

The page table generation script (`arch/x86/gen_mmu.py`) generates the necessary multi-level page tables for code execution and data access using the kernel image produced by the first linker pass. Additional command line arguments can be passed to the script to generate additional

memory mappings. This is useful for static mappings and/or device MMIO access during very early boot. To pass extra command line arguments to the script, populate a CMake list named `X86_EXTRA_GEN_MMU_ARGUMENTS` in the board configuration file. Here is an example:

```
set(X86_EXTRA_GEN_MMU_ARGUMENTS
    --map 0xA0000000,0x2000
    --map 0x80000000,0x400000,LWUX,0xB0000000)
```

The argument `--map` takes the following value: `<physical address>,<size>[,<flags:LUWX>[,<virtual address>]]`, where:

- `<physical address>` is the physical address of the mapping. (Required)
- `<size>` is the size of the region to be mapped. (Required)
- `<flags>` is the flag associated with the mapping: (Optional)
  - `L`: Large page at the page directory level.
  - `U`: Allow userspace access.
  - `W`: Read/write.
  - `X`: Allow execution.
  - `D`: Cache disabled.
    - Default is small page (4KB), supervisor only, read only, and execution disabled.
- `<virtual address` is the virtual address of the mapping. (Optional)

Note that specifying additional memory mappings requires larger storage space for the pre-allocated page tables (both kernel and per-domain tables). `CONFIG_X86_EXTRA_PAGE_TABLE_PAGES` is needed to specify how many more memory pages to be reserved for the page tables. If the needed space is not exactly the same as required space, the `gen_mmu.py` script will print out a message indicating what needs to be the value for the kconfig.

7.1.7 Xtensa Developer Guide

Overview

This page contains information on certain aspects when developing for Xtensa-based platforms.

HiFi Audio Engine DSP

The kernel allows threads to use the HiFi Audio Engine DSP registers on boards that support these registers. The kernel only supports the use of the HiFi registers by threads and not ISRs.

> **Note**
>
> Presently, only the Intel ADSP ACE hardware platforms are configured for HiFi support by default.

Concepts The kernel can be configured for an application to leverage the services provided by the Xtensa HiFi Audio Engine DSP. Three modes of operation are supported, which are described below.

No HiFi registers mode This mode is used when the application has no threads that use the HiFi registers. It is the kernel's default HiFi services mode.

Unshared HiFi registers mode This mode is used when the application has only a single thread that uses the HiFi registers. The HiFi registers are left unchanged whenever a context switch occurs.

> **Note**
>
> The behavior is undefined, if two or more threads attempt to use the HiFi registers, as the kernel does not attempt to detect (nor prevent) multiple threads from using these registers.

Shared HiFi registers mode This mode is used when the application has two or more threads that use HiFi registers. When enabled, the kernel automatically allows all threads to use the HiFi registers. During each thread context switch, the kernel saves the outgoing thread's HiFi registers and loads the incoming thread's HiFi registers, regardless of whether the thread utilizes them or not.

Additional stack space may be required for each thread to account for the extra registers that must be saved.

Configuration Options The unshared HiFi registers mode is selected when configuration option `CONFIG_XTENSA_HIFI_SHARING` is disabled but configuration options `CONFIG_XTENSA_HIFI3` and/or `CONFIG_XTENSA_HIFI4` are enabled.

The shared HiFi registers mode is selected when the configuration option `CONFIG_XTENSA_HIFI_SHARING` is enabled in addition to configuration options `CONFIG_XTENSA_HIFI3` and/or `CONFIG_XTENSA_HIFI4`. Threads must have sufficient stack space for saving the HiFi register values during context switches as described above.

7.2 Barriers API

group `barrier_apis`

Since
3.4

Version
0.1.0

Functions

ALWAYS_INLINE static void **`barrier_dmem_fence_full`**(void)

Full/sequentially-consistent data memory barrier.

This routine acts as a synchronization fence between threads and prevents re-ordering of data accesses instructions across the barrier instruction.

ALWAYS_INLINE static void **barrier_dsync_fence_full**(void)

Full/sequentially-consistent data synchronization barrier.

This routine acts as a synchronization fence between threads and prevents reordering of data accesses instructions across the barrier instruction like *barrier_dmem_fence_full()*, but has the additional effect of blocking execution of any further instructions, not just loads or stores, or both, until synchronization is complete.

> **Note**
>
> When not supported by hardware or architecture, this instruction falls back to a full/sequentially-consistent data memory barrier.

ALWAYS_INLINE static void **barrier_isync_fence_full**(void)

Full/sequentially-consistent instruction synchronization barrier.

This routine is used to guarantee that any subsequent instructions are fetched and to ensure any previously executed context-changing operations, such as writes to system control registers, have completed by the time the routine completes. In hardware terms, this might mean that the instruction pipeline is flushed, for example.

> **Note**
>
> When not supported by hardware or architecture, this instruction falls back to a compiler barrier.

7.3 Cache Interface

This is a high-level guide to cache interface and Kconfig options related to cache controllers. See *Cache API* for API reference material.

Zephyr has different Kconfig options to control how the cache controller is implemented and controlled.

- `CONFIG_CPU_HAS_DCACHE` / `CONFIG_CPU_HAS_ICACHE`: these hidden options should be selected at SoC / platform level when the CPU actually supports a data or instruction cache. The cache controller can be in the core or can be an external cache controller for which a driver is provided.

  These options have the goal to document an available feature and should be set whether we plan to support and use the caches in Zephyr or not.

- `CONFIG_DCACHE` / `CONFIG_ICACHE`: these options must be selected when support for data or instruction cache is present and working in zephyr.

  All the code paths related to cache control must be conditionally enabled depending on these symbols. When the symbol is set the cache is considered enabled and used.

  These symbols say nothing about the actual API interface exposed to the user. For example a platform using the data cache can enable the `CONFIG_DCACHE` symbol and use some HAL exported function in some platform-specific code to enable and manage the d-cache.

- `CONFIG_CACHE_MANAGEMENT`: this option must be selected when the cache operations are exposed to the user through a standard API (see *Cache API*).

  When this option is enabled we assume that all the cache functions are implemented in the architectural code or in an external cache controller driver.

- CONFIG_ARCH_CACHE/CONFIG_EXTERNAL_CACHE: mutually exclusive options for CACHE_TYPE used to define whether the cache operations are implemented at arch level or using an external cache controller with a provided driver.
  - CONFIG_ARCH_CACHE: the cache API is implemented by the arch code
  - CONFIG_EXTERNAL_CACHE: the cache API is implemented by a driver that supports the external cache controller. In this case the driver must be located as usual in the drivers/cache/ directory

7.3.1 Cache API

group cache_interface

Functions

ALWAYS_INLINE static void **sys_cache_data_enable**(void)

Enable the d-cache.

Enable the data cache

ALWAYS_INLINE static void **sys_cache_data_disable**(void)

Disable the d-cache.

Disable the data cache

ALWAYS_INLINE static void **sys_cache_instr_enable**(void)

Enable the i-cache.

Enable the instruction cache

ALWAYS_INLINE static void **sys_cache_instr_disable**(void)

Disable the i-cache.

Disable the instruction cache

ALWAYS_INLINE static int **sys_cache_data_flush_all**(void)

Flush the d-cache.

Flush the whole data cache.

Return values

- **0** – If succeeded.
- **-ENOTSUP** – If not supported.
- **-errno** – Negative errno for other failures.

ALWAYS_INLINE static int **sys_cache_instr_flush_all**(void)

Flush the i-cache.

Flush the whole instruction cache.

Return values

- **0** – If succeeded.
- **-ENOTSUP** – If not supported.
- **-errno** – Negative errno for other failures.

ALWAYS_INLINE static int **sys_cache_data_invd_all**(void)

Invalidate the d-cache.

Invalidate the whole data cache.

Return values

- `0` – If succeeded.
- `-ENOTSUP` – If not supported.
- `-errno` – Negative errno for other failures.

ALWAYS_INLINE static int **sys_cache_instr_invd_all**(void)

Invalidate the i-cache.

Invalidate the whole instruction cache.

Return values

- `0` – If succeeded.
- `-ENOTSUP` – If not supported.
- `-errno` – Negative errno for other failures.

ALWAYS_INLINE static int **sys_cache_data_flush_and_invd_all**(void)

Flush and Invalidate the d-cache.

Flush and Invalidate the whole data cache.

Return values

- `0` – If succeeded.
- `-ENOTSUP` – If not supported.
- `-errno` – Negative errno for other failures.

ALWAYS_INLINE static int **sys_cache_instr_flush_and_invd_all**(void)

Flush and Invalidate the i-cache.

Flush and Invalidate the whole instruction cache.

Return values

- `0` – If succeeded.
- `-ENOTSUP` – If not supported.
- `-errno` – Negative errno for other failures.

int **sys_cache_data_flush_range**(void *addr, size_t size)

Flush an address range in the d-cache.

Flush the specified address range of the data cache.

> **Note**
>
> the cache operations act on cache line. When multiple data structures share the same cache line being flushed, all the portions of the data structures sharing the same line will be flushed. This is usually not a problem because writing back is a non-destructive process that could be triggered by hardware at any time, so having an aligned `addr` or a padded `size` is not strictly necessary.

Parameters

- `addr` – Starting address to flush.

- **size** – Range size.

Return values

- **0** – If succeeded.
- **-ENOTSUP** – If not supported.
- **-errno** – Negative errno for other failures.

ALWAYS_INLINE static int **sys_cache_instr_flush_range**(void *addr, size_t size)

Flush an address range in the i-cache.

Flush the specified address range of the instruction cache.

> **Note**
>
> the cache operations act on cache line. When multiple data structures share the same cache line being flushed, all the portions of the data structures sharing the same line will be flushed. This is usually not a problem because writing back is a non-destructive process that could be triggered by hardware at any time, so having an aligned `addr` or a padded `size` is not strictly necessary.

Parameters

- **addr** – Starting address to flush.
- **size** – Range size.

Return values

- **0** – If succeeded.
- **-ENOTSUP** – If not supported.
- **-errno** – Negative errno for other failures.

int **sys_cache_data_invd_range**(void *addr, size_t size)

Invalidate an address range in the d-cache.

Invalidate the specified address range of the data cache.

> **Note**
>
> the cache operations act on cache line. When multiple data structures share the same cache line being invalidated, all the portions of the non-read-only data structures sharing the same line will be invalidated as well. This is a destructive process that could lead to data loss and/or corruption. When `addr` is not aligned to the cache line and/or `size` is not a multiple of the cache line size the behaviour is undefined.

Parameters

- **addr** – Starting address to invalidate.
- **size** – Range size.

Return values

- **0** – If succeeded.
- **-ENOTSUP** – If not supported.
- **-errno** – Negative errno for other failures.

ALWAYS_INLINE static int **sys_cache_instr_invd_range**(void *addr, size_t size)

Invalidate an address range in the i-cache.

Invalidate the specified address range of the instruction cache.

> **Note**
>
> the cache operations act on cache line. When multiple data structures share the same cache line being invalidated, all the portions of the non-read-only data structures sharing the same line will be invalidated as well. This is a destructive process that could lead to data loss and/or corruption. When `addr` is not aligned to the cache line and/or `size` is not a multiple of the cache line size the behaviour is undefined.

Parameters

- **addr** – Starting address to invalidate.
- **size** – Range size.

Return values

- **0** – If succeeded.
- **-ENOTSUP** – If not supported.
- **-errno** – Negative errno for other failures.

int **sys_cache_data_flush_and_invd_range**(void *addr, size_t size)

Flush and Invalidate an address range in the d-cache.

Flush and Invalidate the specified address range of the data cache.

> **Note**
>
> the cache operations act on cache line. When multiple data structures share the same cache line being flushed, all the portions of the data structures sharing the same line will be flushed before being invalidated. This is usually not a problem because writing back is a non-destructive process that could be triggered by hardware at any time, so having an aligned `addr` or a padded `size` is not strictly necessary.

Parameters

- **addr** – Starting address to flush and invalidate.
- **size** – Range size.

Return values

- **0** – If succeeded.
- **-ENOTSUP** – If not supported.
- **-errno** – Negative errno for other failures.

ALWAYS_INLINE static int **sys_cache_instr_flush_and_invd_range**(void *addr, size_t size)

Flush and Invalidate an address range in the i-cache.

Flush and Invalidate the specified address range of the instruction cache.

Note

the cache operations act on cache line. When multiple data structures share the same cache line being flushed, all the portions of the data structures sharing the same line will be flushed before being invalidated. This is usually not a problem because writing back is a non-destructive process that could be triggered by hardware at any time, so having an aligned `addr` or a padded `size` is not strictly necessary.

Parameters

- **addr** – Starting address to flush and invalidate.
- **size** – Range size.

Return values

- **0** – If succeeded.
- **-ENOTSUP** – If not supported.
- **-errno** – Negative errno for other failures.

ALWAYS_INLINE static size_t **sys_cache_data_line_size_get**(void)

Get the d-cache line size.

The API is provided to get the data cache line.

The cache line size is calculated (in order of priority):

- At run-time when `CONFIG_DCACHE_LINE_SIZE_DETECT` is set.
- At compile time using the value set in `CONFIG_DCACHE_LINE_SIZE` .
- At compile time using the `d-cache-line-size` CPU0 property of the DT.
- 0 otherwise

Return values

- **size** – Size of the d-cache line.
- **0** – If the d-cache is not enabled.

ALWAYS_INLINE static size_t **sys_cache_instr_line_size_get**(void)

Get the i-cache line size.

The API is provided to get the instruction cache line.

The cache line size is calculated (in order of priority):

- At run-time when `CONFIG_ICACHE_LINE_SIZE_DETECT` is set.
- At compile time using the value set in `CONFIG_ICACHE_LINE_SIZE` .
- At compile time using the `i-cache-line-size` CPU0 property of the DT.
- 0 otherwise

Return values

- **size** – Size of the d-cache line.
- **0** – If the d-cache is not enabled.

ALWAYS_INLINE static bool **sys_cache_is_ptr_cached**(void *ptr)

Test if a pointer is in cached region.

Some hardware may map the same physical memory twice so that it can be seen in both (incoherent) cached mappings and a coherent "shared" area. This tests if a particular pointer is within the cached, coherent area.

Parameters

- **ptr** – Pointer

Return values

- **True** – if pointer is in cached region.
- **False** – if pointer is not in cached region.

ALWAYS_INLINE static bool **sys_cache_is_ptr_uncached**(void *ptr)

Test if a pointer is in un-cached region.

Some hardware may map the same physical memory twice so that it can be seen in both (incoherent) cached mappings and a coherent "shared" area. This tests if a particular pointer is within the un-cached, incoherent area.

Parameters

- **ptr** – Pointer

Return values

- **True** – if pointer is not in cached region.
- **False** – if pointer is in cached region.

ALWAYS_INLINE static void ***sys_cache_cached_ptr_get**(void *ptr)

Return cached pointer to a RAM address.

This function takes a pointer to any addressable object (either in cacheable memory or not) and returns a pointer that can be used to refer to the same memory through the L1 data cache. Data read through the resulting pointer will reflect locally cached values on the current CPU if they exist, and writes will go first into the cache and be written back later.

See also

arch_uncached_ptr()

Note

This API returns the same pointer if CONFIG_CACHE_DOUBLEMAP is not enabled.

Parameters

- **ptr** – A pointer to a valid C object

Returns

A pointer to the same object via the L1 dcache

ALWAYS_INLINE static void *sys_cache_uncached_ptr_get(void *ptr)

Return uncached pointer to a RAM address.

This function takes a pointer to any addressable object (either in cacheable memory or not) and returns a pointer that can be used to refer to the same memory while bypassing the L1 data cache. Data in the L1 cache will not be inspected nor modified by the access.

See also

arch_cached_ptr()

Note

This API returns the same pointer if CONFIG_CACHE_DOUBLEMAP is not enabled.

Parameters

- `ptr` – A pointer to a valid C object

Returns

A pointer to the same object bypassing the L1 dcache

7.4 Zephyr's device emulators/simulators

7.4.1 Overview

Zephyr includes in its codebase a set of device emulators/simulators. With this we refer to SW components which are built together with the embedded SW and present themselves as devices of a given class to the rest of the system.

These device emulators/simulators can be built for any target which has sufficient RAM and flash, even if some may have extra functionality which is only available in some targets.

Note

Zephyr also includes and uses many other types of simulators/emulators, including CPU and platform simulators, radio simulators, and several build targets which allow running the embedded code in the development host.
Some of Zephyr communication controllers/drivers include also either loopback modes or loopback devices.
This page does not cover any of these.

Note

Drivers which are specific to some platform, like for example the native_sim specific drivers which emulate a peripheral class by connecting to host APIs are not covered by this page.

7.4.2 Available Emulators

ADC emulator

- A fake driver which pretends to be actual ADC, and can be used for testing higher-level API for ADC devices.
- Main Kconfig option: `CONFIG_ADC_EMUL`
- DT binding: `zephyr,adc-emul`

DMA emulator

- Emulated DMA controller
- Main Kconfig option: `CONFIG_DMA_EMUL`
- DT binding: `zephyr,dma-emul`

EEPROM emulator

- Emulate an EEPROM on a flash partition
- Main Kconfig option: `CONFIG_EEPROM_EMULATOR`
- DT binding: `zephyr,emu-eeprom`

EEPROM simulator

- Emulate an EEPROM on RAM
- Main Kconfig option: `CONFIG_EEPROM_SIMULATOR`
- DT binding: `zephyr,sim-eeprom`
- Note: For native targets it is also possible to keep the content as a file on the host filesystem.

External bus and bus connected peripheral emulators

- *Documentation*
- Allow emulating external buses like I2C or SPI and peripherals connected to them.

Flash simulator

- Emulate a flash on RAM
- Main Kconfig option: `CONFIG_FLASH_SIMULATOR`
- DT binding: `zephyr,sim-flash`
- Note: For native targets it is also possible to keep the content as a file on the host filesystem. Check the native_sim flash simulator section.

GPIO emulator

- Emulated GPIO controllers which can be driven from SW
- Main Kconfig option: `CONFIG_GPIO_EMUL`
- DT binding: `zephyr,gpio-emul`

I2C emulator

- Emulated I2C bus. See *bus emulators*.
- Main Kconfig option: `CONFIG_I2C_EMUL`
- DT binding: `zephyr,i2c-emul-controller`

RTC emulator

- Emulated RTC peripheral. See *RTC emulated device section*

- Main Kconfig option: `CONFIG_RTC_EMUL`
- DT binding: `zephyr,rtc-emul`

SPI emulator

- Emulated SPI bus. See *bus emulators*.
- Main Kconfig option: `CONFIG_SPI_EMUL`
- DT binding: `zephyr,spi-emul-controller`

MSPI emulator

- Emulated MSPI bus. See *bus emulators*.
- Main Kconfig option: `CONFIG_MSPI_EMUL`
- DT binding: `zephyr,mspi-emul-controller`

UART emulator

- Emulated UART bus. See *bus emulators*.
- Main Kconfig option: `CONFIG_UART_EMUL`
- DT binding: `zephyr,uart-emul`

7.5 External Bus and Bus Connected Peripherals Emulators

7.5.1 Overview

Zephyr supports a simple emulator framework to support testing of external peripheral drivers without requiring real hardware.

Emulators are used to emulate external hardware devices, to support testing of various subsystems. For example, it is possible to write an emulator for an I2C compass such that it appears on the I2C bus and can be used just like a real hardware device.

Emulators often implement special features for testing. For example a compass may support returning bogus data if the I2C bus speed is too high, or may return invalid measurements if calibration has not yet been completed. This allows for testing that high-level code can handle these situations correctly. Test coverage can therefore approach 100% if all failure conditions are emulated.

7.5.2 Concept

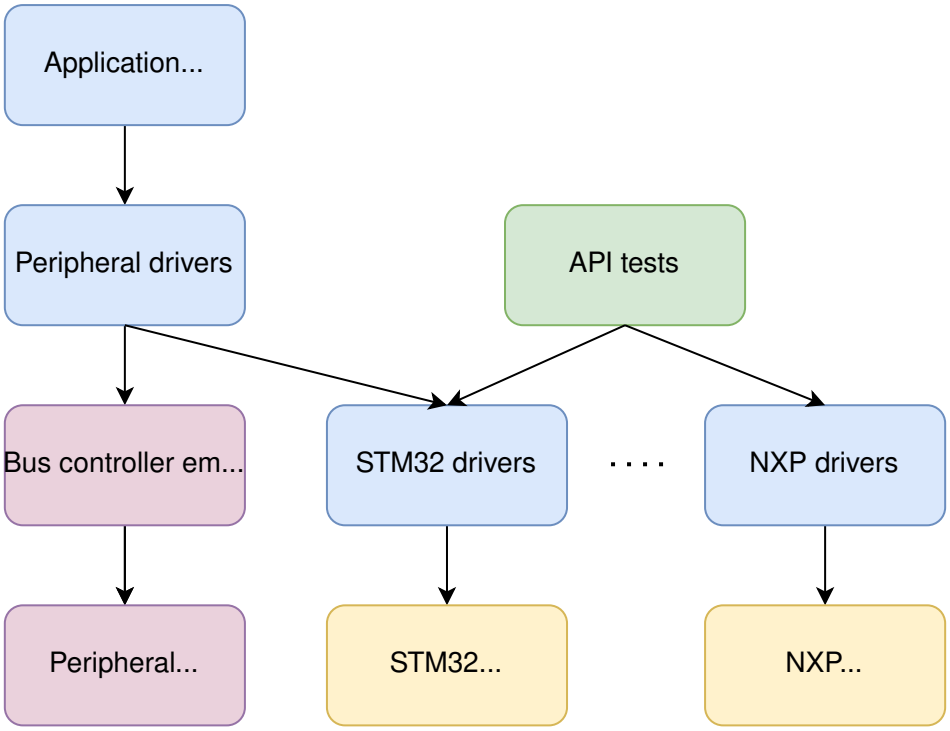
The diagram below shows application code / high-level tests at the top. This is the ultimate application we want to run.

Below that are peripheral drivers, such as the AT24 EEPROM driver. We can test peripheral drivers using an emulation driver connected via a emulated I2C controller/emulator which passes I2C traffic from the AT24 driver to the AT24 simulator.

Separately we can test the STM32 and NXP I2C drivers on real hardware using API tests. These require some sort of device attached to the bus, but with this, we can validate much of the driver functionality.

Putting the two together, we can test the application and peripheral code entirely on native_sim. Since we know that the I2C driver on the real hardware works, we should expect the application and peripheral drivers to work on the real hardware also.

Using the above framework we can test an entire application (e.g. Embedded Controller) on native_sim using emulators for all non-chip drivers.



With this approach we can:

- Write individual tests for each driver (green), covering all failure modes, error conditions, etc.
- Ensure 100% test coverage for drivers (green)
- Write tests for combinations of drivers, such as GPIOs provided by an I2C GPIO expander driver talking over an I2C bus, with the GPIOs controlling a charger. All of this can work in the emulated environment or on real hardware.
- Write a complex application that ties together all of these pieces and runs on native_sim. We can develop on a host, use source-level debugging, etc.
- Transfer the application to any board which provides the required features (e.g. I2C, enough GPIOs), by adding Kconfig and devicetree fragments.

7.5.3 Creating a Device Driver Emulator

The emulator subsystem is modeled on the *Device Driver Model*. You create an emulator instance using one of the *EMUL_DT_DEFINE()* or *EMUL_DT_INST_DEFINE()* APIs.

Emulators for peripheral devices reuse the same devicetree node as the real device driver. This means that your emulator defines *DT_DRV_COMPAT* using the same `compat` value from the real driver.

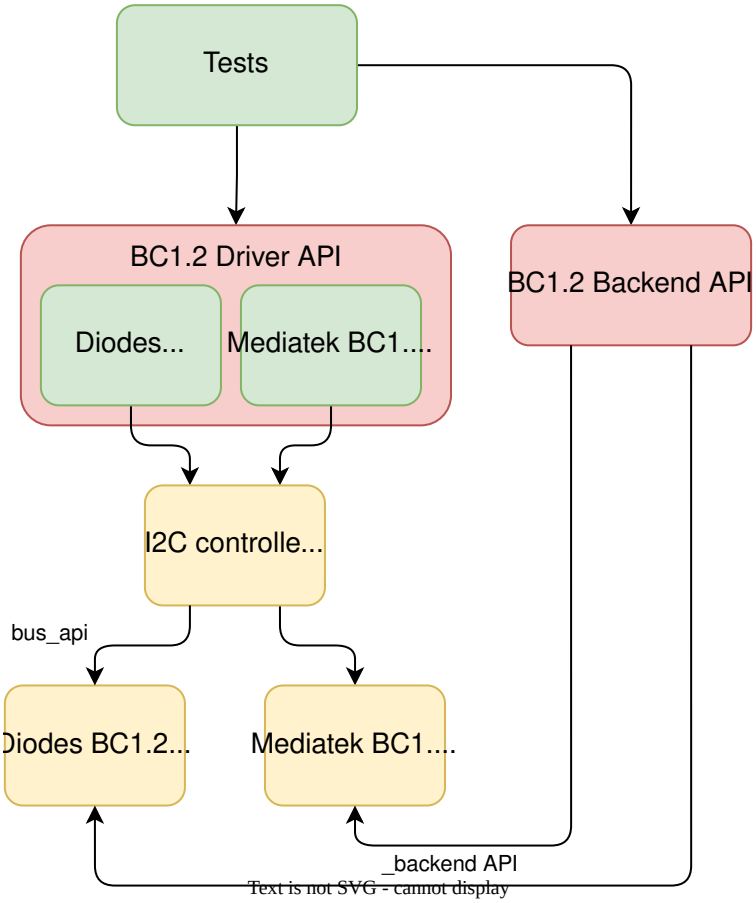
```
/* From drivers/sensor/bm160/bm160.c */
#define DT_DRV_COMPAT bosch_bmi160

/* From drivers/sensor/bmi160/emul_bmi160.c */
#define DT_DRV_COMPAT bosch_bmi160
```

The `EMUL_DT_DEFINE()` function accepts two API types:

1. `bus_api` - This points to the API for the upstream bus that the emulator connects to. The `bus_api` parameter is required. The supported emulated bus types include I2C, SPI, eSPI, and MSPI.
2. `_backend_api` - This points to the device-class specific backend API for the emulator. The `_backend_api` parameter is optional.

The diagram below demonstrates the logical organization of the `bus_api` and `_backend_api` using the BC1.2 charging detector driver as the model device-class.



The real code is shown in green, while the emulator code is shown in yellow.

The `bus_api` connects the BC1.2 emulators to the `native_sim` I2C controller. The real BC1.2 drivers are unchanged and operate exactly as if there was a physical I2C controller present in the system. The `native_sim` I2C controller uses the `bus_api` to initiate register reads and writes to the emulator.

The `_backend_api` provides a mechanism for tests to manipulate the emulator out of band. Each device class defines it's own API functions. The backend API functions focus on high-level behavior and do not provide hooks for specific emulators.

In the case of the BC1.2 charging detector the backend API provides functions to simulate connecting and disconnecting a charger to the emulated BC1.2 device. Each emulator is responsible for updating the correct vendor specific registers and potentially signalling an interrupt.

Example test flow:

1. Test registers BC1.2 detection callback using the Zephyr BC1.2 driver API.
2. Test connects a charger using the BC1.2 emulator backend.
3. Test verifies B1.2 detection callback invoked with correct charger type.
4. Test disconnects a charger using the BC1.2 emulator backend.

With this architecture, the same test can be used will all supported drivers in the same driver class.

7.5.4 Available Emulators

Zephyr includes the following emulators:

- I2C emulator driver, allowing drivers to be connected to an emulator so that tests can be performed without access to the real hardware
- SPI emulator driver, which does the same for SPI
- eSPI emulator driver, which does the same for eSPI. The emulator is being developed to support more functionalities.
- MSPI emulator driver, allowing drivers to be connected to an emulator so that tests can be performed without access to the real hardware.

7.5.5 Samples

Here are some examples present in Zephyr:

1. Bosch BMI160 sensor driver connected via both I2C and SPI to an emulator:

```
west build -b native_sim tests/drivers/sensor/accel/
```

2. The same test can be built with a second EEPROM which is an Atmel AT24 EEPROM driver connected via I2C an emulator:

```
west build -b native_sim tests/drivers/eeprom/api -- -DDTC_OVERLAY_FILE=at2x_emul.
↪overlay -DOVERLAY_CONFIG=at2x_emul.conf
```

API Reference

group `io_emulators`

Emulators used to test drivers and higher-level code that uses them.

Defines

`EMUL_DT_NAME_GET`(node_id)

Use the devicetree node identifier as a unique name.

Parameters

- `node_id` – A devicetree node identifier

`EMUL_DT_DEFINE`(node_id, init_fn, data_ptr, cfg_ptr, bus_api, _backend_api)

Define a new emulator.

This adds a new struct emul to the linker list of emulations. This is typically used in your emulator's *DT_INST_FOREACH_STATUS_OKAY()* clause.

Parameters

- `node_id` – Node ID of the driver to emulate (e.g. *DT_DRV_INST(n)*); the node_id *MUST* have a corresponding *DEVICE_DT_DEFINE()*.

- **init_fn** – function to call to initialise the emulator (see emul_init typedef)
- **data_ptr** – emulator-specific data
- **cfg_ptr** – emulator-specific configuration data
- **bus_api** – emulator-specific bus api
- **_backend_api** – emulator-specific backend api

EMUL_DT_INST_DEFINE(inst, ...)

Like *EMUL_DT_DEFINE()*, but uses an instance of a DT_DRV_COMPAT compatible instead of a node identifier.

Parameters

- **inst** – instance number. The node_id argument to EMUL_DT_DEFINE is set to *DT_DRV_INST(inst)*.
- **...** – other parameters as expected by EMUL_DT_DEFINE.

EMUL_DT_GET(node_id)

Get a const struct emul* from a devicetree node identifier.

Returns a pointer to an emulator object created from a devicetree node, if any device was allocated by an emulator implementation.

If no such device was allocated, this will fail at linker time. If you get an error that looks like undefined reference to __device_dts_ord_<N>, that is what happened. Check to make sure your emulator implementation is being compiled, usually by enabling the Kconfig options it requires.

Parameters

- **node_id** – A devicetree node identifier

Returns

A pointer to the emul object created for that node

EMUL_DT_GET_OR_NULL(node_id)

Utility macro to obtain an optional reference to an emulator.

If the node identifier refers to a node with status okay, this returns *EMUL_DT_GET(node_id)*. Otherwise, it returns NULL.

Parameters

- **node_id** – A devicetree node identifier

Returns

a *emul* reference for the node identifier, which may be NULL.

Typedefs

typedef int (***emul_init_t**)(const struct *emul* *emul, const struct *device* *parent)

Standard callback for emulator initialisation providing the initialiser record and the device that calls the emulator functions.

Param emul

Emulator to init

Param parent

Parent device that is using the emulator

Enums

enum **emul_bus_type**

The types of supported buses.

Values:

enumerator **EMUL_BUS_TYPE_I2C**

enumerator **EMUL_BUS_TYPE_ESPI**

enumerator **EMUL_BUS_TYPE_SPI**

enumerator **EMUL_BUS_TYPE_MSPI**

enumerator **EMUL_BUS_TYPE_NONE**

Functions

int **emul_init_for_bus**(const struct *device* *dev)

Set up a list of emulators.

Parameters

- **dev** – Device the emulators are attached to (e.g. an I2C controller)

Returns
0 if OK

Returns
negative value on error

const struct *emul* ***emul_get_binding**(const char *name)

Retrieve the emul structure for an emulator by name.

Emulator objects are created via the *EMUL_DT_DEFINE()* macro and placed in memory by the linker. If the emulator structure is needed for custom API calls, it can be retrieved by the name that the emulator exposes to the system (this is the devicetree node's label by default).

Parameters

- **name** – Emulator name to search for. A null pointer, or a pointer to an empty string, will cause NULL to be returned.

Returns
pointer to emulator structure; NULL if not found or cannot be used.

struct **emul_link_for_bus**

#include <emul.h> Structure uniquely identifying a device to be emulated.

struct **emul_list_for_bus**

#include <emul.h> List of emulators attached to a bus.

Public Members

const struct *emul_link_for_bus* ***children**

Identifiers for children of the node.

unsigned int **num_children**

Number of children of the node.

struct **no_bus_emul**

#include <*emul.h*> Emulator API stub when an emulator is not actually placed on a bus.

struct **emul**

#include <*emul.h*> An emulator instance - represents the *target* emulated device/peripheral that is interacted with through an emulated bus.

Instances of emulated bus nodes (e.g. i2c_emul) and emulators (i.e. struct emul) are exactly 1..1

Public Members

emul_init_t **init**

function used to initialise the emulator state

const struct *device* ***dev**

handle to the device for which this provides low-level emulation

const void ***cfg**

Emulator-specific configuration data.

void ***data**

Emulator-specific data.

enum *emul_bus_type* **bus_type**

The bus type that the emulator is attached to.

const void ***backend_api**

Address of the API structure exposed by the emulator instance.

union **bus**

#include <*emul.h*> Pointer to the emulated bus node.

Public Members

struct i2c_emul ***i2c**

struct espi_emul ***espi**

struct spi_emul ***spi**

struct mspi_emul ***mspi**

struct *no_bus_emul* ***none**

7.6 Peripherals

7.6.1 1-Wire Bus

Overview

1-Wire is a low speed half-duplex serial bus using only a single wire plus ground for both data transmission and device power supply. Similarly to I2C, 1-Wire uses a bidirectional open-collector data line, and is a single master multidrop bus. This means one master initiates all data exchanges with the slave devices. The 1-Wire bus supports longer bus lines than I2C, while it reaches speeds of up to 15.4 kbps in standard mode and up to 125 kbps in overdrive mode. Reliable communication in standard speed configuration is possible with 10 nodes over a bus length of 100 meters. Using overdrive speed, 3 nodes on a bus of 10 meters length are expected to work solid. Optimized timing parameters and fewer nodes on the bus may allow to reach larger bus extents.

The implementation details are specified in the BOOK OF IBUTTON STANDARDS.

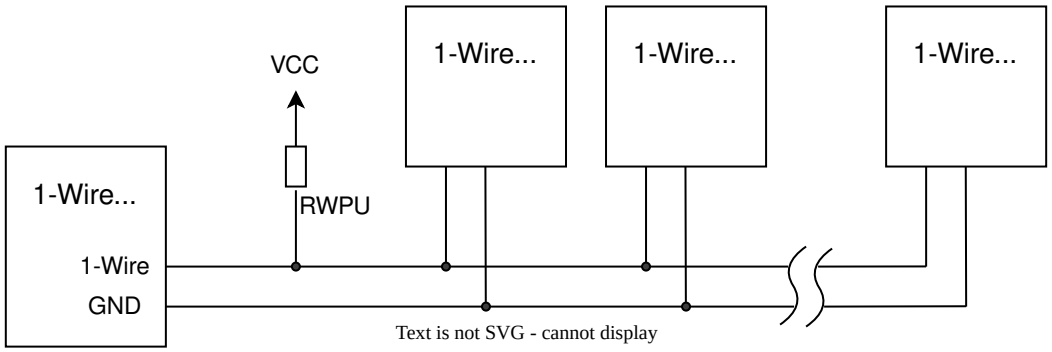


Fig. 1: A typical 1-Wire bus topology

W1 Master API Zephyr's 1-Wire Master API is used to interact with 1-Wire slave devices like temperature sensors and serial memories.

In Zephyr this API is split into the following layers.

- The link layer handles basic communication functions such as bus reset, presence detect and bit transfer operations. It is the only hardware-dependent layer in Zephyr. This layer is supported by a driver using the Zephyr *Universal Asynchronous Receiver-Transmitter (UART)* interface, which should work on most Zephyr platforms. In the future, a GPIO/Timer based driver and hardware specific drivers might be added.
- The 1-Wire network layer handles all means for slave identification and bus arbitration. This includes ROM commands like Match ROM, or Search ROM.
  - All slave devices have a unique 64-bit identification number, which includes a 8-bit 1-Wire Family Code and a 8-bit CRC.

  - In order to find slaves on the bus, the standard specifies an search algorithm which successively detects all slaves on the bus. This algorithm is described in detail by Maxim's Applicationnote 187.

- Transport layer and Presentation layer functions are not implemented in the generic 1-Wire driver and therefore must be handled in individual slave drivers.

The 1-Wire API is considered experimental.

Configuration Options

Related configuration options:

- `CONFIG_W1`
- `CONFIG_W1_NET`

API Reference

1-Wire data link layer

group `w1_data_link`

1-Wire data link layer

Functions

int `w1_reset_bus`(const struct *device* *dev)

Reset the 1-Wire bus to prepare slaves for communication.

This routine resets all 1-Wire bus slaves such that they are ready to receive a command. Connected slaves answer with a presence pulse once they are ready to receive data.

In case the driver supports both standard speed and overdrive speed, the reset routine takes care of sendig either a short or a long reset pulse depending on the current state. The speed can be changed using w1_configure().

Parameters

- `dev` – **[in]** Pointer to the device structure for the driver instance.

Return values

- `0` – If no slaves answer with a present pulse.
- `1` – If at least one slave answers with a present pulse.
- `-errno` – Negative error code on error.

int `w1_read_bit`(const struct *device* *dev)

Read a single bit from the 1-Wire bus.

Parameters

- `dev` – **[in]** Pointer to the device structure for the driver instance.

Return values

- `rx_bit` – The read bit value on success.
- `-errno` – Negative error code on error.

int `w1_write_bit`(const struct *device* *dev, const bool bit)

Write a single bit to the 1-Wire bus.

Parameters

- `dev` – **[in]** Pointer to the device structure for the driver instance.
- `bit` – Transmitting bit value 1 or 0.

Return values

- `0` – If successful.
- `-errno` – Negative error code on error.

int `w1_read_byte`(const struct *device* *dev)

Read a single byte from the 1-Wire bus.

Parameters

- `dev` – **[in]** Pointer to the device structure for the driver instance.

Return values

- `rx_byte` – The read byte value on success.
- `-errno` – Negative error code on error.

int `w1_write_byte`(const struct *device* *dev, uint8_t byte)

Write a single byte to the 1-Wire bus.

Parameters

- `dev` – **[in]** Pointer to the device structure for the driver instance.
- `byte` – Transmitting byte.

Return values

- `0` – If successful.
- `-errno` – Negative error code on error.

int `w1_read_block`(const struct *device* *dev, uint8_t *buffer, size_t len)

Read a block of data from the 1-Wire bus.

Parameters

- `dev` – **[in]** Pointer to the device structure for the driver instance.
- `buffer` – **[out]** Pointer to receive buffer.
- `len` – Length of receiving buffer (in bytes).

Return values

- `0` – If successful.
- `-errno` – Negative error code on error.

int `w1_write_block`(const struct *device* *dev, const uint8_t *buffer, size_t len)

Write a block of data from the 1-Wire bus.

Parameters

- `dev` – **[in]** Pointer to the device structure for the driver instance.
- `buffer` – **[in]** Pointer to transmitting buffer.
- `len` – Length of transmitting buffer (in bytes).

Return values

- `0` – If successful.

- `-errno` – Negative error code on error.

size_t `w1_get_slave_count`(const struct *device* *dev)

Get the number of slaves on the bus.

Parameters

- `dev` – **[in]** Pointer to the device structure for the driver instance.

Return values

- `slave_count` – Positive Number of connected 1-Wire slaves on success.
- `-errno` – Negative error code on error.

int `w1_configure`(const struct *device* *dev, enum *w1_settings_type* type, uint32_t value)

Configure parameters of the 1-Wire master.

Allowed configuration parameters are defined in enum w1_settings_type, but master devices may not support all types.

Parameters

- `dev` – **[in]** Pointer to the device structure for the driver instance.
- `type` – Enum specifying the setting type.
- `value` – The new value for the passed settings type.

Return values

- `0` – If successful.
- `-ENOTSUP` – The master doesn't support the configuration of the supplied type.
- `-EIO` – General input / output error, failed to configure master devices.

1-Wire network layer

group `w1_network`

1-Wire network layer

1-Wire ROM Commands

`W1_CMD_SKIP_ROM`

This command allows the bus master to read the slave devices without providing their ROM code.

`W1_CMD_MATCH_ROM`

This command allows the bus master to address a specific slave device by providing its ROM code.

`W1_CMD_RESUME`

This command allows the bus master to resume a previous read out from where it left off.

`W1_CMD_READ_ROM`

This command allows the bus master to read the ROM code from a single slave device.

This command should be used when there is only a single slave device on the bus.

W1_CMD_SEARCH_ROM

This command allows the bus master to discover the addresses (i.e., ROM codes) of all slave devices on the bus.

W1_CMD_SEARCH_ALARM

This command allows the bus master to identify which devices have experienced an alarm condition.

W1_CMD_OVERDRIVE_SKIP_ROM

This command allows the bus master to address all devices on the bus and then switch them to overdrive speed.

W1_CMD_OVERDRIVE_MATCH_ROM

This command allows the bus master to address a specific device and switch it to overdrive speed.

CRC Defines

W1_CRC8_SEED

Seed value used to calculate the 1-Wire 8-bit crc.

W1_CRC8_POLYNOMIAL

Polynomial used to calculate the 1-Wire 8-bit crc.

W1_CRC16_SEED

Seed value used to calculate the 1-Wire 16-bit crc.

W1_CRC16_POLYNOMIAL

Polynomial used to calculate the 1-Wire 16-bit crc.

Defines

W1_SEARCH_ALL_FAMILIES

This flag can be passed to searches in order to not filter on family ID.

W1_ROM_INIT_ZERO

Initialize all *w1_rom* struct members to zero.

Typedefs

typedef void (*w1_search_callback_t)(struct *w1_rom* rom, void *user_data)

Define the application callback handler function signature for searches.

Param rom
found The ROM of the found slave.

Param user_data
User data provided to the *w1_search_bus()* call.

Functions

int `w1_read_rom`(const struct *device* *dev, struct *w1_rom* *rom)

Read Peripheral 64-bit ROM.

This procedure allows the 1-Wire bus master to read the peripherals' 64-bit ROM without using the Search ROM procedure. This command can be used as long as not more than a single peripheral is connected to the bus. Otherwise data collisions occur and a faulty ROM is read.

Parameters

- `dev` – **[in]** Pointer to the device structure for the driver instance.
- `rom` – **[out]** Pointer to the ROM structure.

Return values

- `0` – If successful.
- `-ENODEV` – In case no slave responds to reset.
- `-errno` – Other negative error code in case of invalid crc and communication errors.

int `w1_match_rom`(const struct *device* *dev, const struct *w1_slave_config* *config)

Select a specific slave by broadcasting a selected ROM.

This routine allows the 1-Wire bus master to select a slave identified by its unique ROM, such that the next command will target only this single selected slave.

This command is only necessary in multidrop environments, otherwise the Skip ROM command can be issued. Once a slave has been selected, to reduce the communication overhead, the resume command can be used instead of this command to communicate with the selected slave.

Parameters

- `dev` – **[in]** Pointer to the device structure for the driver instance.
- `config` – **[in]** Pointer to the slave specific 1-Wire config.

Return values

- `0` – If successful.
- `-ENODEV` – In case no slave responds to reset.
- `-errno` – Other negative error code on error.

int `w1_resume_command`(const struct *device* *dev)

Select the slave last addressed with a Match ROM or Search ROM command.

This routine allows the 1-Wire bus master to re-select a slave device that was already addressed using a Match ROM or Search ROM command.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Return values

- `0` – If successful.
- `-ENODEV` – In case no slave responds to reset.
- `-errno` – Other negative error code on error.

int `w1_skip_rom`(const struct *device* *dev, const struct *w1_slave_config* *config)

Select all slaves regardless of ROM.

This routine sets up the bus slaves to receive a command. It is usually used when there is only one peripheral on the bus to avoid the overhead of the Match ROM command. But it can also be used to concurrently write to all slave devices.

Parameters

- `dev` – **[in]** Pointer to the device structure for the driver instance.
- `config` – **[in]** Pointer to the slave specific 1-Wire config.

Return values

- `0` – If successful.
- `-ENODEV` – In case no slave responds to reset.
- `-errno` – Other negative error code on error.

int `w1_reset_select`(const struct *device* *dev, const struct *w1_slave_config* *config)

In single drop configurations use Skip Select command, otherwise use Match ROM command.

Parameters

- `dev` – **[in]** Pointer to the device structure for the driver instance.
- `config` – **[in]** Pointer to the slave specific 1-Wire config.

Return values

- `0` – If successful.
- `-ENODEV` – In case no slave responds to reset.
- `-errno` – Other negative error code on error.

int `w1_write_read`(const struct *device* *dev, const struct *w1_slave_config* *config, const uint8_t *write_buf, size_t write_len, uint8_t *read_buf, size_t read_len)

Write then read data from the 1-Wire slave with matching ROM.

This routine uses w1_reset_select to select the given ROM. Then writes given data and reads the response back from the slave.

Parameters

- `dev` – **[in]** Pointer to the device structure for the driver instance.
- `config` – **[in]** Pointer to the slave specific 1-Wire config.
- `write_buf` – **[in]** Pointer to the data to be written.
- `write_len` – Number of bytes to write.
- `read_buf` – **[out]** Pointer to storage for read data.
- `read_len` – Number of bytes to read.

Return values

- `0` – If successful.
- `-ENODEV` – In case no slave responds to reset.
- `-errno` – Other negative error code on error.

int `w1_search_bus`(const struct *device* *dev, uint8_t command, uint8_t family, *w1_search_callback_t* callback, void *user_data)

Search 1-wire slaves on the bus.

This function searches slaves on the 1-wire bus, with the possibility to search either all slaves or only slaves that have an active alarm state. If a callback is passed, the callback is called for each found slave.

The algorithm mostly follows the suggestions of https://pdfserv.maximintegrated.com/en/an/AN187.pdf

Note: Filtering on families is not supported.

Parameters

- `dev` – **[in]** Pointer to the device structure for the driver instance.
- `command` – Can either be W1_SEARCH_ALARM or W1_SEARCH_ROM.
- `family` – W1_SEARCH_ALL_FAMILIES searcheas all families, filtering on a specific family is not yet supported.
- `callback` – Application callback handler function to be called for each found slave.
- `user_data` – **[in]** User data to pass to the application callback handler function.

Return values

- `slave_count` – Number of slaves found.
- `-errno` – Negative error code on error.

static inline int `w1_search_rom`(const struct *device* *dev, *w1_search_callback_t* callback, void *user_data)

Search for 1-Wire slave on bus.

This routine can discover unknown slaves on the bus by scanning for the unique 64-bit registration number.

Parameters

- `dev` – **[in]** Pointer to the device structure for the driver instance.
- `callback` – Application callback handler function to be called for each found slave.
- `user_data` – **[in]** User data to pass to the application callback handler function.

Return values

- `slave_count` – Number of slaves found.
- `-errno` – Negative error code on error.

static inline int `w1_search_alarm`(const struct *device* *dev, *w1_search_callback_t* callback, void *user_data)

Search for 1-Wire slaves with an active alarm.

This routine searches 1-Wire slaves on the bus, which currently have an active alarm.

Parameters

- `dev` – **[in]** Pointer to the device structure for the driver instance.
- `callback` – Application callback handler function to be called for each found slave.

- `user_data` – **[in]** User data to pass to the application callback handler function.

Return values

- `slave_count` – Number of slaves found.
- `-errno` – Negative error code on error.

static inline uint64_t `w1_rom_to_uint64`(const struct *w1_rom* *rom)

Function to convert a *w1_rom* struct to an uint64_t.

Parameters

- `rom` – **[in]** Pointer to the ROM struct.

Return values

`rom64` – The ROM converted to an unsigned integer in endianness.

static inline void `w1_uint64_to_rom`(const uint64_t rom64, struct *w1_rom* *rom)

Function to write an uint64_t to struct *w1_rom* pointer.

Parameters

- `rom64` – Unsigned 64 bit integer representing the ROM in host endianness.
- `rom` – **[out]** The ROM struct pointer.

static inline uint8_t `w1_crc8`(const uint8_t *src, size_t len)

Compute CRC-8 chacksum as defined in the 1-Wire specification.

The 1-Wire of CRC 8 variant is using 0x31 as its polynomial with the initial value set to 0x00. This CRC is used to check the correctness of the unique 56-bit ROM.

Parameters

- `src` – **[in]** Input bytes for the computation.
- `len` – Length of the input in bytes.

Return values

`crc` – The computed CRC8 value.

static inline uint16_t `w1_crc16`(const uint16_t seed, const uint8_t *src, const size_t len)

Compute 1-Wire variant of CRC 16.

The 16-bit 1-Wire crc variant is using the reflected polynomial function X^16 + X^15 * + X^2 + 1 with the initial value set to 0x0000. See also APPLICATION NOTE 27: "UNDERSTANDING AND USING CYCLIC REDUNDANCY CHECKS WITH MAXIM 1-WIRE AND IBUTTON PRODUCTS" https://www.maximintegrated.com/en/design/technical-documents/app-notes/2/27.html

Parameters

- `seed` – Init value for the CRC, it is usually set to 0x0000.
- `src` – **[in]** Input bytes for the computation.
- `len` – Length of the input in bytes.

Return values

`crc` – The computed CRC16 value.

struct `w1_rom`

#include <*w1.h*> *w1_rom* struct.

Public Members

uint8_t **family**

The 1-Wire family code identifying the slave device type.

An incomplete list of family codes is available at: https://www.maximintegrated.com/en/app-notes/index.mvp/id/155 others are documented in the respective device data sheet.

uint8_t **serial**[6]

The serial together with the family code composes the unique 56-bit id.

uint8_t **crc**

8-bit checksum of the 56-bit unique id.

struct **w1_slave_config**

#include <*w1.h*> Node specific 1-wire configuration struct.

This struct is passed to network functions, such that they can configure the bus to address the specific slave using the selected speed.

Public Members

struct *w1_rom* **rom**

Unique 1-Wire ROM.

uint32_t **overdrive**

overdrive speed is used if set to 1.

1-Wire generic functions and helpers Functions that are not directly related to any of the networking layers.

Related code samples

1-Wire scanner
Scan for 1-Wire devices and print their family ID and serial number.

group **w1_interface**

1-Wire Interface

Since
3.2

Version
0.1.0

Enums

enum `w1_settings_type`

Defines the 1-Wire master settings types, which are runtime configurable.

Values:

enumerator `W1_SETTING_SPEED`

Overdrive speed is enabled in case a value of 1 is passed and disabled passing 0.

enumerator `W1_SETTING_STRONG_PULLUP`

The strong pullup resistor is activated immediately after the next written data block by passing a value of 1, and deactivated passing 0.

enumerator `W1_SETINGS_TYPE_COUNT`

Number of different settings types.

Functions

static inline int `w1_lock_bus`(const struct *device* *dev)

Lock the 1-wire bus to prevent simultaneous access.

This routine locks the bus to prevent simultaneous access from different threads. The calling thread waits until the bus becomes available. A thread is permitted to lock a mutex it has already locked.

Parameters

- `dev` – **[in]** Pointer to the device structure for the driver instance.

Return values

- `0` – If successful.
- `-errno` – Negative error code on error.

static inline int `w1_unlock_bus`(const struct *device* *dev)

Unlock the 1-wire bus.

This routine unlocks the bus to permit access to bus line.

Parameters

- `dev` – **[in]** Pointer to the device structure for the driver instance.

Return values

- `0` – If successful.
- `-errno` – Negative error code on error.

7.6.2 Analog-to-Digital Converter (ADC)

Overview

API Reference

Related code samples

Analog-to-Digital Converter (ADC) sequence sample
Read analog inputs from ADC channels, using a sequence.

Analog-to-Digital Converter (ADC) with devicetree
Read analog inputs from ADC channels.

group `adc_interface`

ADC driver APIs.

Since
1.0

Version
1.0.0

Defines

`ADC_CHANNEL_CFG_DT`(node_id)

Get ADC channel configuration from a given devicetree node.

This returns a static initializer for a `struct` *adc_channel_cfg* filled with data from a given devicetree node.

Example devicetree fragment:

```
&adc {
   #address-cells = <1>;
   #size-cells = <0>;

   channel@0 {
       reg = <0>;
       zephyr,gain = "ADC_GAIN_1_6";
       zephyr,reference = "ADC_REF_INTERNAL";
       zephyr,acquisition-time = <ADC_ACQ_TIME(ADC_ACQ_TIME_MICROSECONDS, 20)>;
       zephyr,input-positive = <NRF_SAADC_AIN6>;
       zephyr,input-negative = <NRF_SAADC_AIN7>;
   };

   channel@1 {
       reg = <1>;
       zephyr,gain = "ADC_GAIN_1_6";
       zephyr,reference = "ADC_REF_INTERNAL";
       zephyr,acquisition-time = <ADC_ACQ_TIME_DEFAULT>;
       zephyr,input-positive = <NRF_SAADC_AIN0>;
   };
};
```

Example usage:

```
static const struct adc_channel_cfg ch0_cfg_dt =
    ADC_CHANNEL_CFG_DT(DT_CHILD(DT_NODELABEL(adc), channel_0));
static const struct adc_channel_cfg ch1_cfg_dt =
    ADC_CHANNEL_CFG_DT(DT_CHILD(DT_NODELABEL(adc), channel_1));

// Initializes 'ch0_cfg_dt' to:
```

(continues on next page)

(continued from previous page)

```
// {
//     .channel_id = 0,
//     .gain = ADC_GAIN_1_6,
//     .reference = ADC_REF_INTERNAL,
//     .acquisition_time = ADC_ACQ_TIME(ADC_ACQ_TIME_MICROSECONDS, 20),
//     .differential = true,
//     .input_positive = NRF_SAADC_AIN6,
//     .input-negative = NRF_SAADC_AIN7,
// }
// and 'ch1_cfg_dt' to:
// {
//     .channel_id = 1,
//     .gain = ADC_GAIN_1_6,
//     .reference = ADC_REF_INTERNAL,
//     .acquisition_time = ADC_ACQ_TIME_DEFAULT,
//     .input_positive = NRF_SAADC_AIN0,
// }
```

Parameters

- **node_id** – Devicetree node identifier.

Returns

Static initializer for an *adc_channel_cfg* structure.

ADC_DT_SPEC_GET_BY_NAME(node_id, name)

Get ADC io-channel information from devicetree by name.

This returns a static initializer for an *adc_dt_spec* structure given a devicetree node and a channel name. The node must have the “io-channels” property defined.

Example devicetree fragment:

```
/ {
    zephyr,user {
        io-channels = <&adc0 1>, <&adc0 3>;
        io-channel-names = "A0", "A1";
    };
};

&adc0 {
   #address-cells = <1>;
   #size-cells = <0>;

   channel@3 {
       reg = <3>;
       zephyr,gain = "ADC_GAIN_1_5";
       zephyr,reference = "ADC_REF_VDD_1_4";
       zephyr,vref-mv = <750>;
       zephyr,acquisition-time = <ADC_ACQ_TIME_DEFAULT>;
       zephyr,resolution = <12>;
       zephyr,oversampling = <4>;
   };
};
```

Example usage:

```
static const struct adc_dt_spec adc_chan0 =
    ADC_DT_SPEC_GET_BY_NAME(DT_PATH(zephyr_user), a0);
static const struct adc_dt_spec adc_chan1 =
    ADC_DT_SPEC_GET_BY_NAME(DT_PATH(zephyr_user), a1);
```

(continues on next page)

(continued from previous page)

```
// Initializes 'adc_chan0' to:
// {
//     .dev = DEVICE_DT_GET(DT_NODELABEL(adc0)),
//     .channel_id = 1,
// }
// and 'adc_chan1' to:
// {
//     .dev = DEVICE_DT_GET(DT_NODELABEL(adc0)),
//     .channel_id = 3,
//     .channel_cfg_dt_node_exists = true,
//     .channel_cfg = {
//         .channel_id = 3,
//         .gain = ADC_GAIN_1_5,
//         .reference = ADC_REF_VDD_1_4,
//         .acquisition_time = ADC_ACQ_TIME_DEFAULT,
//     },
//     .vref_mv = 750,
//     .resolution = 12,
//     .oversampling = 4,
// }
```

Parameters

- `node_id` – Devicetree node identifier.
- `name` – Channel name.

Returns
Static initializer for an *adc_dt_spec* structure.

`ADC_DT_SPEC_INST_GET_BY_NAME`(inst, name)
Get ADC io-channel information from a DT_DRV_COMPAT devicetree instance by name.

See also

ADC_DT_SPEC_GET_BY_NAME()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `name` – Channel name.

Returns
Static initializer for an *adc_dt_spec* structure.

`ADC_DT_SPEC_GET_BY_IDX`(node_id, idx)
Get ADC io-channel information from devicetree.

This returns a static initializer for an *adc_dt_spec* structure given a devicetree node and a channel index. The node must have the “io-channels” property defined.

Example devicetree fragment:

```
/ {
    zephyr,user {
        io-channels = <&adc0 1>, <&adc0 3>;
    };
```

(continues on next page)

(continued from previous page)

```
};

&adc0 {
   #address-cells = <1>;
   #size-cells = <0>;

   channel@3 {
       reg = <3>;
       zephyr,gain = "ADC_GAIN_1_5";
       zephyr,reference = "ADC_REF_VDD_1_4";
       zephyr,vref-mv = <750>;
       zephyr,acquisition-time = <ADC_ACQ_TIME_DEFAULT>;
       zephyr,resolution = <12>;
       zephyr,oversampling = <4>;
   };
};
```

Example usage:

```
static const struct adc_dt_spec adc_chan0 =
    ADC_DT_SPEC_GET_BY_IDX(DT_PATH(zephyr_user), 0);
static const struct adc_dt_spec adc_chan1 =
    ADC_DT_SPEC_GET_BY_IDX(DT_PATH(zephyr_user), 1);

// Initializes 'adc_chan0' to:
// {
//     .dev = DEVICE_DT_GET(DT_NODELABEL(adc0)),
//     .channel_id = 1,
// }
// and 'adc_chan1' to:
// {
//     .dev = DEVICE_DT_GET(DT_NODELABEL(adc0)),
//     .channel_id = 3,
//     .channel_cfg_dt_node_exists = true,
//     .channel_cfg = {
//         .channel_id = 3,
//         .gain = ADC_GAIN_1_5,
//         .reference = ADC_REF_VDD_1_4,
//         .acquisition_time = ADC_ACQ_TIME_DEFAULT,
//     },
//     .vref_mv = 750,
//     .resolution = 12,
//     .oversampling = 4,
// }
```

See also

ADC_DT_SPEC_GET()

Parameters

- `node_id` – Devicetree node identifier.
- `idx` – Channel index.

Returns

Static initializer for an *adc_dt_spec* structure.

ADC_DT_SPEC_INST_GET_BY_IDX(inst, idx)

Get ADC io-channel information from a DT_DRV_COMPAT devicetree instance.

See also

ADC_DT_SPEC_GET_BY_IDX()

Parameters

- **inst** – DT_DRV_COMPAT instance number
- **idx** – Channel index.

Returns

Static initializer for an *adc_dt_spec* structure.

ADC_DT_SPEC_GET(node_id)

Equivalent to *ADC_DT_SPEC_GET_BY_IDX(node_id, 0)*.

See also

ADC_DT_SPEC_GET_BY_IDX()

Parameters

- **node_id** – Devicetree node identifier.

Returns

Static initializer for an *adc_dt_spec* structure.

ADC_DT_SPEC_INST_GET(inst)

Equivalent to *ADC_DT_SPEC_INST_GET_BY_IDX(inst, 0)*.

See also

ADC_DT_SPEC_GET()

Parameters

- **inst** – DT_DRV_COMPAT instance number

Returns

Static initializer for an *adc_dt_spec* structure.

Typedefs

typedef enum *adc_action* (***adc_sequence_callback**)(const struct *device* *dev, const struct *adc_sequence* *sequence, uint16_t sampling_index)

Type definition of the optional callback function to be called after a requested sampling is done.

Param dev
Pointer to the device structure for the driver instance.

Param sequence
Pointer to the sequence structure that triggered the sampling. This parameter points to a copy of the structure that was supplied to the call that started the sampling sequence, thus it cannot be used with the *CONTAINER_OF()* macro to retrieve some other data associated with the sequence. Instead, the *adc_sequence_options::user_data* field should be used for such purpose.

Param sampling_index
Index (0-65535) of the sampling done.

Return
Action to be performed by the driver. See *adc_action*.

typedef int (*`adc_api_channel_setup`)(const struct *device* *dev, const struct *adc_channel_cfg* *channel_cfg)

Type definition of ADC API function for configuring a channel.

See *adc_channel_setup()* for argument descriptions.

typedef int (*`adc_api_read`)(const struct *device* *dev, const struct *adc_sequence* *sequence)

Type definition of ADC API function for setting a read request.

See *adc_read()* for argument descriptions.

typedef int (*`adc_api_read_async`)(const struct *device* *dev, const struct *adc_sequence* *sequence, struct *k_poll_signal* *async)

Type definition of ADC API function for setting an asynchronous read request.

See *adc_read_async()* for argument descriptions.

Enums

enum `adc_gain`

ADC channel gain factors.

Values:

enumerator `ADC_GAIN_1_6`

x 1/6.

enumerator `ADC_GAIN_1_5`

x 1/5.

enumerator `ADC_GAIN_1_4`

x 1/4.

enumerator `ADC_GAIN_1_3`

x 1/3.

enumerator `ADC_GAIN_2_5`

x 2/5.

enumerator ADC_GAIN_1_2
x 1/2.

enumerator ADC_GAIN_2_3
x 2/3.

enumerator ADC_GAIN_4_5
x 4/5.

enumerator ADC_GAIN_1
x 1.

enumerator ADC_GAIN_2
x 2.

enumerator ADC_GAIN_3
x 3.

enumerator ADC_GAIN_4
x 4.

enumerator ADC_GAIN_6
x 6.

enumerator ADC_GAIN_8
x 8.

enumerator ADC_GAIN_12
x 12.

enumerator ADC_GAIN_16
x 16.

enumerator ADC_GAIN_24
x 24.

enumerator ADC_GAIN_32
x 32.

enumerator ADC_GAIN_64
x 64.

enumerator ADC_GAIN_128
x 128.

enum adc_reference
ADC references.
Values:

enumerator ADC_REF_VDD_1
VDD.

enumerator ADC_REF_VDD_1_2
VDD/2.

enumerator ADC_REF_VDD_1_3
VDD/3.

enumerator ADC_REF_VDD_1_4
VDD/4.

enumerator ADC_REF_INTERNAL
Internal.

enumerator ADC_REF_EXTERNAL0
External, input 0.

enumerator ADC_REF_EXTERNAL1
External, input 1.

enum adc_action
Action to be performed after a sampling is done.

Values:

enumerator ADC_ACTION_CONTINUE = 0
The sequence should be continued normally.

enumerator ADC_ACTION_REPEAT
The sampling should be repeated.

New samples or sample should be read from the ADC and written in the same place as the recent ones.

enumerator ADC_ACTION_FINISH
The sequence should be finished immediately.

Functions

int adc_gain_invert(enum *adc_gain* gain, int32_t *value)
Invert the application of gain to a measurement value.

For example, if the gain passed in is ADC_GAIN_1_6 and the referenced value is 10, the value after the function returns is 60.

Parameters

- gain – the gain used to amplify the input signal.
- value – a pointer to a value that initially has the effect of the applied gain but has that effect removed when this function successfully returns. If the gain cannot be reversed the value remains unchanged.

Return values

- `0` – if the gain was successfully reversed
- `-EINVAL` – if the gain could not be interpreted

int `adc_channel_setup`(const struct *device* *dev, const struct *adc_channel_cfg* *channel_cfg)

Configure an ADC channel.

It is required to call this function and configure each channel before it is selected for a read request.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `channel_cfg` – Channel configuration.

Return values

- `0` – On success.
- `-EINVAL` – If a parameter with an invalid value has been provided.

static inline int `adc_channel_setup_dt`(const struct *adc_dt_spec* *spec)

Configure an ADC channel from a struct *adc_dt_spec*.

See also

adc_channel_setup()

Parameters

- `spec` – ADC specification from Devicetree.

Returns

A value from *adc_channel_setup()* or -ENOTSUP if information from Devicetree is not valid.

int `adc_read`(const struct *device* *dev, const struct *adc_sequence* *sequence)

Set a read request.

If invoked from user mode, any sequence struct options for callback must be NULL.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `sequence` – Structure specifying requested sequence of samplings.

Return values

- `0` – On success.
- `-EINVAL` – If a parameter with an invalid value has been provided.
- `-ENOMEM` – If the provided buffer is to small to hold the results of all requested samplings.
- `-ENOTSUP` – If the requested mode of operation is not supported.

- **-EBUSY** – If another sampling was triggered while the previous one was still in progress. This may occur only when samplings are done with intervals, and it indicates that the selected interval was too small. All requested samples are written in the buffer, but at least some of them were taken with an extra delay compared to what was scheduled.

static inline int **adc_read_dt**(const struct *adc_dt_spec* *spec, const struct *adc_sequence* *sequence)

Set a read request from a struct *adc_dt_spec*.

See also

adc_read()

Parameters

- **spec** – ADC specification from Devicetree.
- **sequence** – Structure specifying requested sequence of samplings.

Returns

A value from *adc_read()*.

int **adc_read_async**(const struct *device* *dev, const struct *adc_sequence* *sequence, struct *k_poll_signal* *async)

Set an asynchronous read request.

If invoked from user mode, any sequence struct options for callback must be NULL.

Note

This function is available only if `CONFIG_ADC_ASYNC` is selected.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **sequence** – Structure specifying requested sequence of samplings.
- **async** – Pointer to a valid and ready to be signaled struct *k_poll_signal*. (Note: if NULL this function will not notify the end of the transaction, and whether it went successfully or not).

Returns

0 on success, negative error code otherwise. See *adc_read()* for a list of possible error codes.

static inline uint16_t **adc_ref_internal**(const struct *device* *dev)

Get the internal reference voltage.

Returns the voltage corresponding to *ADC_REF_INTERNAL*, measured in millivolts.

Returns

a positive value is the reference voltage value. Returns zero if reference voltage information is not available.

static inline int **adc_raw_to_millivolts**(int32_t ref_mv, enum *adc_gain* gain, uint8_t resolution, int32_t *valp)

Convert a raw ADC value to millivolts.

This function performs the necessary conversion to transform a raw ADC measurement to a voltage in millivolts.

Parameters

- `ref_mv` – the reference voltage used for the measurement, in millivolts. This may be from *adc_ref_internal()* or a known external reference.
- `gain` – the ADC gain configuration used to sample the input
- `resolution` – the number of bits in the absolute value of the sample. For differential sampling this needs to be one less than the resolution in struct *adc_sequence*.
- `valp` – pointer to the raw measurement value on input, and the corresponding millivolt value on successful conversion. If conversion fails the stored value is left unchanged.

Return values

- `0` – on successful conversion
- `-EINVAL` – if the gain is not reversible

static inline int **adc_raw_to_millivolts_dt**(const struct *adc_dt_spec* *spec, int32_t *valp)

Convert a raw ADC value to millivolts using information stored in a struct *adc_dt_spec*.

See also

adc_raw_to_millivolts()

Parameters

- `spec` – **[in]** ADC specification from Devicetree.
- `valp` – **[inout]** Pointer to the raw measurement value on input, and the corresponding millivolt value on successful conversion. If conversion fails the stored value is left unchanged.

Returns

A value from *adc_raw_to_millivolts()* or -ENOTSUP if information from Devicetree is not valid.

static inline int **adc_sequence_init_dt**(const struct *adc_dt_spec* *spec, struct *adc_sequence* *seq)

Initialize a struct *adc_sequence* from information stored in struct *adc_dt_spec*.

Note that this function only initializes the following fields:

- *adc_sequence::channels*
- *adc_sequence::resolution*
- *adc_sequence::oversampling*

Other fields should be initialized by the caller.

Parameters

- **spec** – **[in]** ADC specification from Devicetree.
- **seq** – **[out]** Sequence to initialize.

Return values

- **0** – On success
- **-ENOTSUP** – If spec does not have valid channel configuration

static inline bool **adc_is_ready_dt**(const struct *adc_dt_spec* *spec)

Validate that the ADC device is ready.

Parameters

- **spec** – ADC specification from devicetree

Return values

true – if the ADC device is ready for use and false otherwise.

struct **adc_channel_cfg**

#include <*adc.h*> Structure for specifying the configuration of an ADC channel.

Public Members

enum *adc_gain* **gain**

Gain selection.

enum *adc_reference* **reference**

Reference selection.

uint16_t **acquisition_time**

Acquisition time.

Use the ADC_ACQ_TIME macro to compose the value for this field or pass ADC_ACQ_TIME_DEFAULT to use the default setting for a given hardware (e.g. when the hardware does not allow to configure the acquisition time). Particular drivers do not necessarily support all the possible units. Value range is 0-16383 for a given unit.

uint8_t **channel_id**

Channel identifier.

This value primarily identifies the channel within the ADC API - when a read request is done, the corresponding bit in the “channels” field of the “adc_sequence” structure must be set to include this channel in the sampling. For hardware that does not allow selection of analog inputs for given channels, but rather have dedicated ones, this value also selects the physical ADC input to be used in the sampling. Otherwise, when it is needed to explicitly select an analog input for the channel, or two inputs when the channel is a differential one, the selection is done in “input_positive” and “input_negative” fields. Particular drivers indicate which one of the above two cases they support by selecting or not a special hidden Kconfig option named ADC_CONFIGURABLE_INPUTS. If this option is not selected, the macro CONFIG_ADC_CONFIGURABLE_INPUTS is not defined and consequently the mentioned two fields are not present in this structure. While this API allows identifiers from range 0-31, particular drivers may support only a limited number of channel identifiers (dependent on the underlying hardware capabilities or configured via a dedicated Kconfig option).

uint8_t **differential**

Channel type: single-ended or differential.

struct **adc_dt_spec**

#include <*adc.h*> Container for ADC channel information specified in devicetree.

See also

ADC_DT_SPEC_GET_BY_IDX

See also

ADC_DT_SPEC_GET

Public Members

const struct *device* ***dev**

Pointer to the device structure for the ADC driver instance used by this io-channel.

uint8_t **channel_id**

ADC channel identifier used by this io-channel.

bool **channel_cfg_dt_node_exists**

Flag indicating whether configuration of the associated ADC channel is provided as a child node of the corresponding ADC controller in devicetree.

struct *adc_channel_cfg* **channel_cfg**

Configuration of the associated ADC channel specified in devicetree.

This field is valid only when *channel_cfg_dt_node_exists* is set to *true*.

uint16_t **vref_mv**

Voltage of the reference selected for the channel or 0 if this value is not provided in devicetree.

This field is valid only when *channel_cfg_dt_node_exists* is set to *true*.

uint8_t **resolution**

ADC resolution to be used for that channel.

This field is valid only when *channel_cfg_dt_node_exists* is set to *true*.

uint8_t **oversampling**

Oversampling setting to be used for that channel.

This field is valid only when *channel_cfg_dt_node_exists* is set to *true*.

struct **adc_sequence_options**

#include <*adc.h*> Structure defining additional options for an ADC sampling sequence.

Public Members

uint32_t **interval_us**

Interval between consecutive samplings (in microseconds), 0 means sample as fast as possible, without involving any timer.

The accuracy of this interval is dependent on the implementation of a given driver. The default routine that handles the intervals uses a kernel timer for this purpose, thus, it has the accuracy of the kernel's system clock. Particular drivers may use some dedicated hardware timers and achieve a better precision.

adc_sequence_callback **callback**

Callback function to be called after each sampling is done.

Optional - set to NULL if it is not needed.

void ***user_data**

Pointer to user data.

It can be used to associate the sequence with any other data that is needed in the callback function.

uint16_t **extra_samplings**

Number of extra samplings to perform (the total number of samplings is 1 + extra_samplings).

struct **adc_sequence**

#include <*adc.h*> Structure defining an ADC sampling sequence.

Public Members

const struct *adc_sequence_options* ***options**

Pointer to a structure defining additional options for the sequence.

If NULL, the sequence consists of a single sampling.

uint32_t **channels**

Bit-mask indicating the channels to be included in each sampling of this sequence.

All selected channels must be configured with *adc_channel_setup()* before they are used in a sequence. The least significant bit corresponds to channel 0.

void ***buffer**

Pointer to a buffer where the samples are to be written.

Samples from subsequent samplings are written sequentially in the buffer. The number of samples written for each sampling is determined by the number of channels selected in the "channels" field. The values written to the buffer represent a sample from each selected channel starting from the one with the lowest ID. The buffer must be of an appropriate size, taking into account the number of selected channels and the ADC resolution used, as well as the number of samplings contained in the sequence.

size_t **buffer_size**

Specifies the actual size of the buffer pointed by the “buffer” field (in bytes).

The driver must ensure that samples are not written beyond the limit and it must return an error if the buffer turns out to be not large enough to hold all the requested samples.

uint8_t **resolution**

ADC resolution.

For single-ended channels the sample values are from range: 0 .. 2^resolution - 1, for differential ones:

- 2^(resolution-1) .. 2^(resolution-1) - 1.

uint8_t **oversampling**

Oversampling setting.

Each sample is averaged from 2^oversampling conversion results. This feature may be unsupported by a given ADC hardware, or in a specific mode (e.g. when sampling multiple channels).

bool **calibrate**

Perform calibration before the reading is taken if requested.

The impact of channel configuration on the calibration process is specific to the underlying hardware. ADC implementations that do not support calibration should ignore this flag.

struct **adc_driver_api**

#include <*adc.h*> ADC driver API.

This is the mandatory API any ADC driver needs to expose.

7.6.3 Auxiliary Display (auxdisplay)

Overview

Auxiliary Displays are text-based displays that have simple interfaces for displaying textual, numeric or alphanumeric data, as opposed to the *Display Interface*, auxiliary displays do not support custom graphical output to displays (and most often monochrome), the most advanced custom feature supported is generation of custom characters. These inexpensive displays are commonly found with various configurations and sizes, a common display size is 16 characters by 2 lines.

This API is unstable and subject to change.

Configuration Options

Related configuration options:

- CONFIG_AUXDISPLAY
- CONFIG_AUXDISPLAY_INIT_PRIORITY

API Reference

Related code samples

Auxiliary display
Output ”Hello World” to an auxiliary display.

group `auxdisplay_interface`

Auxiliary (Text) Display Interface.

Since
3.4

Version
0.1.0

Defines

`AUXDISPLAY_LIGHT_NOT_SUPPORTED`

Used for minimum and maximum brightness/backlight values if not supported.

Typedefs

typedef uint32_t `auxdisplay_mode_t`

Used to describe the mode of an auxiliary (text) display.

Enums

enum `auxdisplay_position`

Used for moving the cursor or display position.

Values:

enumerator `AUXDISPLAY_POSITION_ABSOLUTE` = 0

Moves to specified X,Y position.

enumerator `AUXDISPLAY_POSITION_RELATIVE`

Shifts current position by +/- X,Y position, does not take display direction into consideration.

enumerator `AUXDISPLAY_POSITION_RELATIVE_DIRECTION`

Shifts current position by +/- X,Y position, takes display direction into consideration.

enumerator `AUXDISPLAY_POSITION_COUNT`

enum `auxdisplay_direction`

Used for setting character append position.

Values:

enumerator `AUXDISPLAY_DIRECTION_RIGHT` = 0

Each character will be placed to the right of existing characters.

enumerator `AUXDISPLAY_DIRECTION_LEFT`

Each character will be placed to the left of existing characters.

enumerator `AUXDISPLAY_DIRECTION_COUNT`

Functions

int `auxdisplay_display_on`(const struct *device* *dev)

Turn display on.

Parameters

- `dev` – Auxiliary display device instance

Return values

- `0` – on success.
- `-ENOSYS` – if not supported/implemented.
- `-errno` – Negative errno code on other failure.

int `auxdisplay_display_off`(const struct *device* *dev)

Turn display off.

Parameters

- `dev` – Auxiliary display device instance

Return values

- `0` – on success.
- `-ENOSYS` – if not supported/implemented.
- `-errno` – Negative errno code on other failure.

int `auxdisplay_cursor_set_enabled`(const struct *device* *dev, bool enabled)

Set cursor enabled status on an auxiliary display.

Parameters

- `dev` – Auxiliary display device instance
- `enabled` – True to enable cursor, false to disable

Return values

- `0` – on success.
- `-ENOSYS` – if not supported/implemented.
- `-errno` – Negative errno code on other failure.

int `auxdisplay_position_blinking_set_enabled`(const struct *device* *dev, bool enabled)

Set cursor blinking status on an auxiliary display.

Parameters

- `dev` – Auxiliary display device instance
- `enabled` – Set to true to enable blinking position, false to disable

Return values

- `0` – on success.
- `-ENOSYS` – if not supported/implemented.
- `-errno` – Negative errno code on other failure.

int `auxdisplay_cursor_shift_set`(const struct *device* *dev, uint8_t direction, bool display_shift)

Set cursor shift after character write and display shift.

Parameters

- `dev` – Auxiliary display device instance
- `direction` – Sets the direction of the display when characters are written
- `display_shift` – If true, will shift the display when characters are written (which makes it look like the display is moving, not the cursor)

Return values

- `0` – on success.
- `-ENOSYS` – if not supported/implemented.
- `-EINVAL` – if provided argument is invalid.
- `-errno` – Negative errno code on other failure.

int `auxdisplay_cursor_position_set`(const struct *device* *dev, enum *auxdisplay_position* type, int16_t x, int16_t y)

Set cursor (and write position) on an auxiliary display.

Parameters

- `dev` – Auxiliary display device instance
- `type` – Type of move, absolute or offset
- `x` – Exact or offset X position
- `y` – Exact or offset Y position

Return values

- `0` – on success.
- `-ENOSYS` – if not supported/implemented.
- `-EINVAL` – if provided argument is invalid.
- `-errno` – Negative errno code on other failure.

int `auxdisplay_cursor_position_get`(const struct *device* *dev, int16_t *x, int16_t *y)

Get current cursor on an auxiliary display.

Parameters

- `dev` – Auxiliary display device instance
- `x` – Will be updated with the exact X position

- `y` – Will be updated with the exact Y position

Return values

- `0` – on success.
- `-ENOSYS` – if not supported/implemented.
- `-EINVAL` – if provided argument is invalid.
- `-errno` – Negative errno code on other failure.

int `auxdisplay_display_position_set`(const struct *device* *dev, enum *auxdisplay_position* type, int16_t x, int16_t y)

Set display position on an auxiliary display.

Parameters

- `dev` – Auxiliary display device instance
- `type` – Type of move, absolute or offset
- `x` – Exact or offset X position
- `y` – Exact or offset Y position

Return values

- `0` – on success.
- `-ENOSYS` – if not supported/implemented.
- `-EINVAL` – if provided argument is invalid.
- `-errno` – Negative errno code on other failure.

int `auxdisplay_display_position_get`(const struct *device* *dev, int16_t *x, int16_t *y)

Get current display position on an auxiliary display.

Parameters

- `dev` – Auxiliary display device instance
- `x` – Will be updated with the exact X position
- `y` – Will be updated with the exact Y position

Return values

- `0` – on success.
- `-ENOSYS` – if not supported/implemented.
- `-EINVAL` – if provided argument is invalid.
- `-errno` – Negative errno code on other failure.

int `auxdisplay_capabilities_get`(const struct *device* *dev, struct *auxdisplay_capabilities* *capabilities)

Fetch capabilities (and details) of auxiliary display.

Parameters

- `dev` – Auxiliary display device instance
- `capabilities` – Will be updated with the details of the auxiliary display

Return values

- `0` – on success.
- `-errno` – Negative errno code on other failure.

int `auxdisplay_clear`(const struct *device* *dev)

Clear display of auxiliary display and return to home position (note that this does not reset the display configuration, e.g.

custom characters and display mode will persist).

Parameters

- `dev` – Auxiliary display device instance

Return values

- `0` – on success.
- `-errno` – Negative errno code on other failure.

int `auxdisplay_brightness_get`(const struct *device* *dev, uint8_t *brightness)

Get the current brightness level of an auxiliary display.

Parameters

- `dev` – Auxiliary display device instance
- `brightness` – Will be updated with the current brightness

Return values

- `0` – on success.
- `-ENOSYS` – if not supported/implemented.
- `-errno` – Negative errno code on other failure.

int `auxdisplay_brightness_set`(const struct *device* *dev, uint8_t brightness)

Update the brightness level of an auxiliary display.

Parameters

- `dev` – Auxiliary display device instance
- `brightness` – The brightness level to set

Return values

- `0` – on success.
- `-ENOSYS` – if not supported/implemented.
- `-EINVAL` – if provided argument is invalid.
- `-errno` – Negative errno code on other failure.

int `auxdisplay_backlight_get`(const struct *device* *dev, uint8_t *backlight)

Get the backlight level details of an auxiliary display.

Parameters

- `dev` – Auxiliary display device instance
- `backlight` – Will be updated with the current backlight level

Return values

- `0` – on success.
- `-ENOSYS` – if not supported/implemented.
- `-errno` – Negative errno code on other failure.

int `auxdisplay_backlight_set`(const struct *device* *dev, uint8_t backlight)

Update the backlight level of an auxiliary display.

Parameters

- `dev` – Auxiliary display device instance
- `backlight` – The backlight level to set

Return values

- `0` – on success.
- `-ENOSYS` – if not supported/implemented.
- `-EINVAL` – if provided argument is invalid.
- `-errno` – Negative errno code on other failure.

int `auxdisplay_is_busy`(const struct *device* *dev)

Check if an auxiliary display driver is busy.

Parameters

- `dev` – Auxiliary display device instance

Return values

- `1` – on success and display busy.
- `0` – on success and display not busy.
- `-ENOSYS` – if not supported/implemented.
- `-errno` – Negative errno code on other failure.

int `auxdisplay_custom_character_set`(const struct *device* *dev, struct *auxdisplay_character* *character)

Sets a custom character in the display, the custom character struct must contain the pixel data for the custom character to add and valid custom character index, if successful then the character_code variable in the struct will be set to the character code that can be used with the *auxdisplay_write()* function to show it.

A character must be valid for a display consisting of a uint8 array of size character width by character height, values should be 0x00 for pixel off or 0xff for pixel on, if a display supports shades then values between 0x00 and 0xff may be used (display driver dependent).

Parameters

- `dev` – Auxiliary display device instance
- `character` – Pointer to custom character structure

Return values

- `0` – on success.
- `-ENOSYS` – if not supported/implemented.
- `-EINVAL` – if provided argument is invalid.
- `-errno` – Negative errno code on other failure.

int `auxdisplay_write`(const struct *device* *dev, const uint8_t *data, uint16_t len)

Write data to auxiliary display screen at current position.

Parameters

- `dev` – Auxiliary display device instance
- `data` – Text data to write

- `len` – Length of text data to write

Return values

- `0` – on success.
- `-EINVAL` – if provided argument is invalid.
- `-errno` – Negative errno code on other failure.

int `auxdisplay_custom_command`(const struct *device* *dev, struct *auxdisplay_custom_data* *data)

Send a custom command to the display (if supported by driver)

Parameters

- `dev` – Auxiliary display device instance
- `data` – Custom command structure (this may be extended by specific drivers)

Return values

- `0` – on success.
- `-ENOSYS` – if not supported/implemented.
- `-EINVAL` – if provided argument is invalid.
- `-errno` – Negative errno code on other failure.

struct `auxdisplay_light`

#include <auxdisplay.h> Light levels for brightness and/or backlight.

If not supported by a display/driver, both minimum and maximum will be AUXDISPLAY_LIGHT_NOT_SUPPORTED.

Public Members

uint8_t `minimum`

Minimum light level supported.

uint8_t `maximum`

Maximum light level supported.

struct `auxdisplay_capabilities`

#include <auxdisplay.h> Structure holding display capabilities.

Public Members

uint16_t `columns`

Number of character columns.

uint16_t `rows`

Number of character rows.

auxdisplay_mode_t **mode**
Display-specific data (e.g.

4-bit or 8-bit mode for HD44780-based displays)

struct *auxdisplay_light* **brightness**
Brightness details for display (if supported)

struct *auxdisplay_light* **backlight**
Backlight details for display (if supported)

uint8_t **custom_characters**
Number of custom characters supported by display (0 if unsupported)

uint8_t **custom_character_width**
Width (in pixels) of a custom character, supplied custom characters should match.

uint8_t **custom_character_height**
Height (in pixels) of a custom character, supplied custom characters should match.

struct **auxdisplay_custom_data**
#include <*auxdisplay.h*> Structure for a custom command.

This may be extended by specific drivers.

Public Members

uint8_t ***data**
Raw command data to be sent.

uint16_t **len**
Length of supplied data.

uint32_t **options**
Display-driver specific options for command.

struct **auxdisplay_character**
#include <*auxdisplay.h*> Structure for a custom character.

Public Members

uint8_t **index**
Custom character index on the display.

uint8_t ***data**
Custom character pixel data, a character must be valid for a display consisting of a uint8 array of size character width by character height, values should be 0x00 for pixel off or 0xff for pixel on, if a display supports shades then values between 0x00 and 0xff may be used (display driver dependent).

uint8_t **character_code**

Will be updated with custom character index to use in the display write function to disaplay this custom character.

7.6.4 Audio

Audio Codec

Overview The Audio Codec API provides access to digital audio codecs.

Configuration Options Related configuration options:

- CONFIG_AUDIO_CODEC

API Reference

group **audio_codec_interface**

Abstraction for audio codecs.

Since
1.13

Version
0.1.0

Typedefs

typedef void (***audio_codec_error_callback_t**)(const struct *device* *dev, uint32_t errors)

Callback for error interrupt.

Param dev
Pointer to the codec device

Param errors
Device errors (bitmask of *audio_codec_error_type* values)

Enums

enum **audio_pcm_rate_t**

PCM audio sample rates.

Values:

enumerator **AUDIO_PCM_RATE_8K** = 8000

8 kHz sample rate

enumerator **AUDIO_PCM_RATE_16K** = 16000

16 kHz sample rate

enumerator AUDIO_PCM_RATE_24K = 24000
24 kHz sample rate

enumerator AUDIO_PCM_RATE_32K = 32000
32 kHz sample rate

enumerator AUDIO_PCM_RATE_44P1K = 44100
44.1 kHz sample rate

enumerator AUDIO_PCM_RATE_48K = 48000
48 kHz sample rate

enumerator AUDIO_PCM_RATE_96K = 96000
96 kHz sample rate

enumerator AUDIO_PCM_RATE_192K = 192000
192 kHz sample rate

enum audio_pcm_width_t
PCM audio sample bit widths.
Values:

enumerator AUDIO_PCM_WIDTH_16_BITS = 16
16-bit sample width

enumerator AUDIO_PCM_WIDTH_20_BITS = 20
20-bit sample width

enumerator AUDIO_PCM_WIDTH_24_BITS = 24
24-bit sample width

enumerator AUDIO_PCM_WIDTH_32_BITS = 32
32-bit sample width

enum audio_dai_type_t
Digital Audio Interface (DAI) type.
Values:

enumerator AUDIO_DAI_TYPE_I2S
I2S Interface.

enumerator AUDIO_DAI_TYPE_INVALID
Other interfaces can be added here.

enum audio_property_t
Codec properties that can be set by *audio_codec_set_property()*.
Values:

enumerator AUDIO_PROPERTY_OUTPUT_VOLUME
Output volume.

enumerator AUDIO_PROPERTY_OUTPUT_MUTE
Output mute/unmute.

enum audio_channel_t
Audio channel identifiers to use in *audio_codec_set_property()*.

Values:

enumerator AUDIO_CHANNEL_FRONT_LEFT
Front left channel.

enumerator AUDIO_CHANNEL_FRONT_RIGHT
Front right channel.

enumerator AUDIO_CHANNEL_LFE
Low frequency effect channel.

enumerator AUDIO_CHANNEL_FRONT_CENTER
Front center channel.

enumerator AUDIO_CHANNEL_REAR_LEFT
Rear left channel.

enumerator AUDIO_CHANNEL_REAR_RIGHT
Rear right channel.

enumerator AUDIO_CHANNEL_REAR_CENTER
Rear center channel.

enumerator AUDIO_CHANNEL_SIDE_LEFT
Side left channel.

enumerator AUDIO_CHANNEL_SIDE_RIGHT
Side right channel.

enumerator AUDIO_CHANNEL_ALL
All channels.

enum audio_codec_error_type
Codec error type.

Values:

enumerator AUDIO_CODEC_ERROR_OVERCURRENT = *BIT*(0)
Output over-current.

enumerator AUDIO_CODEC_ERROR_OVERTEMPERATURE = *BIT*(1)

Codec over-temperature.

enumerator AUDIO_CODEC_ERROR_UNDERVOLTAGE = *BIT*(2)

Power low voltage.

enumerator AUDIO_CODEC_ERROR_OVERVOLTAGE = *BIT*(3)

Power high voltage.

enumerator AUDIO_CODEC_ERROR_DC = *BIT*(4)

Output direct-current.

Functions

static inline int **audio_codec_configure**(const struct *device* *dev, struct *audio_codec_cfg* *cfg)

Configure the audio codec.

Configure the audio codec device according to the configuration parameters provided as input

Parameters

- **dev** – Pointer to the device structure for codec driver instance.
- **cfg** – Pointer to the structure containing the codec configuration.

Returns

0 on success, negative error code on failure

static inline void **audio_codec_start_output**(const struct *device* *dev)

Set codec to start output audio playback.

Setup the audio codec device to start the audio playback

Parameters

- **dev** – Pointer to the device structure for codec driver instance.

static inline void **audio_codec_stop_output**(const struct *device* *dev)

Set codec to stop output audio playback.

Setup the audio codec device to stop the audio playback

Parameters

- **dev** – Pointer to the device structure for codec driver instance.

static inline int **audio_codec_set_property**(const struct *device* *dev, *audio_property_t* property, *audio_channel_t* channel, *audio_property_value_t* val)

Set a codec property defined by audio_property_t.

Set a property such as volume level, clock configuration etc.

Parameters

- **dev** – Pointer to the device structure for codec driver instance.
- **property** – The codec property to set
- **channel** – The audio channel for which the property has to be set

- `val` – pointer to a property value of type audio_codec_property_value_t

Returns
0 on success, negative error code on failure

static inline int **audio_codec_apply_properties**(const struct *device* *dev)

Atomically apply any cached properties.

Following one or more invocations of audio_codec_set_property, that may have been cached by the driver, audio_codec_apply_properties can be invoked to apply all the properties as atomic as possible

Parameters

- `dev` – Pointer to the device structure for codec driver instance.

Returns
0 on success, negative error code on failure

static inline int **audio_codec_clear_errors**(const struct *device* *dev)

Clear any codec errors.

Clear all codec errors. If an error interrupt exists, it will be de-asserted.

Parameters

- `dev` – Pointer to the device structure for codec driver instance.

Returns
0 on success, negative error code on failure

static inline int **audio_codec_register_error_callback**(const struct *device* *dev, *audio_codec_error_callback_t* cb)

Register a callback function for codec error.

The callback will be called from a thread, so I2C or SPI operations are safe. However, the thread's stack is limited and defined by the driver. It is currently up to the caller to ensure that the callback does not overflow the stack.

Parameters

- `dev` – Pointer to the audio codec device
- `cb` – The function that should be called when an error is detected fires

Returns
0 if successful, negative errno code if failure.

union **audio_dai_cfg_t**

#include <*codec.h*> Digital Audio Interface Configuration.

Configuration is dependent on DAI type

Public Members

struct *i2s_config* **i2s**

I2S configuration.

struct **audio_codec_cfg**

#include <*codec.h*> Codec configuration parameters.

Public Members

uint32_t **mclk_freq**
MCLK input frequency in Hz.

audio_dai_type_t **dai_type**
Digital interface type.

audio_dai_cfg_t **dai_cfg**
DAI configuration info.

union **audio_property_value_t**
#include <*codec.h*> Codec property values.

Public Members

int **vol**
Volume level in 0.5dB resolution.

bool **mute**
Mute if *true*, unmute if *false*.

Digital Microphone (DMIC)

Overview The audio DMIC interface provides access to digital microphones.

Configuration Options Related configuration options:

- CONFIG_AUDIO_DMIC

Related code samples

Digital Microphone (DMIC)
Perform PDM transfers using different configurations.

X-NUCLEO-IKS02A1 shield - MEMS microphone
Acquire audio using the digital MEMS microphone on X-NUCLEO-IKS02A1 shield.

API Reference

group **audio_dmic_interface**
Abstraction for digital microphones.

Since
1.13

Version
0.1.0

Enums

enum dmic_state

DMIC driver states.

Values:

enumerator DMIC_STATE_UNINIT

Uninitialized.

enumerator DMIC_STATE_INITIALIZED

Initialized.

enumerator DMIC_STATE_CONFIGURED

Configured.

enumerator DMIC_STATE_ACTIVE

Active.

enumerator DMIC_STATE_PAUSED

Paused.

enumerator DMIC_STATE_ERROR

Error.

enum dmic_trigger

DMIC driver trigger commands.

Values:

enumerator DMIC_TRIGGER_STOP

Stop stream.

enumerator DMIC_TRIGGER_START

Start stream.

enumerator DMIC_TRIGGER_PAUSE

Pause stream.

enumerator DMIC_TRIGGER_RELEASE

Release paused stream.

enumerator DMIC_TRIGGER_RESET

Reset stream.

enum pdm_lr

PDM Channels LEFT / RIGHT.

Values:

enumerator PDM_CHAN_LEFT

Left channel.

enumerator PDM_CHAN_RIGHT

Right channel.

Functions

static inline uint32_t dmic_build_channel_map(uint8_t channel, uint8_t pdm, enum *pdm_lr* lr)

Build the channel map to populate struct *pdm_chan_cfg*.

Returns the map of PDM controller and LEFT/RIGHT channel shifted to the bit position corresponding to the input logical channel value

Parameters

- channel – The logical channel number
- pdm – The PDM hardware controller number
- lr – LEFT/RIGHT channel within the chosen PDM hardware controller

Returns

Bit-map containing the PDM and L/R channel information

static inline void dmic_parse_channel_map(uint32_t channel_map_lo, uint32_t channel_map_hi, uint8_t channel, uint8_t *pdm, enum *pdm_lr* *lr)

Helper function to parse the channel map in *pdm_chan_cfg*.

Returns the PDM controller and LEFT/RIGHT channel corresponding to the channel map and the logical channel provided as input

Parameters

- channel_map_lo – Lower order/significant bits of the channel map
- channel_map_hi – Higher order/significant bits of the channel map
- channel – The logical channel number
- pdm – Pointer to the PDM hardware controller number
- lr – Pointer to the LEFT/RIGHT channel within the PDM controller

static inline uint32_t dmic_build_clk_skew_map(uint8_t pdm, uint8_t skew)

Build a bit map of clock skew values for each PDM channel.

Returns the bit-map of clock skew value shifted to the bit position corresponding to the input PDM controller value

Parameters

- pdm – The PDM hardware controller number
- skew – The skew to apply for the clock output from the PDM controller

Returns

Bit-map containing the clock skew information

static inline int **dmic_configure**(const struct *device* *dev, struct *dmic_cfg* *cfg)

Configure the DMIC driver and controller(s)

Configures the DMIC driver device according to the number of channels, channel mapping, PDM I/O configuration, PCM stream configuration, etc.

Parameters

- **dev** – Pointer to the device structure for DMIC driver instance
- **cfg** – Pointer to the structure containing the DMIC configuration

Returns
0 on success, a negative error code on failure

static inline int **dmic_trigger**(const struct *device* *dev, enum *dmic_trigger* cmd)

Send a command to the DMIC driver.

Sends a command to the driver to perform a specific action

Parameters

- **dev** – Pointer to the device structure for DMIC driver instance
- **cmd** – The command to be sent to the driver instance

Returns
0 on success, a negative error code on failure

static inline int **dmic_read**(const struct *device* *dev, uint8_t stream, void **buffer, size_t *size, int32_t timeout)

Read received decimated PCM data stream.

Optionally waits for audio to be received and provides the received audio buffer from the requested stream

Parameters

- **dev** – Pointer to the device structure for DMIC driver instance
- **stream** – Stream identifier
- **buffer** – Pointer to the received buffer address
- **size** – Pointer to the received buffer size
- **timeout** – Timeout in milliseconds to wait in case audio is not yet received, or SYS_FOREVER_MS

Returns
0 on success, a negative error code on failure

struct **pdm_io_cfg**

#include <dmic.h> PDM Input/Output signal configuration.

Parameters common to all PDM controllers

uint32_t **min_pdm_clk_freq**

Minimum clock frequency supported by the mic.

uint32_t **max_pdm_clk_freq**

Maximum clock frequency supported by the mic.

uint8_t **min_pdm_clk_dc**

Minimum duty cycle in % supported by the mic.

uint8_t **max_pdm_clk_dc**

Maximum duty cycle in % supported by the mic.

Parameters unique to each PDM controller

uint8_t **pdm_clk_pol**

Bit mask to optionally invert PDM clock.

uint8_t **pdm_data_pol**

Bit mask to optionally invert mic data.

uint32_t **pdm_clk_skew**

Collection of clock skew values for each PDM port.

struct **pcm_stream_cfg**

#include <*dmic.h*> Configuration of the PCM streams to be output by the PDM hardware.

> **Note**
>
> if either *pcm_rate* or *pcm_width* is set to 0 for a stream, the stream would be disabled

Public Members

uint32_t **pcm_rate**

PCM sample rate of stream.

uint8_t **pcm_width**

PCM sample width of stream.

uint16_t **block_size**

PCM sample block size per transfer.

struct k_mem_slab ***mem_slab**

SLAB for DMIC driver to allocate buffers for stream.

struct **pdm_chan_cfg**

#include <*dmic.h*> Mapping/ordering of the PDM channels to logical PCM output channel.

Since each controller can have 2 audio channels (stereo), there can be a total of 8x2=16 channels. The actual number of channels shall be described in *act_num_chan*.

If 2 streams are enabled, the channel order will be the same for both streams.

Each channel is described as a 4-bit number, the least significant bit indicates LEFT/RIGHT selection of the PDM controller.

The most significant 3 bits indicate the PDM controller number:

- bits 0-3 are for channel 0, bit 0 indicates LEFT or RIGHT
- bits 4-7 are for channel 1, bit 4 indicates LEFT or RIGHT and so on.

CONSTRAINT: The LEFT and RIGHT channels of EACH PDM controller needs to be adjacent to each other.

Requested channel map

uint32_t **req_chan_map_lo**
Channels 0 to 7.

uint32_t **req_chan_map_hi**
Channels 8 to 15.

Actual channel map that the driver could configure

uint32_t **act_chan_map_lo**
Channels 0 to 7.

uint32_t **act_chan_map_hi**
Channels 8 to 15.

Public Members

uint8_t **req_num_chan**
Requested number of channels.

uint8_t **act_num_chan**
Actual number of channels that the driver could configure.

uint8_t **req_num_streams**
Requested number of streams for each channel.

uint8_t **act_num_streams**
Actual number of streams that the driver could configure.

struct **dmic_cfg**
#include <*dmic.h*> Input configuration structure for the DMIC configuration API.

Public Members

struct *pcm_stream_cfg* ***streams**
Array of *pcm_stream_cfg* for application to provide configuration for each stream.

Inter-IC Sound (I2S) Bus

Overview The I2S (Inter-IC Sound) API provides support for the standard I2S interface as well as common non-standard extensions such as PCM Short/Long Frame Sync and Left/Right Justified Data Formats.

Configuration Options Related configuration options:

- `CONFIG_I2S`

Related code samples

I2S echo
Process an audio stream to add an echo effect.

I2S output
Send I2S output stream

USB Audio asynchronous explicit feedback sample
USB Audio 2 explicit feedback sample playing audio on I2S.

API Reference

group `i2s_interface`

I2S (Inter-IC Sound) Interface.

Since
1.9

Version
1.0.0

The I2S API provides support for the standard I2S interface standard as well as common non-standard extensions such as PCM Short/Long Frame Sync, Left/Right Justified Data Format.

Defines

`I2S_FMT_DATA_FORMAT_SHIFT`

Data Format bit field position.

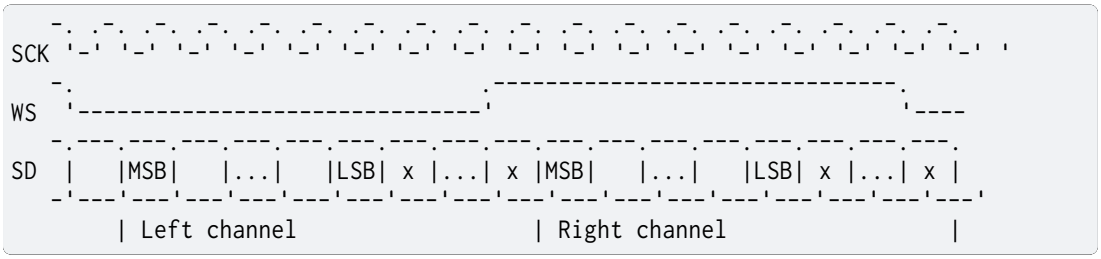
`I2S_FMT_DATA_FORMAT_MASK`

Data Format bit field mask.

`I2S_FMT_DATA_FORMAT_I2S`

Standard I2S Data Format.

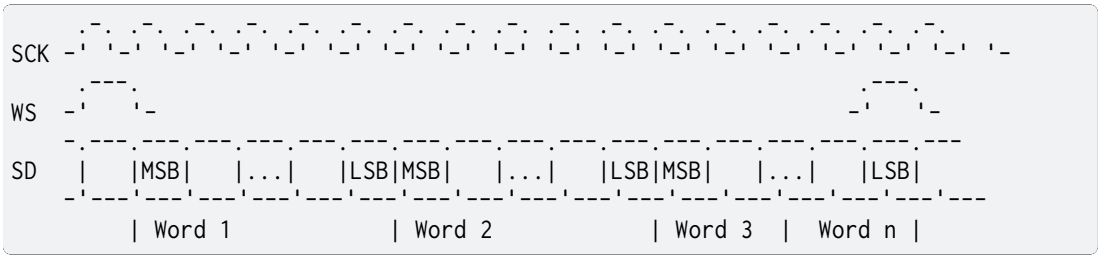
Serial data is transmitted in two's complement with the MSB first. Both Word Select (WS) and Serial Data (SD) signals are sampled on the rising edge of the clock signal (SCK). The MSB is always sent one clock period after the WS changes. Left channel data are sent first indicated by WS = 0, followed by right channel data indicated by WS = 1.



I2S_FMT_DATA_FORMAT_PCM_SHORT

PCM Short Frame Sync Data Format.

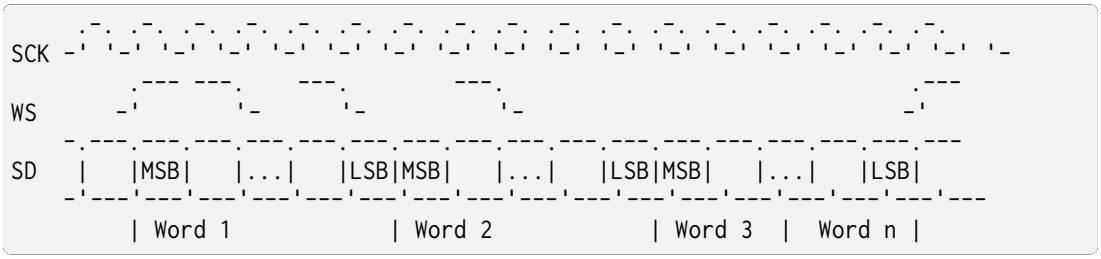
Serial data is transmitted in two's complement with the MSB first. Both Word Select (WS) and Serial Data (SD) signals are sampled on the falling edge of the clock signal (SCK). The falling edge of the frame sync signal (WS) indicates the start of the PCM word. The frame sync is one clock cycle long. An arbitrary number of data words can be sent in one frame.



I2S_FMT_DATA_FORMAT_PCM_LONG

PCM Long Frame Sync Data Format.

Serial data is transmitted in two's complement with the MSB first. Both Word Select (WS) and Serial Data (SD) signals are sampled on the falling edge of the clock signal (SCK). The rising edge of the frame sync signal (WS) indicates the start of the PCM word. The frame sync has an arbitrary length, however it has to fall before the start of the next frame. An arbitrary number of data words can be sent in one frame.



I2S_FMT_DATA_FORMAT_LEFT_JUSTIFIED

Left Justified Data Format.

Serial data is transmitted in two's complement with the MSB first. Both Word Select (WS) and Serial Data (SD) signals are sampled on the rising edge of the clock signal (SCK). The bits within the data word are left justified such that the MSB is always sent in the clock period following the WS transition. Left channel data are sent first indicated by WS = 1, followed by right channel data indicated by WS = 0.



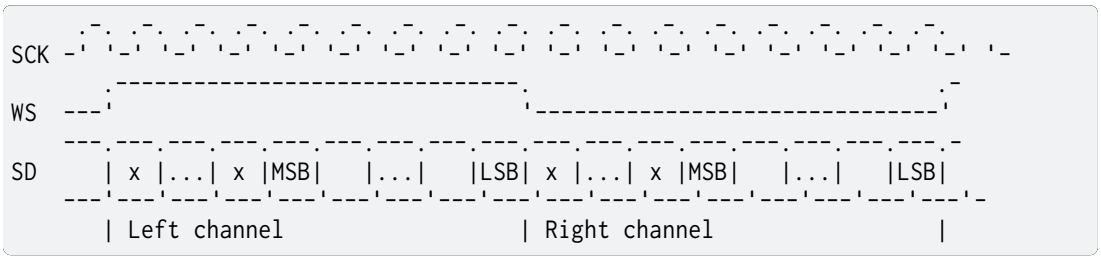
(continues on next page)

(continued from previous page)



I2S_FMT_DATA_FORMAT_RIGHT_JUSTIFIED
: Right Justified Data Format.

  Serial data is transmitted in two's complement with the MSB first. Both Word Select (WS) and Serial Data (SD) signals are sampled on the rising edge of the clock signal (SCK). The bits within the data word are right justified such that the LSB is always sent in the clock period preceding the WS transition. Left channel data are sent first indicated by WS = 1, followed by right channel data indicated by WS = 0.



I2S_FMT_DATA_ORDER_MSB
: Send MSB first.

I2S_FMT_DATA_ORDER_LSB
: Send LSB first.

I2S_FMT_DATA_ORDER_INV
: Invert bit ordering, send LSB first.

I2S_FMT_CLK_FORMAT_SHIFT
: Data Format bit field position.

I2S_FMT_CLK_FORMAT_MASK
: Data Format bit field mask.

I2S_FMT_BIT_CLK_INV
: Invert bit clock.

I2S_FMT_FRAME_CLK_INV
: Invert frame clock.

I2S_FMT_CLK_NF_NB
: Normal Frame, Normal Bit Clk.

I2S_FMT_CLK_NF_IB
: Normal Frame, Inverted Bit Clk.

I2S_FMT_CLK_IF_NB
: Inverted Frame, Normal Bit Clk.

I2S_FMT_CLK_IF_IB
: Inverted Frame, Inverted Bit Clk.

I2S_OPT_BIT_CLK_CONT
: Run bit clock continuously.

I2S_OPT_BIT_CLK_GATED
: Run bit clock when sending data only.

I2S_OPT_BIT_CLK_MASTER
: I2S driver is bit clock master.

I2S_OPT_BIT_CLK_SLAVE
: I2S driver is bit clock slave.

I2S_OPT_FRAME_CLK_MASTER
: I2S driver is frame clock master.

I2S_OPT_FRAME_CLK_SLAVE
: I2S driver is frame clock slave.

I2S_OPT_LOOPBACK
: Loop back mode.

  In loop back mode RX input will be connected internally to TX output. This is used primarily for testing.

I2S_OPT_PINGPONG
: Ping pong mode.

  In ping pong mode TX output will keep alternating between a ping buffer and a pong buffer. This is normally used in audio streams when one buffer is being populated while the other is being played (DMAed) and vice versa. So, in this mode, 2 sets of buffers fixed in size are used. Static Arrays are used to achieve this and hence they are never freed.

Typedefs

typedef uint8_t **i2s_fmt_t**
: I2S data stream format options.

typedef uint8_t **i2s_opt_t**
: I2S configuration options.

Enums

enum **i2s_dir**
: I2C Direction.

  Values:

enumerator I2S_DIR_RX
Receive data.

enumerator I2S_DIR_TX
Transmit data.

enumerator I2S_DIR_BOTH
Both receive and transmit data.

enum i2s_state
Interface state.
Values:

enumerator I2S_STATE_NOT_READY
The interface is not ready.

```
The interface was initialized but is not yet ready to receive /
transmit data. Call i2s_configure() to configure interface and change
its state to READY.
```

enumerator I2S_STATE_READY
The interface is ready to receive / transmit data.

enumerator I2S_STATE_RUNNING
The interface is receiving / transmitting data.

enumerator I2S_STATE_STOPPING
The interface is draining its transmit queue.

enumerator I2S_STATE_ERROR
TX buffer underrun or RX buffer overrun has occurred.

enum i2s_trigger_cmd
Trigger command.
Values:

enumerator I2S_TRIGGER_START
Start the transmission / reception of data.

```
If I2S_DIR_TX is set some data has to be queued for transmission by
the i2s_write() function. This trigger can be used in READY state
only and changes the interface state to RUNNING.
```

enumerator I2S_TRIGGER_STOP
Stop the transmission / reception of data.

```
Stop the transmission / reception of data at the end of the current
memory block. This trigger can be used in RUNNING state only and at
first changes the interface state to STOPPING. When the current TX /
RX block is transmitted / received the state is changed to READY.
```

(continues on next page)

(continued from previous page)

```
Subsequent START trigger will resume transmission / reception where
it stopped.
```

enumerator I2S_TRIGGER_DRAIN

Empty the transmit queue.

```
Send all data in the transmit queue and stop the transmission.
If the trigger is applied to the RX queue it has the same effect as
I2S_TRIGGER_STOP. This trigger can be used in RUNNING state only and
at first changes the interface state to STOPPING. When all TX blocks
are transmitted the state is changed to READY.
```

enumerator I2S_TRIGGER_DROP

Discard the transmit / receive queue.

```
Stop the transmission / reception immediately and discard the
contents of the respective queue. This trigger can be used in any
state other than NOT_READY and changes the interface state to READY.
```

enumerator I2S_TRIGGER_PREPARE

Prepare the queues after underrun/overrun error has occurred.

```
This trigger can be used in ERROR state only and changes the
interface state to READY.
```

Functions

int **i2s_configure**(const struct *device* *dev, enum *i2s_dir* dir, const struct *i2s_config* *cfg)

Configure operation of a host I2S controller.

The dir parameter specifies if Transmit (TX) or Receive (RX) direction will be configured by data provided via cfg parameter.

The function can be called in NOT_READY or READY state only. If executed successfully the function will change the interface state to READY.

If the function is called with the parameter cfg->frame_clk_freq set to 0 the interface state will be changed to NOT_READY.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **dir** – Stream direction: RX, TX, or both, as defined by I2S_DIR_*. The I2S_DIR_BOTH value may not be supported by some drivers. For those, the RX and TX streams need to be configured separately.
- **cfg** – Pointer to the structure containing configuration parameters.

Return values

- **0** – If successful.
- **-EINVAL** – Invalid argument.
- **-ENOSYS** – I2S_DIR_BOTH value is not supported.

static inline const struct *i2s_config* *`i2s_config_get`(const struct *device* *dev, enum *i2s_dir* dir)

Fetch configuration information of a host I2S controller.

Parameters

- `dev` – Pointer to the device structure for the driver instance
- `dir` – Stream direction: RX or TX as defined by I2S_DIR_*

Return values

`Pointer` – to the structure containing configuration parameters, or NULL if un-configured

static inline int `i2s_read`(const struct *device* *dev, void **mem_block, size_t *size)

Read data from the RX queue.

Data received by the I2S interface is stored in the RX queue consisting of memory blocks preallocated by this function from rx_mem_slab (as defined by i2s_configure). Ownership of the RX memory block is passed on to the user application which has to release it.

The data is read in chunks equal to the size of the memory block. If the interface is in READY state the number of bytes read can be smaller.

If there is no data in the RX queue the function will block waiting for the next RX memory block to fill in. This operation can timeout as defined by i2s_configure. If the timeout value is set to K_NO_WAIT the function is non-blocking.

Reading from the RX queue is possible in any state other than NOT_READY. If the interface is in the ERROR state it is still possible to read all the valid data stored in RX queue. Afterwards the function will return -EIO error.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `mem_block` – Pointer to the RX memory block containing received data.
- `size` – Pointer to the variable storing the number of bytes read.

Return values

- `0` – If successful.
- `-EIO` – The interface is in NOT_READY or ERROR state and there are no more data blocks in the RX queue.
- `-EBUSY` – Returned without waiting.
- `-EAGAIN` – Waiting period timed out.

int `i2s_buf_read`(const struct *device* *dev, void *buf, size_t *size)

Read data from the RX queue into a provided buffer.

Data received by the I2S interface is stored in the RX queue consisting of memory blocks preallocated by this function from rx_mem_slab (as defined by i2s_configure). Calling this function removes one block from the queue which is copied into the provided buffer and then freed.

The provided buffer must be large enough to contain a full memory block of data, which is parameterized for the channel via *i2s_configure()*.

This function is otherwise equivalent to *i2s_read()*.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

- `buf` – Destination buffer for read data, which must be at least the as large as the configured memory block size for the RX channel.
- `size` – Pointer to the variable storing the number of bytes read.

Return values

- `0` – If successful.
- `-EIO` – The interface is in NOT_READY or ERROR state and there are no more data blocks in the RX queue.
- `-EBUSY` – Returned without waiting.
- `-EAGAIN` – Waiting period timed out.

static inline int `i2s_write`(const struct *device* *dev, void *mem_block, size_t size)

Write data to the TX queue.

Data to be sent by the I2S interface is stored first in the TX queue. TX queue consists of memory blocks preallocated by the user from tx_mem_slab (as defined by i2s_configure). This function takes ownership of the memory block and will release it when all data are transmitted.

If there are no free slots in the TX queue the function will block waiting for the next TX memory block to be send and removed from the queue. This operation can timeout as defined by i2s_configure. If the timeout value is set to K_NO_WAIT the function is non-blocking.

Writing to the TX queue is only possible if the interface is in READY or RUNNING state.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `mem_block` – Pointer to the TX memory block containing data to be sent.
- `size` – Number of bytes to write. This value has to be equal or smaller than the size of the memory block.

Return values

- `0` – If successful.
- `-EIO` – The interface is not in READY or RUNNING state.
- `-EBUSY` – Returned without waiting.
- `-EAGAIN` – Waiting period timed out.

int `i2s_buf_write`(const struct *device* *dev, void *buf, size_t size)

Write data to the TX queue from a provided buffer.

This function acquires a memory block from the I2S channel TX queue and copies the provided data buffer into it. It is otherwise equivalent to *i2s_write()*.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `buf` – Pointer to a buffer containing the data to transmit.
- `size` – Number of bytes to write. This value has to be equal or smaller than the size of the channel's TX memory block configuration.

Return values

- `0` – If successful.
- `-EIO` – The interface is not in READY or RUNNING state.
- `-EBUSY` – Returned without waiting.

- `-EAGAIN` – Waiting period timed out.
- `-ENOMEM` – No memory in TX slab queue.
- `-EINVAL` – Size parameter larger than TX queue memory block.

int `i2s_trigger`(const struct *device* *dev, enum *i2s_dir* dir, enum *i2s_trigger_cmd* cmd)

Send a trigger command.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `dir` – Stream direction: RX, TX, or both, as defined by I2S_DIR_*. The I2S_DIR_BOTH value may not be supported by some drivers. For those, triggering need to be done separately for the RX and TX streams.
- `cmd` – Trigger command.

Return values

- `0` – If successful.
- `-EINVAL` – Invalid argument.
- `-EIO` – The trigger cannot be executed in the current state or a DMA channel cannot be allocated.
- `-ENOMEM` – RX/TX memory block not available.
- `-ENOSYS` – I2S_DIR_BOTH value is not supported.

struct `i2s_config`

#include <i2s.h> Interface configuration options.

Memory slab pointed to by the mem_slab field has to be defined and initialized by the user. For I2S driver to function correctly number of memory blocks in a slab has to be at least 2 per queue. Size of the memory block should be multiple of frame_size where frame_size = (channels * word_size_bytes). As an example 16 bit word will occupy 2 bytes, 24 or 32 bit word will occupy 4 bytes.

Please check Zephyr Kernel Primer for more information on memory slabs.

Remark

When I2S data format is selected parameter channels is ignored, number of words in a frame is always 2.

Public Members

uint8_t `word_size`

Number of bits representing one data word.

uint8_t `channels`

Number of words per frame.

i2s_fmt_t `format`

Data stream format as defined by I2S_FMT_* constants.

i2s_opt_t **options**

Configuration options as defined by I2S_OPT_* constants.

uint32_t **frame_clk_freq**

Frame clock (WS) frequency, this is sampling rate.

struct k_mem_slab ***mem_slab**

Memory slab to store RX/TX data.

size_t **block_size**

Size of one RX/TX memory block (buffer) in bytes.

int32_t **timeout**

Read/Write timeout.

Number of milliseconds to wait in case TX queue is full or RX queue is empty, or 0, or SYS_FOREVER_MS.

Digital Audio Interface (DAI)

Overview The DAI (Digital Audio Interface) is a generic high level API for audio drivers. It can be configured with bespoke data for vendor specific configuration.

Configuration Options Related configuration options:

- CONFIG_DAI

API Reference

group **dai_interface**

DAI Interface.

Since
3.1

Version
0.1.0

The DAI API provides support for the standard I2S (SSP) and its common variants. It supports also DMIC, HDA and SDW backends. The API has a config function with bespoke data argument for device/vendor specific config. There are also optional timestamping functions to get device specific audio clock time.

Defines

DAI_FORMAT_CLOCK_PROVIDER_MASK

Used to extract the clock configuration from the format attribute of struct *dai_config*.

DAI_FORMAT_PROTOCOL_MASK

Used to extract the protocol from the format attribute of struct *dai_config*.

DAI_FORMAT_CLOCK_INVERSION_MASK

Used to extract the clock inversion from the format attribute of struct *dai_config*.

Enums

enum dai_clock_provider

DAI clock configurations.

This is used to describe all of the possible clock-related configurations w.r.t the DAI and the codec.

Values:

enumerator DAI_CBP_CFP = (0 « 12)

codec BLCK provider, codec FSYNC provider

codec BCLK consumer, codec FSYNC provider

enumerator DAI_CBC_CFP = (2 « 12)

codec BCLK provider, codec FSYNC consumer

enumerator DAI_CBP_CFC = (3 « 12)

codec BCLK consumer, codec FSYNC consumer

enumerator DAI_CBC_CFC = (4 « 12)

enum dai_protocol

DAI protocol.

The communication between the DAI and the CODEC may use different protocols depending on the scenario.

Values:

enumerator DAI_PROTO_I2S = 1

I2S.

enumerator DAI_PROTO_RIGHT_J

Right Justified.

enumerator DAI_PROTO_LEFT_J

Left Justified.

enumerator DAI_PROTO_DSP_A

TDM, FSYNC asserted 1 BCLK early.

enumerator DAI_PROTO_DSP_B

TDM, FSYNC asserted at the same time as MSB.

enumerator DAI_PROTO_PDM

Pulse Density Modulation.

enum dai_clock_inversion

DAI clock inversion.

Some applications may require a different clock polarity (FSYNC/BCLK) compared to the default one chosen based on the protocol.

Values:

enumerator DAI_INVERSION_NB_NF = 0

no BCLK inversion, no FSYNC inversion

no BCLK inversion, FSYNC inversion

enumerator DAI_INVERSION_NB_IF = (2 « 8)

BCLK inversion, no FSYNC inversion.

enumerator DAI_INVERSION_IB_NF = (3 « 8)

BCLK inversion, FSYNC inversion.

enumerator DAI_INVERSION_IB_IF = (4 « 8)

enum dai_type

Types of DAI.

The type of the DAI. This ID type is used to configure bespoke DAI HW settings.

DAIs have a lot of physical link feature variability and therefore need different configuration data to cater for different use cases. We usually need to pass extra bespoke configuration prior to DAI start.

Values:

enumerator DAI_LEGACY_I2S = 0

Legacy I2S compatible with i2s.h.

enumerator DAI_INTEL_SSP

Intel SSP.

enumerator DAI_INTEL_DMIC

Intel DMIC.

enumerator DAI_INTEL_HDA

Intel HD/A.

enumerator DAI_INTEL_ALH

Intel ALH.

enumerator DAI_IMX_SAI

i.MX SAI

enumerator DAI_IMX_ESAI

i.MX ESAI

enumerator DAI_AMD_BT
Amd BT.

enumerator DAI_AMD_SP
Amd SP.

enumerator DAI_AMD_DMIC
Amd DMIC.

enumerator DAI_MEDIATEK_AFE
Mtk AFE.

enumerator DAI_INTEL_SSP_NHLT
nhlt ssp

enumerator DAI_INTEL_DMIC_NHLT
nhlt ssp

enumerator DAI_INTEL_HDA_NHLT
nhlt Intel HD/A

enumerator DAI_INTEL_ALH_NHLT
nhlt Intel ALH

enum dai_dir
DAI Direction.
Values:

enumerator DAI_DIR_TX = 0
Transmit data.

enumerator DAI_DIR_RX
Receive data.

enumerator DAI_DIR_BOTH
Both receive and transmit data.

enum dai_state
Interface state.
Values:

enumerator DAI_STATE_NOT_READY = 0
The interface is not ready.

```
The interface was initialized but is not yet ready to receive /
transmit data. Call dai_config_set() to configure interface and change
its state to READY.
```

enumerator DAI_STATE_READY
The interface is ready to receive / transmit data.

enumerator DAI_STATE_RUNNING
The interface is receiving / transmitting data.

enumerator DAI_STATE_PRE_RUNNING
The interface is clocking but not receiving / transmitting data.

enumerator DAI_STATE_PAUSED
The interface paused.

enumerator DAI_STATE_STOPPING
The interface is draining its transmit queue.

enumerator DAI_STATE_ERROR
TX buffer underrun or RX buffer overrun has occurred.

enum dai_trigger_cmd
Trigger command.

Values:

enumerator DAI_TRIGGER_START = 0
Start the transmission / reception of data.

```
If DAI_DIR_TX is set some data has to be queued for transmission by
the dai_write() function. This trigger can be used in READY state
only and changes the interface state to RUNNING.
```

enumerator DAI_TRIGGER_PRE_START
Optional - Pre Start the transmission / reception of data.

```
Allows the DAI and downstream codecs to prepare for audio Tx/Rx by
starting any required clocks for downstream PLL/FLL locking.
```

enumerator DAI_TRIGGER_STOP
Stop the transmission / reception of data.

```
Stop the transmission / reception of data at the end of the current
memory block. This trigger can be used in RUNNING state only and at
first changes the interface state to STOPPING. When the current TX /
RX block is transmitted / received the state is changed to READY.
Subsequent START trigger will resume transmission / reception where
it stopped.
```

enumerator DAI_TRIGGER_PAUSE
Pause the transmission / reception of data.

```
Pause the transmission / reception of data at the end of the current
memory block. Behavior is implementation specific but usually this
state doesn't completely stop the clocks or transmission. The DAI could
be transmitting 0's (silence), but it is not consuming data from outside.
```

enumerator DAI_TRIGGER_POST_STOP

Optional - Post Stop the transmission / reception of data.

```
Allows the DAI and downstream codecs to shutdown cleanly after audio
Tx/Rx by stopping any required clocks for downstream audio completion.
```

enumerator DAI_TRIGGER_DRAIN

Empty the transmit queue.

```
Send all data in the transmit queue and stop the transmission.
If the trigger is applied to the RX queue it has the same effect as
DAI_TRIGGER_STOP. This trigger can be used in RUNNING state only and
at first changes the interface state to STOPPING. When all TX blocks
are transmitted the state is changed to READY.
```

enumerator DAI_TRIGGER_DROP

Discard the transmit / receive queue.

```
Stop the transmission / reception immediately and discard the
contents of the respective queue. This trigger can be used in any
state other than NOT_READY and changes the interface state to READY.
```

enumerator DAI_TRIGGER_PREPARE

Prepare the queues after underrun/overrun error has occurred.

```
This trigger can be used in ERROR state only and changes the
interface state to READY.
```

enumerator DAI_TRIGGER_RESET

Reset.

```
This trigger frees resources and moves the driver back to initial
state.
```

enumerator DAI_TRIGGER_COPY

Copy.

```
This trigger prepares for data copying.
```

Functions

static inline int **dai_probe**(const struct *device* *dev)

Probe operation of DAI driver.

The function will be called to power up the device and update for example possible reference count of the users. It can be used also to initialize internal variables and memory allocation.

Parameters

- **dev** – Pointer to the device structure for the driver instance.

Return values

0 – If successful.

static inline int **dai_remove**(const struct *device* *dev)

Remove operation of DAI driver.

The function will be called to unregister/unbind the device, for example to power down the device or decrease the usage reference count.

Parameters

- **dev** – Pointer to the device structure for the driver instance.

Return values

0 – If successful.

static inline int **dai_config_set**(const struct *device* *dev, const struct *dai_config* *cfg, const void *bespoke_cfg)

Configure operation of a DAI driver.

The dir parameter specifies if Transmit (TX) or Receive (RX) direction will be configured by data provided via cfg parameter.

The function can be called in NOT_READY or READY state only. If executed successfully the function will change the interface state to READY.

If the function is called with the parameter cfg->frame_clk_freq set to 0 the interface state will be changed to NOT_READY.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **cfg** – Pointer to the structure containing configuration parameters.
- **bespoke_cfg** – Pointer to the structure containing bespoke config.

Return values

- **0** – If successful.
- **-EINVAL** – Invalid argument.
- **-ENOSYS** – DAI_DIR_BOTH value is not supported.

static inline int **dai_config_get**(const struct *device* *dev, struct *dai_config* *cfg, enum *dai_dir* dir)

Fetch configuration information of a DAI driver.

Parameters

- **dev** – Pointer to the device structure for the driver instance
- **cfg** – Pointer to the config structure to be filled by the instance
- **dir** – Stream direction: RX or TX as defined by DAI_DIR_*

Return values

0 – if success, negative if invalid parameters or DAI un-configured

static inline const struct *dai_properties* ***dai_get_properties**(const struct *device* *dev, enum *dai_dir* dir, int stream_id)

Fetch properties of a DAI driver.

Parameters

- **dev** – Pointer to the device structure for the driver instance
- **dir** – Stream direction: RX or TX as defined by DAI_DIR_*
- **stream_id** – Stream id: some drivers may have stream specific properties, this id specifies the stream.

Return values
`Pointer` – to the structure containing properties, or NULL if error or no properties

static inline int **dai_trigger**(const struct *device* *dev, enum *dai_dir* dir, enum *dai_trigger_cmd* cmd)

Send a trigger command.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `dir` – Stream direction: RX, TX, or both, as defined by DAI_DIR_*. The DAI_DIR_BOTH value may not be supported by some drivers. For those, triggering need to be done separately for the RX and TX streams.
- `cmd` – Trigger command.

Return values

- `0` – If successful.
- `-EINVAL` – Invalid argument.
- `-EIO` – The trigger cannot be executed in the current state or a DMA channel cannot be allocated.
- `-ENOMEM` – RX/TX memory block not available.
- `-ENOSYS` – DAI_DIR_BOTH value is not supported.

static inline int **dai_ts_config**(const struct *device* *dev, struct *dai_ts_cfg* *cfg)

Configures timestamping in attached DAI.

Optional method.

Parameters

- `dev` – Component device.
- `cfg` – Timestamp config.

Return values
`0` – If successful.

static inline int **dai_ts_start**(const struct *device* *dev, struct *dai_ts_cfg* *cfg)

Starts timestamping.

Optional method

Parameters

- `dev` – Component device.
- `cfg` – Timestamp config.

Return values
`0` – If successful.

static inline int **dai_ts_stop**(const struct *device* *dev, struct *dai_ts_cfg* *cfg)

Stops timestamping.

Optional method.

Parameters

- `dev` – Component device.
- `cfg` – Timestamp config.

Return values
`0` – If successful.

static inline int `dai_ts_get`(const struct *device* *dev, struct *dai_ts_cfg* *cfg, struct *dai_ts_data* *tsd)

Gets timestamp.

Optional method.

Parameters

- `dev` – Component device.
- `cfg` – Timestamp config.
- `tsd` – Receives timestamp data.

Return values
`0` – If successful.

struct `dai_properties`

#include <*dai.h*> DAI properties.

This struct is used with APIs get_properties function to query DAI properties like fifo address and dma handshake. These are needed for example to setup dma outside the driver code.

Public Members

uint32_t `fifo_address`

Fifo hw address for e.g.

when connecting to dma.

uint32_t `fifo_depth`

Fifo depth.

uint32_t `dma_hs_id`

DMA handshake id.

uint32_t `reg_init_delay`

Delay for initializing registers.

int `stream_id`

Stream ID.

struct `dai_config`

#include <*dai.h*> Main DAI config structure.

Generic DAI interface configuration options.

Public Members

enum *dai_type* **type**
Type of the DAI.

uint32_t **dai_index**
Index of the DAI.

uint8_t **channels**
Number of audio channels, words in frame.

uint32_t **rate**
Frame clock (WS) frequency, sampling rate.

uint16_t **format**
DAI specific data stream format.

uint8_t **options**
DAI specific configuration options.

uint8_t **word_size**
Number of bits representing one data word.

size_t **block_size**
Size of one RX/TX memory block (buffer) in bytes.

uint16_t **link_config**
DAI specific link configuration.

tdm slot group number

struct **dai_ts_cfg**
#include <*dai.h*> DAI timestamp configuration.

Public Members

uint32_t **walclk_rate**
Rate in Hz, e.g.

19200000

int **type**
Type of the DAI (SSP, DMIC, HDA, etc.).

int **direction**
Direction (playback/capture)

int **index**
Index for SSPx to select correct timestamp register.

int **dma_id**
DMA instance id.

int **dma_chan_index**
Used DMA channel index.

int **dma_chan_count**
Number of channels in single DMA.

struct **dai_ts_data**
#include <*dai.h*> DAI timestamp data.

Public Members

uint64_t **walclk**
Wall clock.

uint64_t **sample**
Sample count.

uint32_t **walclk_rate**
Rate in Hz, e.g.
19200000

7.6.5 Battery Backed RAM (BBRAM)

The BBRAM APIs allow interfacing with the unique properties of this memory region. The following common types of BBRAM properties are easily accessed via this API:

- IBBR (invalid) state - check that the BBRAM is not corrupt.
- VSBY (voltage standby) state - check if the BBRAM is using standby voltage.
- VCC (active power) state - check if the BBRAM is on normal power.
- Size - get the size (in bytes) of the BBRAM region.

Along with these, the API provides a means for reading and writing to the memory region via *bbram_read()* and *bbram_write()* respectively. Both functions are expected to only succeed if the BBRAM is in a valid state and the operation is bounded to the memory region.

API Reference

group **bbram_interface**
BBRAM Interface.

Typedefs

typedef int (*`bbram_api_check_invalid_t`)(const struct *device* *dev)

API template to check if the BBRAM is invalid.

See also

bbram_check_invalid

typedef int (*`bbram_api_check_standby_power_t`)(const struct *device* *dev)

API template to check for standby power failure.

See also

bbram_check_standby_power

typedef int (*`bbram_api_check_power_t`)(const struct *device* *dev)

API template to check for V CC1 power failure.

See also

bbram_check_power

typedef int (*`bbram_api_get_size_t`)(const struct *device* *dev, size_t *size)

API template to check the size of the BBRAM.

See also

bbram_get_size

typedef int (*`bbram_api_read_t`)(const struct *device* *dev, size_t offset, size_t size, uint8_t *data)

API template to read from BBRAM.

See also

bbram_read

typedef int (*`bbram_api_write_t`)(const struct *device* *dev, size_t offset, size_t size, const uint8_t *data)

API template to write to BBRAM.

See also

bbram_write

Functions

int `bbram_check_invalid`(const struct *device* *dev)

Check if BBRAM is invalid.

Check if "Invalid Battery-Backed RAM" status is set then reset the status bit. This may occur as a result to low voltage at the VBAT pin.

Parameters

- `dev` – **[in]** BBRAM device pointer.

Returns

0 if the Battery-Backed RAM data is valid, -EFAULT otherwise.

int `bbram_check_standby_power`(const struct *device* *dev)

Check for standby (Volt SBY) power failure.

Check if the V standby power domain is turned on after it was off then reset the status bit.

Parameters

- `dev` – **[in]** BBRAM device pointer.

Returns

0 if V SBY power domain is in normal operation.

int `bbram_check_power`(const struct *device* *dev)

Check for V CC1 power failure.

This will return an error if the V CC1 power domain is turned on after it was off and reset the status bit.

Parameters

- `dev` – **[in]** BBRAM device pointer.

Returns

0 if the V CC1 power domain is in normal operation, -EFAULT otherwise.

int `bbram_get_size`(const struct *device* *dev, size_t *size)

Get the size of the BBRAM (in bytes).

Parameters

- `dev` – **[in]** BBRAM device pointer.
- `size` – **[out]** Pointer to write the size to.

Returns

0 for success, -EFAULT otherwise.

int **bbram_read**(const struct *device* *dev, size_t offset, size_t size, uint8_t *data)

Read bytes from BBRAM.

Parameters

- **dev** – **[in]** The BBRAM device pointer to read from.
- **offset** – **[in]** The offset into the RAM address to start reading from.
- **size** – **[in]** The number of bytes to read.
- **data** – **[out]** The buffer to load the data into.

Returns

0 on success, -EFAULT if the address range is out of bounds.

int **bbram_write**(const struct *device* *dev, size_t offset, size_t size, const uint8_t *data)

Write bytes to BBRAM.

Parameters

- **dev** – **[in]** The BBRAM device pointer to write to.
- **offset** – **[in]** The offset into the RAM address to start writing to.
- **size** – **[in]** The number of bytes to write.
- **data** – **[out]** Pointer to the start of data to write.

Returns

0 on success, -EFAULT if the address range is out of bounds.

int **bbram_emul_set_invalid**(const struct *device* *dev, bool is_invalid)

Set the emulated BBRAM driver's invalid state.

Calling this will affect the emulated behavior of *bbram_check_invalid()*.

Parameters

- **dev** – **[in]** The emulated device to modify
- **is_invalid** – **[in]** The new invalid state

Returns

0 on success, negative values on error.

int **bbram_emul_set_standby_power_state**(const struct *device* *dev, bool failure)

Set the emulated BBRAM driver's standby power state.

Calling this will affect the emulated behavior of *bbram_check_standby_power()*.

Parameters

- **dev** – **[in]** The emulated device to modify
- **failure** – **[in]** Whether or not standby power failure should be emulated

Returns

0 on success, negative values on error.

int **bbram_emul_set_power_state**(const struct *device* *dev, bool failure)

Set the emulated BBRAM driver's power state.

Calling this will affect the emulated behavior of *bbram_check_power()*.

Parameters

- **dev** – **[in]** The emulated device to modify
- **failure** – **[in]** Whether or not a power failure should be emulated

Returns
0 on success, negative values on error.

struct `bbram_driver_api`
#include <bbram.h>

7.6.6 BC1.2 Devices (Experimental)

The Battery Charging specification, currently at revision 1.2, is commonly referred to as just BC1.2. BC1.2 defines limits and detection mechanisms for USB devices to draw current exceeding the USB 2.0 specification limit of 0.5A, 2.5W.

The BC1.2 specification uses the term Charging Port for the device that supplies VBUS on the USB connection and the term Portable Device for the device that sinks current from the USB connection.

Note that the BC1.2 Specification uses the acronym PD for Portable Device. This should not be confused with the USB-C Power Delivery, which also uses the acronym PD.

On many devices, BC1.2 is the fallback mechanism to determine the connected charger capability on USB type C ports when the attached type-C partner does not support Power Delivery.

Key parameters from the BC1.2 Specification include:

Parameter	Symbol	Range
Allowed PD (portable device) Current Draw from Charging Port	IDEV_CHG	1.5 A
Charging Downstream Port Rated Current	ICDP	1.5 - 5.0 A
Maximum Configured Current when connected to a SDP	ICFG_MAX	500 mA
Dedicated Charging Port Rated Current	IDCP	1.5 - 5.0 A
Suspend current	ISUSP	2.5 mA
Unit load current	IUNIT	100 mA

While the ICDP and IDCP rated currents go up to 5.0 A, the BC1.2 current is limited by the IDEV_CHG parameter. So the BC1.2 support is capped at 1.5 A in the Zephyr implementation when using portable-device mode.

Basic Operation

The BC1.2 device driver provides only two APIs, `bc12_set_role()` and `bc12_set_result_cb()`.

The application calls `bc12_set_role()` to transition the BC1.2 device to either a disconnected, portable-device, or charging port mode.

For the disconnected state, the BC1.2 driver powers down the detection chip. The power down operation is vendor specific.

The application calls `bc12_set_role()` with the type set to BC12_PORTABLE_DEVICE when both the following conditions are true:

- The application configured the port as an upstream facing port, i.e. a USB device port.
- The application detects VBUS on the USB connection.

For portable-device mode, the BC1.2 driver powers up the detection chip and starts charger detection. At completion of the charger detection, the BC1.2 driver notifies the callback registered with `bc12_set_result_cb()`. By default, the BC1.2 driver clamps the current to 1.5A to comply with the BC1.2 specification.

To comply with the USB 2.0 specification, when the driver detects a SDP (Standard Downstream Port) charging partner or if BC1.2 detection fails, the driver reports the available current as ISUSP (2.5 mA). The application may increase the current draw to IUNIT (100 mA) when the connected USB host resumes the USB bus and may increase the current draw to ICFG_MAX (500 mA) when the USB host configures the USB device.

Charging port mode is used by the application when the USB port is configured as a downstream facing port, i.e. a USB host port. For charging port mode, the BC1.2 driver powers up the detection chip and configures the charger type specified by a devicetree property. If the driver supports detection of plug and unplug events, the BC1.2 driver notifies the callback registered with `bc12_set_result_cb()` to indicate the current connection state of the portable device partner.

Configuration Options

Related configuration options:

- `CONFIG_USB_BC12`

API Reference

group **b12_interface**

BC1.2 driver APIs.

BC1.2 constants

BC12_CHARGER_VOLTAGE_UV

BC1.2 USB charger voltage.

BC12_CHARGER_MIN_CURR_UA

BC1.2 USB charger minimum current.

Set to match the Isusp of 2.5 mA parameter. This is returned by the driver when either BC1.2 detection fails, or the attached partner is a SDP (standard downstream port).

The application may increase the current draw after determining the USB device state of suspended/unconfigured/configured. Suspended: 2.5 mA Unconfigured: 100 mA Configured: 500 mA (USB 2.0)

BC12_CHARGER_MAX_CURR_UA

BC1.2 USB charger maximum current.

Typedefs

typedef void (***bc12_callback_t**)(const struct *device* *dev, struct *bc12_partner_state* *state, void *user_data)

BC1.2 callback for charger configuration.

Param dev
BC1.2 device which is notifying of the new charger state.

Param state
Current state of the BC1.2 client, including BC1.2 type detected, voltage, and current limits. If NULL, then the partner charger is disconnected or the BC1.2 device is operating in host mode.

Param user_data
Requester supplied data which is passed along to the callback.

Enums

enum `bc12_role`

BC1.2 device role.

Values:

enumerator `BC12_DISCONNECTED`

enumerator `BC12_PORTABLE_DEVICE`

enumerator `BC12_CHARGING_PORT`

enum `bc12_type`

BC1.2 charging partner type.

Values:

enumerator `BC12_TYPE_NONE`
No partner connected.

enumerator `BC12_TYPE_SDP`
Standard Downstream Port.

enumerator `BC12_TYPE_DCP`
Dedicated Charging Port.

enumerator `BC12_TYPE_CDP`
Charging Downstream Port.

enumerator `BC12_TYPE_PROPRIETARY`
Proprietary charging port.

enumerator `BC12_TYPE_UNKNOWN`
Unknown charging port, BC1.2 detection failed.

enumerator `BC12_TYPE_COUNT`
Count of valid BC12 types.

Functions

int `bc12_set_role`(const struct *device* *dev, enum *bc12_role* role)

Set the BC1.2 role.

Parameters

- `dev` – Pointer to the device structure for the BC1.2 driver instance.
- `role` – New role for the BC1.2 device.

Return values

- `0` – If successful.
- `-EIO` – general input/output error.

int `bc12_set_result_cb`(const struct *device* *dev, *bc12_callback_t* cb, void *user_data)

Register a callback for BC1.2 results.

Parameters

- `dev` – Pointer to the device structure for the BC1.2 driver instance.
- `cb` – Function pointer for the result callback.
- `user_data` – Requester supplied data which is passed along to the callback.

Return values

- `0` – If successful.
- `-EIO` – general input/output error.

struct `bc12_partner_state`

#include <*usb_bc12.h*> BC1.2 detected partner state.

Param bc12_role
Current role of the BC1.2 device.

Param type
Charging partner type. Valid when bc12_role is BC12_PORTABLE_DEVICE.

Param current_ma
Current, in uA, that the charging partner provides. Valid when bc12_role is BC12_PORTABLE_DEVICE.

Param voltage_mv
Voltage, in uV, that the charging partner provides. Valid when bc12_role is BC12_PORTABLE_DEVICE.

Param pd_partner_connected
True if a PD partner is currently connected. Valid when bc12_role is BC12_CHARGING_PORT.

group `b12_emulator_backend`

BC1.2 backend emulator APIs.

Functions

static inline int `bc12_emul_set_charging_partner`(const struct *emul* *target, enum *bc12_type* partner_type)

Set the charging partner type connected to the BC1.2 device.

The corresponding BC1.2 emulator updates the vendor specific registers to simulate connection of the specified charging partner type. The emulator also generates an interrupt for processing by the real driver, if supported.

Parameters

- `target` – Pointer to the emulator structure for the BC1.2 emulator instance.
- `partner_type` – The simulated partner type. Set to BC12_TYPE_NONE to disconnect the charging partner.

Return values

- `0` – If successful.
- `-EINVAL` – if the partner type is not supported.

static inline int `bc12_emul_set_pd_partner`(const struct *emul* *target, bool connected)

Set the portable device partner state.

The corresponding BC1.2 emulator updates the vendor specific registers to simulate connection or disconnection of a portable device partner. The emulator also generates an interrupt for processing by the real driver, if supported.

Parameters

- `target` – Pointer to the emulator structure for the BC1.2 emulator instance.
- `connected` – If true, emulate a connection of a portable device partner. If false, emulate a disconnect event.

Return values

- `0` – If successful.
- `-EINVAL` – if the connection/disconnection of PD partner is not supported.

7.6.7 Clock Control

Overview

The clock control API provides access to clocks in the system, including the ability to turn them on and off.

Configuration Options

Related configuration options:

- `CONFIG_CLOCK_CONTROL`

API Reference

Related code samples

LiteX clock control driver
Use LiteX clock control driver to generate multiple clock signals.

group `clock_control_interface`

Clock Control Interface.

Since
1.0

Version
1.0.0

Defines

`CLOCK_CONTROL_SUBSYS_ALL`

Typedefs

typedef void ***clock_control_subsys_t**

clock_control_subsys_t is a type to identify a clock controller sub-system.

Such data pointed is opaque and relevant only to the clock controller driver instance being used.

typedef void ***clock_control_subsys_rate_t**

clock_control_subsys_rate_t is a type to identify a clock controller sub-system rate.

Such data pointed is opaque and relevant only to set the clock controller rate of the driver instance being used.

typedef void (***clock_control_cb_t**)(const struct *device* *dev, *clock_control_subsys_t* subsys, void *user_data)

Callback called on clock started.

Param dev
Device structure whose driver controls the clock.

Param subsys
Opaque data representing the clock.

Param user_data
User data.

typedef int (***clock_control**)(const struct *device* *dev, *clock_control_subsys_t* sys)

typedef int (***clock_control_get**)(const struct *device* *dev, *clock_control_subsys_t* sys, uint32_t *rate)

typedef int (***clock_control_async_on_fn**)(const struct *device* *dev, *clock_control_subsys_t* sys, *clock_control_cb_t* cb, void *user_data)

typedef enum *clock_control_status* (***clock_control_get_status_fn**)(const struct *device* *dev, *clock_control_subsys_t* sys)

typedef int (*clock_control_set)(const struct *device* *dev, *clock_control_subsys_t* sys, *clock_control_subsys_rate_t* rate)

typedef int (*clock_control_configure_fn)(const struct *device* *dev, *clock_control_subsys_t* sys, void *data)

Enums

enum clock_control_status

Current clock status.

Values:

enumerator CLOCK_CONTROL_STATUS_STARTING

enumerator CLOCK_CONTROL_STATUS_OFF

enumerator CLOCK_CONTROL_STATUS_ON

enumerator CLOCK_CONTROL_STATUS_UNKNOWN

Functions

static inline int clock_control_on(const struct *device* *dev, *clock_control_subsys_t* sys)

Enable a clock controlled by the device.

On success, the clock is enabled and ready when this function returns. This function may sleep, and thus can only be called from thread context.

Use *clock_control_async_on()* for non-blocking operation.

Parameters

- dev – Device structure whose driver controls the clock.
- sys – Opaque data representing the clock.

Returns

0 on success, negative errno on failure.

static inline int clock_control_off(const struct *device* *dev, *clock_control_subsys_t* sys)

Disable a clock controlled by the device.

This function is non-blocking and can be called from any context. On success, the clock is disabled when this function returns.

Parameters

- dev – Device structure whose driver controls the clock
- sys – Opaque data representing the clock

Returns

0 on success, negative errno on failure.

static inline int **clock_control_async_on**(const struct *device* *dev, *clock_control_subsys_t* sys, *clock_control_cb_t* cb, void *user_data)

Request clock to start with notification when clock has been started.

Function is non-blocking and can be called from any context. User callback is called when clock is started.

Parameters

- `dev` – Device.
- `sys` – A pointer to an opaque data representing the sub-system.
- `cb` – Callback.
- `user_data` – User context passed to the callback.

Return values

- `0` – if start is successfully initiated.
- `-EALREADY` – if clock was already started and is starting or running.
- `-ENOTSUP` – If the requested mode of operation is not supported.
- `-ENOSYS` – if the interface is not implemented.
- `other` – negative errno on vendor specific error.

static inline enum *clock_control_status* **clock_control_get_status**(const struct *device* *dev, *clock_control_subsys_t* sys)

Get clock status.

Parameters

- `dev` – Device.
- `sys` – A pointer to an opaque data representing the sub-system.

Returns
Status.

static inline int **clock_control_get_rate**(const struct *device* *dev, *clock_control_subsys_t* sys, uint32_t *rate)

Obtain the clock rate of given sub-system.

Parameters

- `dev` – Pointer to the device structure for the clock controller driver instance
- `sys` – A pointer to an opaque data representing the sub-system
- `rate` – **[out]** Subsystem clock rate

Return values

- `0` – on successful rate reading.
- `-EAGAIN` – if rate cannot be read. Some drivers do not support returning the rate when the clock is off.
- `-ENOTSUP` – if reading the clock rate is not supported for the given subsystem.
- `-ENOSYS` – if the interface is not implemented.

static inline int **clock_control_set_rate**(const struct *device* *dev, *clock_control_subsys_t* sys, *clock_control_subsys_rate_t* rate)

Set the rate of the clock controlled by the device.

On success, the new clock rate is set and ready when this function returns. This function may sleep, and thus can only be called from thread context.

Parameters

- `dev` – Device structure whose driver controls the clock.
- `sys` – Opaque data representing the clock.
- `rate` – Opaque data representing the clock rate to be used.

Return values

- `-EALREADY` – if clock was already in the given rate.
- `-ENOTSUP` – If the requested mode of operation is not supported.
- `-ENOSYS` – if the interface is not implemented.
- `other` – negative errno on vendor specific error.

static inline int **clock_control_configure**(const struct *device* *dev, *clock_control_subsys_t* sys, void *data)

Configure a source clock.

This function is non-blocking and can be called from any context. On success, the selected clock is configured as per caller's request.

It is caller's responsibility to ensure that subsequent calls to the API provide the right information to allows clock_control driver to perform the right action (such as using the right clock source on clock_control_get_rate call).

`data` is implementation specific and could be used to convey supplementary information required for expected clock configuration.

Parameters

- `dev` – Device structure whose driver controls the clock
- `sys` – Opaque data representing the clock
- `data` – Opaque data providing additional input for clock configuration

Return values

- `0` – On success
- `-ENOSYS` – If the device driver does not implement this call
- `-errno` – Other negative errno on failure.

struct **clock_control_driver_api**

#include <clock_control.h>

7.6.8 Controller Area Network (CAN)

CAN Controller

- *Overview*
- *Sending*
- *Receiving*
- *Setting the bitrate*
- *SocketCAN*
- *Samples*
- *CAN Controller API Reference*

Overview Controller Area Network is a two-wire serial bus specified by the Bosch CAN Specification, Bosch CAN with Flexible Data-Rate specification and the ISO 11898-1:2003 standard. CAN is mostly known for its application in the automotive domain. However, it is also used in home and industrial automation and other products.

Warning

CAN controllers can only initialize when the bus is in the idle (recessive) state for at least 11 recessive bits. Therefore you have to make sure that CAN RX is high, at least for a short time. This is also necessary for loopback mode.

The bit-timing as defined in ISO 11898-1:2003 looks as following:



Figure 2: CAN Timing. ©Alexander Wachter

A single bit is split into four segments.

- Sync_Seg: The nodes synchronize at the edge of the Sync_Seg. It is always one time quantum in length.
- Prop_Seg: The signal propagation delay of the bus and other delays of the transceiver and node.
- Phase_Seg1 and Phase_Seg2 :Define the sampling point. The bit is sampled at the end of Phase_Seg1.

The bit-rate is calculated from the time of a time quantum and the values defined above. A bit has the length of Sync_Seg plus Prop_Seg plus Phase_Seg1 plus Phase_Seg2 multiplied by the time of single time quantum. The bit-rate is the inverse of the length of a single bit.

A bit is sampled at the sampling point. The sample point is between Phase_Seg1 and PhaseSeg2 and therefore is a parameter that the user needs to choose. The CiA recommends setting the sample point to 87.5% of the bit.

The resynchronization jump width (SJW) defines the amount of time quantum the sample point can be moved. The sample point is moved when resynchronization is needed.

The timing parameters (SJW, bitrate and sampling point, or bitrate, Prop_Seg, Phase_Seg1and Phase_Seg2) are initially set from the device-tree and can be changed at run-time from the timing-API.

CAN uses so-called identifiers to identify the frame instead of addresses to identify a node. This identifier can either have 11-bit width (Standard or Basic Frame) or 29-bit in case of an Extended Frame. The Zephyr CAN API supports both Standard and Extended identifiers concurrently. A

CAN frame starts with a dominant Start Of Frame bit. After that, the identifiers follow. This phase is called the arbitration phase. During the arbitration phase, write collisions are allowed. They resolve by the fact that dominant bits override recessive bits. Nodes monitor the bus and notice when their transmission is being overridden and in case, abort their transmission. This effectively gives lower number identifiers priority over higher number identifiers.

Filters are used to whitelist identifiers that are of interest for the specific node. An identifier that doesn't match any filter is ignored. Filters can either match exactly or a specified part of the identifier. This method is called masking. As an example, a mask with 11 bits set for standard or 29 bits set for extended identifiers must match perfectly. Bits that are set to zero in the mask are ignored when matching an identifier. Most CAN controllers implement a limited number of filters in hardware. The number of filters is also limited in Kconfig to save memory.

Errors may occur during transmission. In case a node detects an erroneous frame, it partially overrides the current frame with an error-frame. Error-frames can either be error passive or error active, depending on the state of the controller. In case the controller is in error active state, it sends six consecutive dominant bits, which is a violation of the stuffing rule that all nodes can detect. The sender may resend the frame right after.

An initialized node can be in one of the following states:

- Error-active
- Error-passive
- Bus-off

After initialization, the node is in the error-active state. In this state, the node is allowed to send active error frames, ACK, and overload frames. Every node has a receive- and transmit-error counter. If either the receive- or the transmit-error counter exceeds 127, the node changes to error-passive state. In this state, the node is not allowed to send error-active frames anymore. If the transmit-error counter increases further to 255, the node changes to the bus-off state. In this state, the node is not allowed to send any dominant bits to the bus. Nodes in the bus-off state may recover after receiving 128 occurrences of 11 concurrent recessive bits.

You can read more about CAN bus in this CAN Wikipedia article.

Zephyr supports following CAN features:

- Standard and Extended Identifiers
- Filters with Masking
- Loopback and Silent mode
- Remote Request

Sending The following code snippets show how to send data.

This basic sample sends a CAN frame with standard identifier 0x123 and eight bytes of data. When passing NULL as the callback, as shown in this example, the send function blocks until the frame is sent and acknowledged by at least one other node or an error occurred. The timeout only takes effect on acquiring a mailbox. When a transmitting mailbox is assigned, sending cannot be canceled.

```
struct can_frame frame = {
        .flags = 0,
        .id = 0x123,
        .dlc = 8,
        .data = {1,2,3,4,5,6,7,8}
};
const struct device *const can_dev = DEVICE_DT_GET(DT_CHOSEN(zephyr_canbus));
int ret;
```

(continues on next page)

(continued from previous page)

```
ret = can_send(can_dev, &frame, K_MSEC(100), NULL, NULL);
if (ret != 0) {
        LOG_ERR("Sending failed [%d]", ret);
}
```

This example shows how to send a frame with extended identifier 0x1234567 and two bytes of data. The provided callback is called when the message is sent, or an error occurred. Passing *K_FOREVER* to the timeout causes the function to block until a transfer mailbox is assigned to the frame or an error occurred. It does not block until the message is sent like the example above.

```
void tx_callback(const struct device *dev, int error, void *user_data)
{
        char *sender = (char *)user_data;

        if (error != 0) {
                LOG_ERR("Sending failed [%d]\nSender: %s\n", error, sender);
        }
}

int send_function(const struct device *can_dev)
{
        struct can_frame frame = {
                .flags = CAN_FRAME_IDE,
                .id = 0x1234567,
                .dlc = 2
        };

        frame.data[0] = 1;
        frame.data[1] = 2;

        return can_send(can_dev, &frame, K_FOREVER, tx_callback, "Sender 1");
}
```

Receiving Frames are only received when they match a filter. The following code snippets show how to receive frames by adding filters.

Here we have an example for a receiving callback as used for *can_add_rx_filter()*. The user data argument is passed when the filter is added.

```
void rx_callback_function(const struct device *dev, struct can_frame *frame, void *user_
↪data)
{
        ... do something with the frame ...
}
```

The following snippet shows how to add a filter with a callback function. It is the most efficient but also the most critical way to receive messages. The callback function is called from an interrupt context, which means that the callback function should be as short as possible and must not block. Adding callback functions is not allowed from userspace context.

The filter for this example is configured to match the identifier 0x123 exactly.

```
const struct can_filter my_filter = {
        .flags = 0U,
        .id = 0x123,
        .mask = CAN_STD_ID_MASK
};
int filter_id;
const struct device *const can_dev = DEVICE_DT_GET(DT_CHOSEN(zephyr_canbus));
```

(continues on next page)

(continued from previous page)

```
filter_id = can_add_rx_filter(can_dev, rx_callback_function, callback_arg, &my_filter);
if (filter_id < 0) {
  LOG_ERR("Unable to add rx filter [%d]", filter_id);
}
```

Here an example for *can_add_rx_filter_msgq()* is shown. With this function, it is possible to receive frames synchronously. This function can be called from userspace context. The size of the message queue should be as big as the expected backlog.

The filter for this example is configured to match the extended identifier 0x1234567 exactly.

```
const struct can_filter my_filter = {
        .flags = CAN_FILTER_IDE,
        .id = 0x1234567,
        .mask = CAN_EXT_ID_MASK
};
CAN_MSGQ_DEFINE(my_can_msgq, 2);
struct can_frame rx_frame;
int filter_id;
const struct device *const can_dev = DEVICE_DT_GET(DT_CHOSEN(zephyr_canbus));

filter_id = can_add_rx_filter_msgq(can_dev, &my_can_msgq, &my_filter);
if (filter_id < 0) {
  LOG_ERR("Unable to add rx msgq [%d]", filter_id);
  return;
}

while (true) {
  k_msgq_get(&my_can_msgq, &rx_frame, K_FOREVER);
  ... do something with the frame ...
}
```

can_remove_rx_filter() removes the given filter.

```
can_remove_rx_filter(can_dev, filter_id);
```

Setting the bitrate The bitrate and sampling point is initially set at runtime. To change it from the application, one can use the *can_set_timing()* API. The *can_calc_timing()* function can calculate timing from a bitrate and sampling point in permille. The following example sets the bitrate to 250k baud with the sampling point at 87.5%.

```
struct can_timing timing;
const struct device *const can_dev = DEVICE_DT_GET(DT_CHOSEN(zephyr_canbus));
int ret;

ret = can_calc_timing(can_dev, &timing, 250000, 875);
if (ret > 0) {
  LOG_INF("Sample-Point error: %d", ret);
}

if (ret < 0) {
  LOG_ERR("Failed to calc a valid timing");
  return;
}

ret = can_stop(can_dev);
if (ret != 0) {
  LOG_ERR("Failed to stop CAN controller");
}
```

(continues on next page)

(continued from previous page)

```
ret = can_set_timing(can_dev, &timing);
if (ret != 0) {
  LOG_ERR("Failed to set timing");
}

ret = can_start(can_dev);
if (ret != 0) {
  LOG_ERR("Failed to start CAN controller");
}
```

A similar API exists for calculating and setting the timing for the data phase for CAN FD capable controllers. See *can_set_timing_data()* and *can_calc_timing_data()*.

SocketCAN Zephyr additionally supports SocketCAN, a BSD socket implementation of the Zephyr CAN API. SocketCAN brings the convenience of the well-known BSD Socket API to Controller Area Networks. It is compatible with the Linux SocketCAN implementation, where many other high-level CAN projects build on top. Note that frames are routed to the network stack instead of passed directly, which adds some computation and memory overhead.

Samples We have two ready-to-build samples demonstrating use of the Zephyr CAN API: Zephyr CAN counter sample and SocketCAN sample.

Related code samples

Controller Area Network (CAN) babbling node
Simulate a babbling CAN node.

Controller Area Network (CAN) counter
Send and receive CAN messages.

CAN Controller API Reference

group `can_interface`

CAN Interface.

Since
1.12

Version
1.1.0

CAN controller configuration

int `can_get_core_clock`(const struct *device* *dev, uint32_t *rate)

Get the CAN core clock rate.

Returns the CAN core clock rate. One minimum time quantum (mtq) is 1/(core clock rate). The CAN core clock can be further divided by the CAN clock prescaler (see the can_timing struct), providing the time quantum (tq).

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **rate** – **[out]** CAN core clock rate in Hz.

Returns
0 on success, or a negative error code on error

uint32_t **can_get_bitrate_min**(const struct *device* *dev)

Get minimum supported bitrate.

Get the minimum supported bitrate for the CAN controller/transceiver combination.

Parameters

- **dev** – Pointer to the device structure for the driver instance.

Returns
Minimum supported bitrate in bits/s

static inline int **can_get_min_bitrate**(const struct *device* *dev, uint32_t *min_bitrate)

Get minimum supported bitrate.

Get the minimum supported bitrate for the CAN controller/transceiver combination.

Deprecated:
Use can_get_bitrate_min() instead.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **min_bitrate** – **[out]** Minimum supported bitrate in bits/s

Return values

- **-EIO** – General input/output error.
- **-ENOSYS** – If this function is not implemented by the driver.

uint32_t **can_get_bitrate_max**(const struct *device* *dev)

Get maximum supported bitrate.

Get the maximum supported bitrate for the CAN controller/transceiver combination.

Parameters

- **dev** – Pointer to the device structure for the driver instance.

Returns
Maximum supported bitrate in bits/s

static inline int **can_get_max_bitrate**(const struct *device* *dev, uint32_t *max_bitrate)

Get maximum supported bitrate.

Get the maximum supported bitrate for the CAN controller/transceiver combination.

Deprecated:
Use can_get_bitrate_max() instead.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **max_bitrate** – **[out]** Maximum supported bitrate in bits/s

Return values

- `0` – If successful.
- `-EIO` – General input/output error.
- `-ENOSYS` – If this function is not implemented by the driver.

const struct *can_timing* *`can_get_timing_min`(const struct *device* *dev)

Get the minimum supported timing parameter values.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Returns

Pointer to the minimum supported timing parameter values.

const struct *can_timing* *`can_get_timing_max`(const struct *device* *dev)

Get the maximum supported timing parameter values.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Returns

Pointer to the maximum supported timing parameter values.

int `can_calc_timing`(const struct *device* *dev, struct *can_timing* *res, uint32_t bitrate, uint16_t sample_pnt)

Calculate timing parameters from bitrate and sample point.

Calculate the timing parameters from a given bitrate in bits/s and the sampling point in permill (1/1000) of the entire bit time. The bitrate must always match perfectly. If no result can be reached for the given parameters, -EINVAL is returned.

If the sample point is set to 0, this function defaults to a sample point of 75.0% for bitrates over 800 kbit/s, 80.0% for bitrates over 500 kbit/s, and 87.5% for all other bitrates.

Note

The requested `sample_pnt` will not always be matched perfectly. The algorithm calculates the best possible match.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `res` – **[out]** Result is written into the can_timing struct provided.
- `bitrate` – Target bitrate in bits/s.
- `sample_pnt` – Sample point in permille of the entire bit time or 0 for automatic sample point location.

Return values

- `0` – or positive sample point error on success.
- `-EINVAL` – if the requested bitrate or sample point is out of range.
- `-ENOTSUP` – if the requested bitrate is not supported.
- `-EIO` – if can_get_core_clock() is not available.

const struct *can_timing* *`can_get_timing_data_min`(const struct *device* *dev)

Get the minimum supported timing parameter values for the data phase.

Same as can_get_timing_min() but for the minimum values for the data phase.

> **Note**
>
> CONFIG_CAN_FD_MODE must be selected for this function to be available.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Returns

Pointer to the minimum supported timing parameter values, or NULL if CAN FD support is not implemented by the driver.

const struct *can_timing* *`can_get_timing_data_max`(const struct *device* *dev)

Get the maximum supported timing parameter values for the data phase.

Same as can_get_timing_max() but for the maximum values for the data phase.

> **Note**
>
> CONFIG_CAN_FD_MODE must be selected for this function to be available.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Returns

Pointer to the maximum supported timing parameter values, or NULL if CAN FD support is not implemented by the driver.

int `can_calc_timing_data`(const struct *device* *dev, struct *can_timing* *res, uint32_t bitrate, uint16_t sample_pnt)

Calculate timing parameters for the data phase.

Same as can_calc_timing() but with the maximum and minimum values from the data phase.

> **Note**
>
> CONFIG_CAN_FD_MODE must be selected for this function to be available.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `res` – **[out]** Result is written into the can_timing struct provided.
- `bitrate` – Target bitrate for the data phase in bits/s
- `sample_pnt` – Sample point for the data phase in permille of the entire bit time or 0 for automatic sample point location.

Return values

- `0` – or positive sample point error on success.
- `-EINVAL` – if the requested bitrate or sample point is out of range.

- `-ENOTSUP` – if the requested bitrate is not supported.
- `-EIO` – if can_get_core_clock() is not available.

int `can_set_timing_data`(const struct *device* *dev, const struct *can_timing* *timing_data)

Configure the bus timing for the data phase of a CAN FD controller.

See also

can_set_timing()

Note

`CONFIG_CAN_FD_MODE` must be selected for this function to be available.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `timing_data` – Bus timings for data phase

Return values

- `0` – If successful.
- `-EBUSY` – if the CAN controller is not in stopped state.
- `-EIO` – General input/output error, failed to configure device.
- `-ENOTSUP` – if the timing parameters are not supported by the driver.
- `-ENOSYS` – if CAN FD support is not implemented by the driver.

int `can_set_bitrate_data`(const struct *device* *dev, uint32_t bitrate_data)

Set the bitrate for the data phase of the CAN FD controller.

CAN in Automation (CiA) 301 v4.2.0 recommends a sample point location of 87.5% percent for all bitrates. However, some CAN controllers have difficulties meeting this for higher bitrates.

This function defaults to using a sample point of 75.0% for bitrates over 800 kbit/s, 80.0% for bitrates over 500 kbit/s, and 87.5% for all other bitrates. This is in line with the sample point locations used by the Linux kernel.

See also

can_set_bitrate()

Note

`CONFIG_CAN_FD_MODE` must be selected for this function to be available.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

- `bitrate_data` – Desired data phase bitrate.

Return values

- `0` – If successful.
- `-EBUSY` – if the CAN controller is not in stopped state.
- `-EINVAL` – if the requested bitrate is out of range.
- `-ENOTSUP` – if the requested bitrate not supported by the CAN controller/transceiver combination.
- `-ERANGE` – if the resulting sample point is off by more than +/- 5%.
- `-EIO` – General input/output error, failed to set bitrate.

int `can_calc_prescaler`(const struct *device* *dev, struct *can_timing* *timing, uint32_t bitrate)

Fill in the prescaler value for a given bitrate and timing.

Fill the prescaler value in the timing struct. The sjw, prop_seg, phase_seg1 and phase_seg2 must be given.

The returned bitrate error is remainder of the division of the clock rate by the bitrate times the timing segments.

Deprecated:

This function allows for bitrate errors, but bitrate errors between nodes on the same network leads to them drifting apart after the start-of-frame (SOF) synchronization has taken place.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `timing` – Result is written into the *can_timing* struct provided.
- `bitrate` – Target bitrate.

Return values

- `0` – or positive bitrate error.
- `Negative` – error code on error.

int `can_set_timing`(const struct *device* *dev, const struct *can_timing* *timing)

Configure the bus timing of a CAN controller.

See also

can_set_timing_data()

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `timing` – Bus timings.

Return values

- `0` – If successful.
- `-EBUSY` – if the CAN controller is not in stopped state.

- `-ENOTSUP` – if the timing parameters are not supported by the driver.
- `-EIO` – General input/output error, failed to configure device.

int **can_get_capabilities**(const struct *device* *dev, *can_mode_t* *cap)

Get the supported modes of the CAN controller.

The returned capabilities may not necessarily be supported at the same time (e.g. some CAN controllers support both `CAN_MODE_LOOPBACK` and `CAN_MODE_LISTENONLY`, but not at the same time).

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `cap` – **[out]** Supported capabilities.

Return values

- `0` – If successful.
- `-EIO` – General input/output error, failed to get capabilities.

const struct *device* ***can_get_transceiver**(const struct *device* *dev)

Get the CAN transceiver associated with the CAN controller.

Get a pointer to the device structure for the CAN transceiver associated with the CAN controller.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Returns

Pointer to the device structure for the associated CAN transceiver driver instance, or NULL if no transceiver is associated.

int **can_start**(const struct *device* *dev)

Start the CAN controller.

Bring the CAN controller out of `CAN_STATE_STOPPED`. This will reset the RX/TX error counters, enable the CAN controller to participate in CAN communication, and enable the CAN transceiver, if supported.

Starting the CAN controller resets all the CAN controller statistics.

See also

can_stop()

See also

can_transceiver_enable()

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Return values

- `0` – if successful.
- `-EALREADY` – if the device is already started.

- `-EIO` – General input/output error, failed to start device.

int `can_stop`(const struct *device* *dev)

Stop the CAN controller.

Bring the CAN controller into `CAN_STATE_STOPPED`. This will disallow the CAN controller from participating in CAN communication, abort any pending CAN frame transmissions, and disable the CAN transceiver, if supported.

See also

can_start()

See also

can_transceiver_disable()

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Return values

- `0` – if successful.
- `-EALREADY` – if the device is already stopped.
- `-EIO` – General input/output error, failed to stop device.

int `can_set_mode`(const struct *device* *dev, *can_mode_t* mode)

Set the CAN controller to the given operation mode.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `mode` – Operation mode.

Return values

- `0` – If successful.
- `-EBUSY` – if the CAN controller is not in stopped state.
- `-EIO` – General input/output error, failed to configure device.

can_mode_t `can_get_mode`(const struct *device* *dev)

Get the operation mode of the CAN controller.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Returns

Current operation mode.

int `can_set_bitrate`(const struct *device* *dev, uint32_t bitrate)

Set the bitrate of the CAN controller.

CAN in Automation (CiA) 301 v4.2.0 recommends a sample point location of 87.5% percent for all bitrates. However, some CAN controllers have difficulties meeting this for higher bitrates.

This function defaults to using a sample point of 75.0% for bitrates over 800 kbit/s, 80.0% for bitrates over 500 kbit/s, and 87.5% for all other bitrates. This is in line with the sample point locations used by the Linux kernel.

See also

can_set_bitrate_data()

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `bitrate` – Desired arbitration phase bitrate.

Return values

- `0` – If successful.
- `-EBUSY` – if the CAN controller is not in stopped state.
- `-EINVAL` – if the requested bitrate is out of range.
- `-ENOTSUP` – if the requested bitrate not supported by the CAN controller/transceiver combination.
- `-ERANGE` – if the resulting sample point is off by more than +/- 5%.
- `-EIO` – General input/output error, failed to set bitrate.

Transmitting CAN frames

int **can_send**(const struct *device* *dev, const struct *can_frame* *frame, *k_timeout_t* timeout, *can_tx_callback_t* callback, void *user_data)

Queue a CAN frame for transmission on the CAN bus.

Queue a CAN frame for transmission on the CAN bus with optional timeout and completion callback function.

Queued CAN frames are transmitted in order according to the their priority:

- The lower the CAN-ID, the higher the priority.
- Data frames have higher priority than Remote Transmission Request (RTR) frames with identical CAN-IDs.
- Frames with standard (11-bit) identifiers have higher priority than frames with extended (29-bit) identifiers with identical base IDs (the higher 11 bits of the extended identifier).
- Transmission order for queued frames with the same priority is hardware dependent.

By default, the CAN controller will automatically retry transmission in case of lost bus arbitration or missing acknowledge. Some CAN controllers support disabling automatic retransmissions via `CAN_MODE_ONE_SHOT`.

Note

If transmitting segmented messages spanning multiple CAN frames with identical CAN-IDs, the sender must ensure to only queue one frame at a time if FIFO order is required.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `frame` – CAN frame to transmit.
- `timeout` – Timeout waiting for a empty TX mailbox or `K_FOREVER`.
- `callback` – Optional callback for when the frame was sent or a transmission error occurred. If `NULL`, this function is blocking until frame is sent. The callback must be `NULL` if called from user mode.
- `user_data` – User data to pass to callback function.

Return values

- `0` – if successful.
- `-EINVAL` – if an invalid parameter was passed to the function.
- `-ENOTSUP` – if an unsupported parameter was passed to the function.
- `-ENETDOWN` – if the CAN controller is in stopped state.
- `-ENETUNREACH` – if the CAN controller is in bus-off state.
- `-EBUSY` – if CAN bus arbitration was lost (only applicable if automatic retransmissions are disabled).
- `-EIO` – if a general transmit error occurred (e.g. missing ACK if automatic retransmissions are disabled).
- `-EAGAIN` – on timeout.

Receiving CAN frames

int `can_add_rx_filter`(const struct *device* *dev, *can_rx_callback_t* callback, void *user_data, const struct *can_filter* *filter)

Add a callback function for a given CAN filter.

Add a callback to CAN identifiers specified by a filter. When a received CAN frame matching the filter is received by the CAN controller, the callback function is called in interrupt context.

If a received frame matches more than one filter (i.e., the filter IDs/masks or flags overlap), the priority of the match is hardware dependent.

The same callback function can be used for multiple filters.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `callback` – This function is called by the CAN controller driver whenever a frame matching the filter is received.
- `user_data` – User data to pass to callback function.
- `filter` – Pointer to a can_filter structure defining the filter.

Return values

- `filter_id` – on success.
- `-ENOSPC` – if there are no free filters.
- `-EINVAL` – if the requested filter type is invalid.
- `-ENOTSUP` – if the requested filter type is not supported.

int `can_add_rx_filter_msgq`(const struct *device* *dev, struct *k_msgq* *msgq, const struct *can_filter* *filter)

Simple wrapper function for adding a message queue for a given filter.

Wrapper function for can_add_rx_filter() which puts received CAN frames matching the filter in a message queue instead of calling a callback.

If a received frame matches more than one filter (i.e., the filter IDs/masks or flags overlap), the priority of the match is hardware dependent.

The same message queue can be used for multiple filters.

Note

The message queue must be initialized before calling this function and the caller must have appropriate permissions on it.

Warning

Message queue overruns are silently ignored and overrun frames discarded. Custom error handling can be implemented by using can_add_rx_filter() and k_msgq_put() directly.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `msgq` – Pointer to the already initialized k_msgq struct.
- `filter` – Pointer to a can_filter structure defining the filter.

Return values

- `filter_id` – on success.
- `-ENOSPC` – if there are no free filters.
- `-ENOTSUP` – if the requested filter type is not supported.

void `can_remove_rx_filter`(const struct *device* *dev, int filter_id)

Remove a CAN RX filter.

This routine removes a CAN RX filter based on the filter ID returned by can_add_rx_filter() or can_add_rx_filter_msgq().

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `filter_id` – Filter ID

int `can_get_max_filters`(const struct *device* *dev, bool ide)

Get maximum number of RX filters.

Get the maximum number of concurrent RX filters for the CAN controller.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `ide` – Get the maximum standard (11-bit) CAN ID filters if false, or extended (29-bit) CAN ID filters if true.

Return values

- `Positive` – number of maximum concurrent filters.
- `-EIO` – General input/output error.
- `-ENOSYS` – If this function is not implemented by the driver.

`CAN_MSGQ_DEFINE`(name, max_frames)

Statically define and initialize a CAN RX message queue.

The message queue's ring buffer contains space for *max_frames* CAN frames.

See also

can_add_rx_filter_msgq()

Parameters

- `name` – Name of the message queue.
- `max_frames` – Maximum number of CAN frames that can be queued.

CAN bus error reporting and handling

int `can_get_state`(const struct *device* *dev, enum *can_state* *state, struct *can_bus_err_cnt* *err_cnt)

Get current CAN controller state.

Returns the current state and optionally the error counter values of the CAN controller.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `state` – **[out]** Pointer to the state destination enum or NULL.
- `err_cnt` – **[out]** Pointer to the err_cnt destination structure or NULL.

Return values

- `0` – If successful.
- `-EIO` – General input/output error, failed to get state.

int `can_recover`(const struct *device* *dev, *k_timeout_t* timeout)

Recover from bus-off state.

Recover the CAN controller from bus-off state to error-active state.

Note

`CONFIG_CAN_MANUAL_RECOVERY_MODE` must be enabled for this function to be available.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **timeout** – Timeout for waiting for the recovery or K_FOREVER.

Return values

- **0** – on success.
- **-ENOTSUP** – if the CAN controller is not in manual recovery mode.
- **-ENETDOWN** – if the CAN controller is in stopped state.
- **-EAGAIN** – on timeout.
- **-ENOSYS** – If this function is not implemented by the driver.

static inline void **can_set_state_change_callback**(const struct *device* *dev, *can_state_change_callback_t* callback, void *user_data)

Set a callback for CAN controller state change events.

Set the callback for CAN controller state change events. The callback function will be called in interrupt context.

Only one callback can be registered per controller. Calling this function again overrides any previously registered callback.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **callback** – Callback function.
- **user_data** – User data to pass to callback function.

CAN statistics

uint32_t **can_stats_get_bit_errors**(const struct *device* *dev)

Get the bit error counter for a CAN device.

The bit error counter is incremented when the CAN controller is unable to transmit either a dominant or a recessive bit.

> **Note**
>
> CONFIG_CAN_STATS must be selected for this function to be available.

Parameters

- **dev** – Pointer to the device structure for the driver instance.

Returns

bit error counter

uint32_t **can_stats_get_bit0_errors**(const struct *device* *dev)

Get the bit0 error counter for a CAN device.

The bit0 error counter is incremented when the CAN controller is unable to transmit a dominant bit.

See also

can_stats_get_bit_errors()

Note

CONFIG_CAN_STATS must be selected for this function to be available.

Parameters

- **dev** – Pointer to the device structure for the driver instance.

Returns
bit0 error counter

uint32_t **can_stats_get_bit1_errors**(const struct *device* *dev)

Get the bit1 error counter for a CAN device.

The bit1 error counter is incremented when the CAN controller is unable to transmit a recessive bit.

See also

can_stats_get_bit_errors()

Note

CONFIG_CAN_STATS must be selected for this function to be available.

Parameters

- **dev** – Pointer to the device structure for the driver instance.

Returns
bit1 error counter

uint32_t **can_stats_get_stuff_errors**(const struct *device* *dev)

Get the stuffing error counter for a CAN device.

The stuffing error counter is incremented when the CAN controller detects a bit stuffing error.

Note

CONFIG_CAN_STATS must be selected for this function to be available.

Parameters

- **dev** – Pointer to the device structure for the driver instance.

Returns
stuffing error counter

uint32_t **can_stats_get_crc_errors**(const struct *device* *dev)

Get the CRC error counter for a CAN device.

The CRC error counter is incremented when the CAN controller detects a frame with an invalid CRC.

> **Note**
>
> CONFIG_CAN_STATS must be selected for this function to be available.

Parameters

- **dev** – Pointer to the device structure for the driver instance.

Returns

CRC error counter

uint32_t **can_stats_get_form_errors**(const struct *device* *dev)

Get the form error counter for a CAN device.

The form error counter is incremented when the CAN controller detects a fixed-form bit field containing illegal bits.

> **Note**
>
> CONFIG_CAN_STATS must be selected for this function to be available.

Parameters

- **dev** – Pointer to the device structure for the driver instance.

Returns

form error counter

uint32_t **can_stats_get_ack_errors**(const struct *device* *dev)

Get the acknowledge error counter for a CAN device.

The acknowledge error counter is incremented when the CAN controller does not monitor a dominant bit in the ACK slot.

> **Note**
>
> CONFIG_CAN_STATS must be selected for this function to be available.

Parameters

- **dev** – Pointer to the device structure for the driver instance.

Returns

acknowledge error counter

uint32_t **can_stats_get_rx_overruns**(const struct *device* *dev)

Get the RX overrun counter for a CAN device.

The RX overrun counter is incremented when the CAN controller receives a CAN frame matching an installed filter but lacks the capacity to store it (either due to an already full RX mailbox or a full RX FIFO).

> **Note**
>
> CONFIG_CAN_STATS must be selected for this function to be available.

Parameters

- **dev** – Pointer to the device structure for the driver instance.

Returns
RX overrun counter

CAN utility functions

static inline uint8_t **can_dlc_to_bytes**(uint8_t dlc)

Convert from Data Length Code (DLC) to the number of data bytes.

Parameters

- **dlc** – Data Length Code (DLC).

Return values
Number – of bytes.

static inline uint8_t **can_bytes_to_dlc**(uint8_t num_bytes)

Convert from number of bytes to Data Length Code (DLC)

Parameters

- **num_bytes** – Number of bytes.

Return values
Data – Length Code (DLC).

static inline bool **can_frame_matches_filter**(const struct *can_frame* *frame, const struct *can_filter* *filter)

Check if a CAN frame matches a CAN filter.

Parameters

- **frame** – CAN frame.
- **filter** – CAN filter.

Returns
true if the CAN frame matches the CAN filter, false otherwise

CAN frame definitions

CAN_STD_ID_MASK

Bit mask for a standard (11-bit) CAN identifier.

CAN_MAX_STD_ID

Maximum value for a standard (11-bit) CAN identifier.

Deprecated:
Use CAN_STD_ID_MASK instead.

CAN_EXT_ID_MASK
: Bit mask for an extended (29-bit) CAN identifier.

CAN_MAX_EXT_ID
: Maximum value for an extended (29-bit) CAN identifier.

  Deprecated:
  Use CAN_EXT_ID_MASK instead.

CAN_MAX_DLC
: Maximum data length code for CAN 2.0A/2.0B.

CANFD_MAX_DLC
: Maximum data length code for CAN FD.

CAN controller mode flags

CAN_MODE_NORMAL
: Normal mode.

CAN_MODE_LOOPBACK
: Controller is in loopback mode (receives own frames).

CAN_MODE_LISTENONLY
: Controller is not allowed to send dominant bits.

CAN_MODE_FD
: Controller allows transmitting/receiving CAN FD frames.

CAN_MODE_ONE_SHOT
: Controller does not retransmit in case of lost arbitration or missing ACK.

CAN_MODE_3_SAMPLES
: Controller uses triple sampling mode.

CAN_MODE_MANUAL_RECOVERY
: Controller requires manual recovery after entering bus-off state.

CAN frame flags

CAN_FRAME_IDE
: Frame uses extended (29-bit) CAN ID.

`CAN_FRAME_RTR`
: Frame is a Remote Transmission Request (RTR)

`CAN_FRAME_FDF`
: Frame uses CAN FD format (FDF)

`CAN_FRAME_BRS`
: Frame uses CAN FD Baud Rate Switch (BRS).

  Only valid in combination with `CAN_FRAME_FDF`.

`CAN_FRAME_ESI`
: CAN FD Error State Indicator (ESI).

  Indicates that the transmitting node is in error-passive state. Only valid in combination with `CAN_FRAME_FDF`.

CAN filter flags

`CAN_FILTER_IDE`
: Filter matches frames with extended (29-bit) CAN IDs.

Defines

`CAN_STATS_BIT_ERROR_INC(dev_)`
: Increment the bit error counter for a CAN device.

  The bit error counter is incremented when the CAN controller is unable to transmit either a dominant or a recessive bit.

See also

CAN_STATS_BIT0_ERROR_INC()

See also

CAN_STATS_BIT1_ERROR_INC()

Note

This error counter should only be incremented if the CAN controller is unable to distinguish between failure to transmit a dominant versus failure to transmit a recessive bit. If the CAN controller supports distinguishing between the two, the `bit0` or `bit1` error counter shall be incremented instead.

Parameters

- dev_ – Pointer to the device structure for the driver instance.

CAN_STATS_BIT0_ERROR_INC(dev_)

Increment the bit0 error counter for a CAN device.

The bit0 error counter is incremented when the CAN controller is unable to transmit a dominant bit.

Incrementing this counter will automatically increment the bit error counter.

See also

CAN_STATS_BIT_ERROR_INC()

Parameters

- dev_ – Pointer to the device structure for the driver instance.

CAN_STATS_BIT1_ERROR_INC(dev_)

Increment the bit1 (recessive) error counter for a CAN device.

The bit1 error counter is incremented when the CAN controller is unable to transmit a recessive bit.

Incrementing this counter will automatically increment the bit error counter.

See also

CAN_STATS_BIT_ERROR_INC()

Parameters

- dev_ – Pointer to the device structure for the driver instance.

CAN_STATS_STUFF_ERROR_INC(dev_)

Increment the stuffing error counter for a CAN device.

The stuffing error counter is incremented when the CAN controller detects a bit stuffing error.

Parameters

- dev_ – Pointer to the device structure for the driver instance.

CAN_STATS_CRC_ERROR_INC(dev_)

Increment the CRC error counter for a CAN device.

The CRC error counter is incremented when the CAN controller detects a frame with an invalid CRC.

Parameters

- dev_ – Pointer to the device structure for the driver instance.

CAN_STATS_FORM_ERROR_INC(dev_)

Increment the form error counter for a CAN device.

The form error counter is incremented when the CAN controller detects a fixed-form bit field containing illegal bits.

Parameters

- dev_ – Pointer to the device structure for the driver instance.

CAN_STATS_ACK_ERROR_INC(dev_)

Increment the acknowledge error counter for a CAN device.

The acknowledge error counter is incremented when the CAN controller does not monitor a dominant bit in the ACK slot.

Parameters

- dev_ – Pointer to the device structure for the driver instance.

CAN_STATS_RX_OVERRUN_INC(dev_)

Increment the RX overrun counter for a CAN device.

The RX overrun counter is incremented when the CAN controller receives a CAN frame matching an installed filter but lacks the capacity to store it (either due to an already full RX mailbox or a full RX FIFO).

Parameters

- dev_ – Pointer to the device structure for the driver instance.

CAN_STATS_RESET(dev_)

Zero all statistics for a CAN device.

The driver is responsible for resetting the statistics before starting the CAN controller.

Parameters

- dev_ – Pointer to the device structure for the driver instance.

CAN_DEVICE_DT_DEFINE(node_id, init_fn, pm, data, config, level, prio, api, ...)

Like *DEVICE_DT_DEFINE()* with CAN device specifics.

Defines a device which implements the CAN API. May generate a custom *device_state* container struct and init_fn wrapper when needed depending on CONFIG_CAN_STATS.

Parameters

- node_id – The devicetree node identifier.
- init_fn – Name of the init function of the driver.
- pm – PM device resources reference (NULL if device does not use PM).
- data – Pointer to the device's private data.
- config – The address to the structure containing the configuration information for this instance of the driver.
- level – The initialization level. See SYS_INIT() for details.
- prio – Priority within the selected initialization level. See SYS_INIT() for details.
- api – Provides an initial pointer to the API function struct used by the driver. Can be NULL.

CAN_DEVICE_DT_INST_DEFINE(inst, ...)

Like *CAN_DEVICE_DT_DEFINE()* for an instance of a DT_DRV_COMPAT compatible.

Parameters

- inst – Instance number. This is replaced by DT_DRV_COMPAT(inst) in the call to *CAN_DEVICE_DT_DEFINE()*.
- ... – Other parameters as expected by *CAN_DEVICE_DT_DEFINE()*.

Typedefs

typedef uint32_t **can_mode_t**

Provides a type to hold CAN controller configuration flags.

The lower 24 bits are reserved for common CAN controller mode flags. The upper 8 bits are reserved for CAN controller/driver specific flags.

See also

CAN_MODE_FLAGS.

typedef void (***can_tx_callback_t**)(const struct *device* *dev, int error, void *user_data)

Defines the application callback handler function signature.

Param dev
Pointer to the device structure for the driver instance.

Param error
Status of the performed send operation. See the list of return values for can_send() for value descriptions.

Param user_data
User data provided when the frame was sent.

typedef void (***can_rx_callback_t**)(const struct *device* *dev, struct *can_frame* *frame, void *user_data)

Defines the application callback handler function signature for receiving.

Param dev
Pointer to the device structure for the driver instance.

Param frame
Received frame.

Param user_data
User data provided when the filter was added.

typedef void (***can_state_change_callback_t**)(const struct *device* *dev, enum *can_state* state, struct *can_bus_err_cnt* err_cnt, void *user_data)

Defines the state change callback handler function signature.

Param dev
Pointer to the device structure for the driver instance.

Param state
State of the CAN controller.

Param err_cnt
CAN controller error counter values.

Param user_data
User data provided the callback was set.

Enums

enum **can_state**

Defines the state of the CAN controller.

Values:

enumerator **CAN_STATE_ERROR_ACTIVE**

Error-active state (RX/TX error count < 96).

enumerator **CAN_STATE_ERROR_WARNING**

Error-warning state (RX/TX error count < 128).

enumerator **CAN_STATE_ERROR_PASSIVE**

Error-passive state (RX/TX error count < 256).

enumerator **CAN_STATE_BUS_OFF**

Bus-off state (RX/TX error count >= 256).

enumerator **CAN_STATE_STOPPED**

CAN controller is stopped and does not participate in CAN communication.

struct **can_frame**

#include <*can.h*> CAN frame structure.

Public Members

uint32_t **id**

Standard (11-bit) or extended (29-bit) CAN identifier.

uint8_t **dlc**

Data Length Code (DLC) indicating data length in bytes.

uint8_t **flags**

Flags.

See also

CAN_FRAME_FLAGS.

uint16_t **timestamp**

Captured value of the free-running timer in the CAN controller when this frame was received.

The timer is incremented every bit time and captured at the start of frame bit (SOF).

Note

CONFIG_CAN_RX_TIMESTAMP must be selected for this field to be available.

uint8_t **data**[CAN_MAX_DLEN]

Payload data accessed as unsigned 8 bit values.

uint32_t **data_32**[*DIV_ROUND_UP*(CAN_MAX_DLEN, sizeof(uint32_t))]

Payload data accessed as unsigned 32 bit values.

union can_frame

The frame payload data.

struct **can_filter**

#include <*can.h*> CAN filter structure.

Public Members

uint32_t **id**

CAN identifier to match.

uint32_t **mask**

CAN identifier matching mask.

If a bit in this mask is 0, the value of the corresponding bit in the `id` field is ignored by the filter.

uint8_t **flags**

Flags.

See also

CAN_FILTER_FLAGS.

struct **can_bus_err_cnt**

#include <*can.h*> CAN controller error counters.

Public Members

uint8_t **tx_err_cnt**

Value of the CAN controller transmit error counter.

uint8_t **rx_err_cnt**

Value of the CAN controller receive error counter.

struct **can_timing**

#include <*can.h*> CAN bus timing structure.

This struct is used to pass bus timing values to the configuration and bitrate calculation functions.

The propagation segment represents the time of the signal propagation. Phase segment 1 and phase segment 2 define the sampling point. The prop_seg and phase_seg1 values affect the sampling point in the same way and some controllers only have a register for the sum of those two. The sync segment always has a length of 1 time quantum (see below).

```
+---------+----------+------------+------------+
|sync_seg | prop_seg | phase_seg1 | phase_seg2 |
+---------+----------+------------+------------+
                                 ^
                            Sampling-Point
```

1 time quantum (tq) has the length of 1/(core_clock / prescaler). The bitrate is defined by the core clock divided by the prescaler and the sum of the segments:

br = (core_clock / prescaler) / (1 + prop_seg + phase_seg1 + phase_seg2)

The Synchronization Jump Width (SJW) defines the amount of time quanta the sample point can be moved. The sample point is moved when resynchronization is needed.

Public Members

uint16_t **sjw**

Synchronisation jump width.

uint16_t **prop_seg**

Propagation segment.

uint16_t **phase_seg1**

Phase segment 1.

uint16_t **phase_seg2**

Phase segment 2.

uint16_t **prescaler**

Prescaler value.

struct **can_device_state**

#include <can.h> CAN specific device state which allows for CAN device class specific additions.

Public Members

struct *device_state* **devstate**

Common device state.

struct stats_can **stats**

CAN device statistics.

CAN Transceiver

- *Overview*
- *CAN Transceiver API Reference*

Overview A CAN transceiver is an external device that converts the logic level signals from the CAN controller to the bus-levels. The bus lines are called CAN High (CAN H) and CAN Low (CAN L). The transmit wire from the controller to the transceiver is called CAN TX, and the receive wire is called CAN RX. These wires use the logic levels whereas the bus-level is interpreted differentially between CAN H and CAN L. The bus can be either in the recessive (logical one) or dominant (logical zero) state. The recessive state is when both lines, CAN H and CAN L, are roughly at the same voltage level. This state is also the idle state. To write a dominant bit to the bus, open-drain transistors tie CAN H to Vdd and CAN L to ground. The first and last node use a 120-ohm resistor between CAN H and CAN L to terminate the bus. The dominant state always overrides the recessive state. This structure is called a wired-AND.

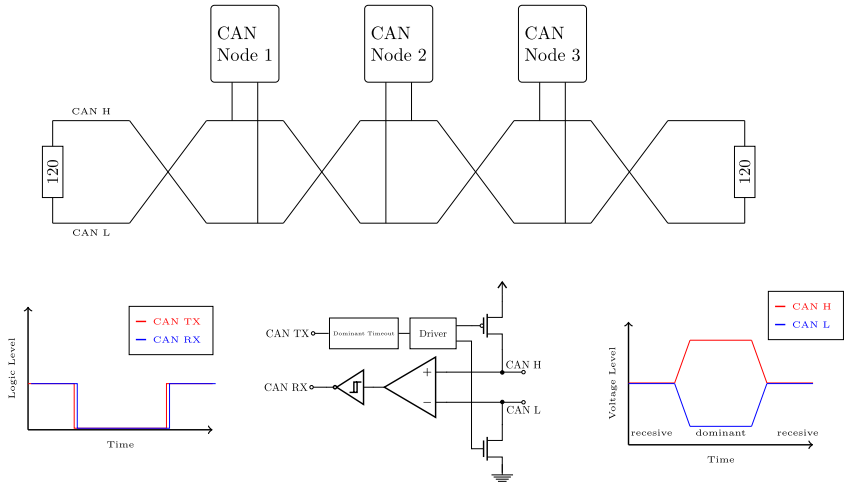


Figure 1: CAN Transceiver, from Logic Levels to Bus Levels. ©Alexander Wachter

CAN Transceiver API Reference

group `can_transceiver`

CAN Transceiver Driver APIs.

Since
3.1

Version
0.1.0

Functions

static inline int **`can_transceiver_enable`**(const struct *device* *dev, *can_mode_t* mode)

Enable CAN transceiver.

Enable the CAN transceiver.

See also

can_start()

Note

The CAN transceiver is controlled by the CAN controller driver and should not normally be controlled by the application.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `mode` – Operation mode.

Return values

- `0` – If successful.
- `-EIO` – General input/output error, failed to enable device.

static inline int `can_transceiver_disable`(const struct *device* *dev)

Disable CAN transceiver.

Disable the CAN transceiver.

See also

can_stop()

Note

The CAN transceiver is controlled by the CAN controller driver and should not normally be controlled by the application.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Return values

- `0` – If successful.
- `-EIO` – General input/output error, failed to disable device.

CAN Shell

- *Overview*
- *Inspection*
- *Configuration*

- *Receiving*
- *Sending*
- *Bus Recovery*

Overview The CAN shell provides a `can` command with a set of subcommands for the *shell* module. It allows for testing and exploring the *CAN Controller* driver API through an interactive interface without having to write a dedicated application. The CAN shell can also be enabled in existing applications to aid in interactive debugging of CAN issues.

The CAN shell provides access to most CAN controller features, including inspection, configuration, sending and receiving of CAN frames, and bus recovery.

In order to enable the CAN shell, the following *Kconfig* options must be enabled:

- `CONFIG_SHELL`
- `CONFIG_CAN`
- `CONFIG_CAN_SHELL`

The following *Kconfig* options enable additional subcommands and features of the `can` command:

- `CONFIG_CAN_FD_MODE` enables CAN FD specific subcommands (e.g. for setting the timing for the CAN FD data phase).
- `CONFIG_CAN_RX_TIMESTAMP` enables printing of timestamps for received CAN frames.
- `CONFIG_CAN_STATS` enables printing of various statistics for the CAN controller in the `can show` subcommand. This depends on `CONFIG_STATS` being enabled as well.
- `CONFIG_CAN_MANUAL_RECOVERY_MODE` enables the `can recover` subcommand.

For example, building the hello_world sample for the frdm_k64f with the CAN shell and CAN statistics enabled:

```
# From the root of the zephyr repository
west build -b frdm_k64f samples/hello_world -- -DCONFIG_SHELL=y -DCONFIG_CAN=y -DCONFIG_CAN_
↪SHELL=y -DCONFIG_STATS=y -DCONFIG_CAN_STATS=y
```

See the *shell* documentation for general instructions on how to connect and interact with the shell. The CAN shell comes with built-in help (unless `CONFIG_SHELL_HELP` is disabled). The built-in help messages can be printed by passing `-h` or `--help` to the `can` command or any of its subcommands. All subcommands also support tab-completion of their arguments.

Tip

All of the CAN shell subcommands take the name of a CAN controller as their first argument, which also supports tab-completion. A list of all devices available can be obtained using the `device list` shell command when `CONFIG_DEVICE_SHELL` is enabled. The examples below all use the device name `can@0`.

Inspection The properties of a given CAN controller can be inspected using the `can show` subcommand as shown below. The properties include the core CAN clock rate, the maximum supported bitrate, the number of RX filters supported, capabilities, current mode, current state, error counters, timing limits, and more:

```
uart:~$ can show can@0
core clock:      144000000 Hz
max bitrate:     5000000 bps
max std filters: 15
max ext filters: 15
capabilities:    normal loopback listen-only fd
mode:            normal
state:           stopped
rx errors:       0
tx errors:       0
timing:          sjw 1..128, prop_seg 0..0, phase_seg1 2..256, phase_seg2 2..128, prescaler␣
↪1..512
timing data:     sjw 1..16, prop_seg 0..0, phase_seg1 1..32, phase_seg2 1..16, prescaler 1..
↪32
transceiver:     passive/none
statistics:
  bit errors:    0
    bit0 errors: 0
    bit1 errors: 0
  stuff errors:  0
  crc errors:    0
  form errors:   0
  ack errors:    0
  rx overruns:   0
```

Note

The statistics are only printed if `CONFIG_CAN_STATS` is enabled.

Configuration The CAN shell allows for configuring the CAN controller mode and timing, along with starting and stopping the processing of CAN frames.

Note

The CAN controller mode and timing can only be changed while the CAN controller is stopped, which is the initial setting upon boot-up. The initial CAN controller mode is set to `normal` and the initial timing is set according to the `bitrate`, `sample-point`, `bitrate-data`, and `sample-point-data` *Devicetree* properties.

Timing The classic CAN bitrate/CAN FD arbitration phase bitrate can be configured using the `can bitrate` subcommand as shown below. The bitrate is specified in bits per second.

```
uart:~$ can bitrate can@0 125000
setting bitrate to 125000 bps
```

If `CONFIG_CAN_FD_MODE` is enabled, the data phase bitrate can be configured using the `can dbitrate` subcommand as shown below. The bitrate is specified in bits per second.

```
uart:~$ can dbitrate can@0 1000000
setting data bitrate to 1000000 bps
```

Both of these subcommands allow specifying an optional sample point in per mille and a (Re)Synchronization Jump Width (SJW) in Time Quanta as positional arguments. Refer to the interactive help of the subcommands for more details.

It is also possible to configure the raw bit timing using the `can timing` and `can dtiming` subcommands. Refer to the interactive help output for these subcommands for details on the required arguments.

Mode The CAN shell allows for setting the mode of the CAN controller using the `can mode` subcommand. An example for enabling loopback mode is shown below.

```
uart:~$ can mode can@0 loopback
setting mode 0x00000001
```

The subcommand accepts multiple modes given on the same command line (e.g. `can mode can@0 fd loopback` for setting CAN FD and loopback mode). Vendor-specific modes can be specified in hexadecimal.

Starting and Stopping After the timing and mode has been configured as needed, the CAN controller can be started using the `can start` subcommand as shown below. This will enable reception and transmission of CAN frames.

```
uart:~$ can start can@0
starting can@0
```

Prior to reconfiguring the timing or mode, the CAN controller needs to be stopped using the `can stop` subcommand as shown below:

```
uart:~$ can stop can@0
stopping can@0
```

Receiving In order to receive CAN frames, one or more CAN RX filters need to be configured. CAN RX filters are added using the `can filter add` subcommand as shown below. The subcommand accepts a CAN ID in hexadecimal format along with an optional CAN ID mask, also in hexadecimal format, for setting which bits in the CAN ID are to be matched. Refer to the interactive help output for this subcommand for further details on the supported arguments.

```
uart:~$ can filter add can@0 010
adding filter with standard (11-bit) CAN ID 0x010, CAN ID mask 0x7ff, data frames 1, RTR␣
↪frames 0, CAN FD frames 0
filter ID: 0
```

The filter ID (0 in the example above) returned is to be used when removing the CAN RX filter.

Received CAN frames matching the added filter(s) are printed to the shell. A few examples are shown below:

```
# Dev Flags    ID   Size  Data bytes
can0  --       010   [8]  01 02 03 04 05 06 07 08
can0  B-       010  [08]  01 02 03 04 05 06 07 08
can0  BP       010  [03]  01 aa bb
can0  --  00000010   [0]
can0  --       010   [1]  20
can0  --       010   [8]  remote transmission request
```

The columns have the following meaning:

- Dev
  - Name of the device receiving the frame.
- Flags
  - B: The frame has the CAN FD Baud Rate Switch (BRS) flag set.

  - P: The frame has the CAN FD Error State Indicator (ESI) flag set. The transmitting node is in error-passive state.
  - -: Unset flag.
- ID
  - 010: The standard (11-bit) CAN ID of the frame in hexadecimal format, here 10h.
  - 00000010: The extended (29-bit) CAN ID of the frame in hexadecimal format, here 10h.
- Size
  - [8]: The number of frame data bytes in decimal format, here a classic CAN frame with 8 data bytes.
  - [08]: The number of frame data bytes in decimal format, here a CAN FD frame with 8 data bytes.
- Data bytes
  - 01 02 03 04 05 06 07 08: The frame data bytes in hexadecimal format, here the numbers from 1 through 8.
  - remote transmission request: The frame is a Remote Transmission Request (RTR) frame and thus carries no data bytes.

Tip

If CONFIG_CAN_RX_TIMESTAMP is enabled, each line will be prepended with a timestamp from the free-running timestamp counter in the CAN controller.

Configured CAN RX filters can be removed again using the can filter remove subcommand as shown below. The filter ID is the ID returned by the can filter add subcommand (0 in the example below).

```
uart:~$ can filter remove can@0 0
removing filter with ID 0
```

Sending CAN frames can be queued for transmission using the can send subcommand as shown below. The subcommand accepts a CAN ID in hexadecimal format and optionally a number of data bytes, also specified in hexadecimal. Refer to the interactive help output for this subcommand for further details on the supported arguments.

```
uart:~$ can send can@0 010 1 2 3 4 5 6 7 8
enqueuing CAN frame #2 with standard (11-bit) CAN ID 0x010, RTR 0, CAN FD 0, BRS 0, DLC 8
CAN frame #2 successfully sent
```

Bus Recovery The can recover subcommand can be used for initiating manual recovery from a CAN bus-off event as shown below:

```
uart:~$ can recover can@0
recovering, no timeout
```

The subcommand accepts an optional bus recovery timeout in milliseconds. If no timeout is specified, the command will wait indefinitely for the bus recovery to succeed.

Note

The recover subcommand is only available if CONFIG_CAN_MANUAL_RECOVERY_MODE is enabled.

7.6.9 Chargers

The charger subsystem exposes an API to uniformly access battery charger devices.

A charger device, or charger peripheral, is a device used to take external power provided to the system as an input and provide power as an output downstream to the battery pack(s) and system. The charger device can exist as a module, an integrated circuit, or as a functional block in a power management integrated circuit (PMIC).

The action of charging a battery pack is referred to as a charge cycle. When the charge cycle is executed the battery pack is charged according to the charge profile configured on the charger device. The charge profile is defined in the battery pack's specification that is provided by the manufacturer. On charger devices with a control port, the charge profile can be configured by the host controller by setting the relevant properties, and can be adjusted at runtime to respond to environmental changes.

Basic Operation

Initiating a Charge Cycle A charge cycle is initiated or terminated using *charger_charge_enable()*.

Properties Fundamentally, a property is a configurable setting, state, or quantity that a charger device can measure.

Chargers typically support multiple properties, such as temperature readings of the battery-pack or present-time current/voltage.

Properties are fetched by the client one at a time using *charger_get_prop()*. Properties are set by the client one at a time using *charger_set_prop()*.

API Reference

Related code samples

Charger
Charge a battery using the charger driver API.

group `charger_interface`

Charger Interface.

Typedefs

typedef uint16_t `charger_prop_t`

A charger property's identifier.

See charger_property for a list of identifiers

typedef void (*`charger_status_notifier_t`)(enum *charger_status* status)

The charger status change callback to notify the system.

Param status
Current charging state

typedef void (*`charger_online_notifier_t`)(enum *charger_online* online)

The charger online change callback to notify the system.

Param online
Current external supply state

typedef int (*`charger_get_property_t`)(const struct *device* *dev, const *charger_prop_t* prop, union *charger_propval* *val)

Callback API for getting a charger property.

See charger_get_property() for argument description

typedef int (*`charger_set_property_t`)(const struct *device* *dev, const *charger_prop_t* prop, const union *charger_propval* *val)

Callback API for setting a charger property.

See charger_set_property() for argument description

typedef int (*`charger_charge_enable_t`)(const struct *device* *dev, const bool enable)

Callback API enabling or disabling a charge cycle.

See *charger_charge_enable()* for argument description

Enums

enum `charger_property`

Runtime Dynamic Battery Parameters.

Values:

enumerator `CHARGER_PROP_ONLINE` = 0

Indicates if external supply is present for the charger.

Value should be of type enum charger_online

enumerator `CHARGER_PROP_PRESENT`

Reports whether or not a battery is present.

Value should be of type bool

enumerator `CHARGER_PROP_STATUS`

Represents the charging status of the charger.

Value should be of type enum charger_status

enumerator `CHARGER_PROP_CHARGE_TYPE`

Represents the charging algo type of the charger.

Value should be of type enum charger_charge_type

enumerator `CHARGER_PROP_HEALTH`

Represents the health of the charger.

Value should be of type enum charger_health

enumerator CHARGER_PROP_CONSTANT_CHARGE_CURRENT_UA

Configuration of current sink used for charging in µA.

enumerator CHARGER_PROP_PRECHARGE_CURRENT_UA

Configuration of current sink used for conditioning in µA.

enumerator CHARGER_PROP_CHARGE_TERM_CURRENT_UA

Configuration of charge termination target in µA.

enumerator CHARGER_PROP_CONSTANT_CHARGE_VOLTAGE_UV

Configuration of charge voltage regulation target in µV.

enumerator CHARGER_PROP_INPUT_REGULATION_CURRENT_UA

Configuration of the input current regulation target in µA.

This value is a rising current threshold that is regulated by reducing the charge current output

enumerator CHARGER_PROP_INPUT_REGULATION_VOLTAGE_UV

Configuration of the input voltage regulation target in µV.

This value is a falling voltage threshold that is regulated by reducing the charge current output

enumerator CHARGER_PROP_INPUT_CURRENT_NOTIFICATION

Configuration to issue a notification to the system based on the input current level and timing.

Value should be of type struct *charger_current_notifier*

enumerator CHARGER_PROP_DISCHARGE_CURRENT_NOTIFICATION

Configuration to issue a notification to the system based on the battery discharge current level and timing.

Value should be of type struct *charger_current_notifier*

enumerator CHARGER_PROP_SYSTEM_VOLTAGE_NOTIFICATION_UV

Configuration of the falling system voltage threshold where a notification is issued to the system, measured in µV.

enumerator CHARGER_PROP_STATUS_NOTIFICATION

Configuration to issue a notification to the system based on the charger status change.

Value should be of type charger_status_notifier_t

enumerator CHARGER_PROP_ONLINE_NOTIFICATION

Configuration to issue a notification to the system based on the charger online change.

Value should be of type charger_online_notifier_t

enumerator CHARGER_PROP_COMMON_COUNT

Reserved to demark end of common charger properties.

enumerator CHARGER_PROP_CUSTOM_BEGIN = *CHARGER_PROP_COMMON_COUNT* + 1

Reserved to demark downstream custom properties - use this value as the actual value may change over future versions of this API.

enumerator CHARGER_PROP_MAX = UINT16_MAX

Reserved to demark end of valid enum properties.

enum charger_online

External supply states.

Values:

enumerator CHARGER_ONLINE_OFFLINE = 0

External supply not present.

enumerator CHARGER_ONLINE_FIXED

External supply is present and of fixed output.

enumerator CHARGER_ONLINE_PROGRAMMABLE

External supply is present and of programmable output.

enum charger_status

Charging states.

Values:

enumerator CHARGER_STATUS_UNKNOWN = 0

Charging device state is unknown.

enumerator CHARGER_STATUS_CHARGING

Charging device is charging a battery.

enumerator CHARGER_STATUS_DISCHARGING

Charging device is not able to charge a battery.

enumerator CHARGER_STATUS_NOT_CHARGING

Charging device is not charging a battery.

enumerator CHARGER_STATUS_FULL

The battery is full and the charging device will not attempt charging.

enum charger_charge_type

Charge algorithm types.

Values:

enumerator CHARGER_CHARGE_TYPE_UNKNOWN = 0

Charge type is unknown.

enumerator CHARGER_CHARGE_TYPE_NONE

Charging is not occurring.

enumerator CHARGER_CHARGE_TYPE_TRICKLE

Charging is occurring at the slowest desired charge rate, typically for battery detection or preconditioning.

enumerator CHARGER_CHARGE_TYPE_FAST

Charging is occurring at the fastest desired charge rate.

enumerator CHARGER_CHARGE_TYPE_STANDARD

Charging is occurring at a moderate charge rate.

enumerator CHARGER_CHARGE_TYPE_ADAPTIVE

enumerator CHARGER_CHARGE_TYPE_LONGLIFE

enumerator CHARGER_CHARGE_TYPE_BYPASS

enum charger_health

Charger health conditions.

These conditions determine the ability to, or the rate of, charge

Values:

enumerator CHARGER_HEALTH_UNKNOWN = 0

Charger health condition is unknown.

enumerator CHARGER_HEALTH_GOOD

Charger health condition is good.

enumerator CHARGER_HEALTH_OVERHEAT

The charger device is overheated.

enumerator CHARGER_HEALTH_OVERVOLTAGE

The battery voltage has exceeded its overvoltage threshold.

enumerator CHARGER_HEALTH_UNSPEC_FAILURE

The battery or charger device is experiencing an unspecified failure.

enumerator CHARGER_HEALTH_COLD

The battery temperature is below the “cold” threshold.

enumerator CHARGER_HEALTH_WATCHDOG_TIMER_EXPIRE

The charger device’s watchdog timer has expired.

enumerator CHARGER_HEALTH_SAFETY_TIMER_EXPIRE

The charger device’s safety timer has expired.

enumerator CHARGER_HEALTH_CALIBRATION_REQUIRED

The charger device requires calibration.

enumerator CHARGER_HEALTH_WARM
The battery temperature is in the “warm” range.

enumerator CHARGER_HEALTH_COOL
The battery temperature is in the “cool” range.

enumerator CHARGER_HEALTH_HOT
The battery temperature is below the “hot” threshold.

enumerator CHARGER_HEALTH_NO_BATTERY
The charger device does not detect a battery.

enum charger_notification_severity
Charger severity levels for system notifications.

Values:

enumerator CHARGER_SEVERITY_PEAK = 0
Most severe level, typically triggered instantaneously.

enumerator CHARGER_SEVERITY_CRITICAL
More severe than the warning level, less severe than peak.

enumerator CHARGER_SEVERITY_WARNING
Base severity level.

Functions

int charger_get_prop(const struct *device* *dev, const *charger_prop_t* prop, union *charger_propval* *val)
Fetch a battery charger property.

Parameters

- dev – Pointer to the battery charger device
- prop – Charger property to get
- val – Pointer to *charger_propval* union

Return values

- 0 – if successful
- < – 0 if getting property failed

int charger_set_prop(const struct *device* *dev, const *charger_prop_t* prop, const union *charger_propval* *val)
Set a battery charger property.

Parameters

- dev – Pointer to the battery charger device
- prop – Charger property to set
- val – Pointer to *charger_propval* union

Return values

- `0` – if successful
- `<` – 0 if setting property failed

int `charger_charge_enable`(const struct *device* *dev, const bool enable)

Enable or disable a charge cycle.

Parameters

- `dev` – Pointer to the battery charger device
- `enable` – true enables a charge cycle, false disables a charge cycle

Return values

- `0` – if successful
- `-EIO` – if communication with the charger failed
- `-EINVAL` – if the conditions for initiating charging are invalid

struct `charger_current_notifier`

#include <*charger.h*> The input current thresholds for the charger to notify the system.

Public Members

uint8_t `severity`

The severity of the notification where CHARGER_SEVERITY_PEAK is the most severe.

uint32_t `current_ua`

The current threshold to be exceeded.

uint32_t `duration_us`

The duration of excess current before notifying the system.

union `charger_propval`

#include <*charger.h*> container for a charger_property value

Public Members

enum *charger_online* `online`

CHARGER_PROP_ONLINE.

bool `present`

CHARGER_PROP_PRESENT.

enum *charger_status* `status`

CHARGER_PROP_STATUS.

enum *charger_charge_type* `charge_type`

CHARGER_PROP_CHARGE_TYPE.

enum *charger_health* **health**
CHARGER_PROP_HEALTH.

uint32_t **const_charge_current_ua**
CHARGER_PROP_CONSTANT_CHARGE_CURRENT_UA.

uint32_t **precharge_current_ua**
CHARGER_PROP_PRECHARGE_CURRENT_UA.

uint32_t **charge_term_current_ua**
CHARGER_PROP_CHARGE_TERM_CURRENT_UA.

uint32_t **const_charge_voltage_uv**
CHARGER_PROP_CONSTANT_CHARGE_VOLTAGE_UV.

uint32_t **input_current_regulation_current_ua**
CHARGER_PROP_INPUT_REGULATION_CURRENT_UA.

uint32_t **input_voltage_regulation_voltage_uv**
CHARGER_PROP_INPUT_REGULATION_VOLTAGE_UV.

struct *charger_current_notifier* **input_current_notification**
CHARGER_PROP_INPUT_CURRENT_NOTIFICATION.

struct *charger_current_notifier* **discharge_current_notification**
CHARGER_PROP_DISCHARGE_CURRENT_NOTIFICATION.

uint32_t **system_voltage_notification**
CHARGER_PROP_SYSTEM_VOLTAGE_NOTIFICATION_UV.

charger_status_notifier_t **status_notification**
CHARGER_PROP_STATUS_NOTIFICATION.

charger_online_notifier_t **online_notification**
CHARGER_PROP_ONLINE_NOTIFICATION.

struct **charger_driver_api**
#include <*charger.h*> Charging device API.

Caching is entirely on the onus of the client

7.6.10 Coredump Device

Overview

The coredump device is a pseudo-device driver with two types.A COREDUMP_TYPE_MEMCPY type exposes device tree bindings for memory address/size values to be included in any dump. And the driver exposes an API to add/remove dump memory regions at runtime. A COREDUMP_TYPE_CALLBACK device requires exactly one entry in the memory-regions array with a size of 0 and a desired size. The driver will statically allocate memory of the desired size and provide an API to register a callback function to fill that memory when a dump occurs.

Configuration Options

Related configuration options:

- CONFIG_COREDUMP_DEVICE

API Reference

group **coredump_device_interface**

Coredump pseudo-device driver APIs.

Typedefs

typedef void (***coredump_dump_callback_t**)(uintptr_t dump_area, size_t dump_area_size)

Callback that occurs at dump time, data copied into dump_area will be included in the dump that is generated.

Param dump_area
Pointer to area to copy data into for inclusion in dump

Param dump_area_size
Size of available memory at dump_area

Functions

static inline bool **coredump_device_register_memory**(const struct *device* *dev, struct *coredump_mem_region_node* *region)

Register a region of memory to be stored in core dump at the time it is generated.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **region** – Struct describing memory to be collected

Returns
true if registration succeeded

Returns
false if registration failed

static inline bool **coredump_device_unregister_memory**(const struct *device* *dev, struct *coredump_mem_region_node* *region)

Unregister a region of memory to be stored in core dump at the time it is generated.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **region** – Struct describing memory to be collected

Returns
true if unregistration succeeded

Returns
false if unregistration failed

static inline bool **coredump_device_register_callback**(const struct *device* *dev, *coredump_dump_callback_t* callback)

Register a callback to be invoked at dump time.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **callback** – Callback to be invoked at dump time

Returns
true if registration succeeded

Returns
false if registration failed

struct **coredump_mem_region_node**

#include <*coredump.h*> Structure describing a region in memory that may be stored in core dump at the time it is generated.

Instances of this are passed to the *coredump_device_register_memory()* and *coredump_device_unregister_memory()* functions to indicate addition and removal of memory regions to be captured

Public Members

sys_snode_t **node**

Node of single-linked list, do not modify.

uintptr_t **start**

Address of start of memory region.

size_t **size**

Size of memory region.

7.6.11 Counter

Overview

API Reference

Related code samples

Counter Alarm
Implement an alarm application using the counter API.

DS3231 TCXO RTC
Interact with a DS3231 real-time clock using the counter API and dedicated driver API.

group **counter_interface**

Counter Interface.

Since
1.14

Version
0.8.0

Counter device capabilities

`COUNTER_CONFIG_INFO_COUNT_UP`
Counter count up flag.

Flags used by counter_top_cfg.

`COUNTER_TOP_CFG_DONT_RESET`
Flag preventing counter reset when top value is changed.

If flags is set then counter is free running while top value is updated, otherwise counter is reset (see *counter_set_top_value()*).

`COUNTER_TOP_CFG_RESET_WHEN_LATE`
Flag instructing counter to reset itself if changing top value results in counter going out of new top value bound.

See *COUNTER_TOP_CFG_DONT_RESET*.

Alarm configuration flags

Used in alarm configuration structure (*counter_alarm_cfg*).

`COUNTER_ALARM_CFG_ABSOLUTE`
Counter alarm absolute value flag.

Ticks relation to counter value. If set ticks are treated as absolute value, else it is relative to the counter reading performed during the call.

`COUNTER_ALARM_CFG_EXPIRE_WHEN_LATE`
Alarm flag enabling immediate expiration when driver detects that absolute alarm was set too late.

Alarm callback must be called from the same context as if it was set on time.

Counter guard period flags

Used by *counter_set_guard_period* and *counter_get_guard_period*.

`COUNTER_GUARD_PERIOD_LATE_TO_SET`
Identifies guard period needed for detection of late setting of absolute alarm (see *counter_set_channel_alarm*).

Typedefs

typedef void (*`counter_alarm_callback_t`)(const struct *device* *dev, uint8_t chan_id, uint32_t ticks, void *user_data)

Alarm callback.

Param dev
Pointer to the device structure for the driver instance.

Param chan_id
Channel ID.

Param ticks
Counter value that triggered the alarm.

Param user_data
User data.

typedef void (*`counter_top_callback_t`)(const struct *device* *dev, void *user_data)

Callback called when counter turns around.

Param dev
Pointer to the device structure for the driver instance.

Param user_data
User data provided in *counter_set_top_value*.

typedef int (*`counter_api_start`)(const struct *device* *dev)

typedef int (*`counter_api_stop`)(const struct *device* *dev)

typedef int (*`counter_api_get_value`)(const struct *device* *dev, uint32_t *ticks)

typedef int (*`counter_api_get_value_64`)(const struct *device* *dev, uint64_t *ticks)

typedef int (*`counter_api_set_alarm`)(const struct *device* *dev, uint8_t chan_id, const struct *counter_alarm_cfg* *alarm_cfg)

typedef int (*`counter_api_cancel_alarm`)(const struct *device* *dev, uint8_t chan_id)

typedef int (*`counter_api_set_top_value`)(const struct *device* *dev, const struct *counter_top_cfg* *cfg)

typedef uint32_t (*`counter_api_get_pending_int`)(const struct *device* *dev)

typedef uint32_t (*`counter_api_get_top_value`)(const struct *device* *dev)

typedef uint32_t (*`counter_api_get_guard_period`)(const struct *device* *dev, uint32_t flags)

typedef int (*`counter_api_set_guard_period`)(const struct *device* *dev, uint32_t ticks, uint32_t flags)

typedef uint32_t (*`counter_api_get_freq`)(const struct *device* *dev)

Functions

bool `counter_is_counting_up`(const struct *device* *dev)

Function to check if counter is counting up.

Parameters

- `dev` – **[in]** Pointer to the device structure for the driver instance.

Return values

- `true` – if counter is counting up.
- `false` – if counter is counting down.

uint8_t `counter_get_num_of_channels`(const struct *device* *dev)

Function to get number of alarm channels.

Parameters

- `dev` – **[in]** Pointer to the device structure for the driver instance.

Returns

Number of alarm channels.

uint32_t `counter_get_frequency`(const struct *device* *dev)

Function to get counter frequency.

Parameters

- `dev` – **[in]** Pointer to the device structure for the driver instance.

Returns

Frequency of the counter in Hz, or zero if the counter does not have a fixed frequency.

uint32_t `counter_us_to_ticks`(const struct *device* *dev, uint64_t us)

Function to convert microseconds to ticks.

Parameters

- `dev` – **[in]** Pointer to the device structure for the driver instance.
- `us` – **[in]** Microseconds.

Returns

Converted ticks. Ticks will be saturated if exceed 32 bits.

uint64_t `counter_ticks_to_us`(const struct *device* *dev, uint32_t ticks)

Function to convert ticks to microseconds.

Parameters

- `dev` – **[in]** Pointer to the device structure for the driver instance.
- `ticks` – **[in]** Ticks.

Returns

Converted microseconds.

uint32_t `counter_get_max_top_value`(const struct *device* *dev)

Function to retrieve maximum top value that can be set.

Parameters

- `dev` – **[in]** Pointer to the device structure for the driver instance.

Returns

Max top value.

int `counter_start`(const struct *device* *dev)

Start counter device in free running mode.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Return values

- `0` – If successful.
- `Negative` – errno code if failure.

int `counter_stop`(const struct *device* *dev)

Stop counter device.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Return values

- `0` – If successful.
- `-ENOTSUP` – if the device doesn't support stopping the counter.

int `counter_get_value`(const struct *device* *dev, uint32_t *ticks)

Get current counter value.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `ticks` – Pointer to where to store the current counter value

Return values

- `0` – If successful.
- `Negative` – error code on failure getting the counter value

int `counter_get_value_64`(const struct *device* *dev, uint64_t *ticks)

Get current counter 64-bit value.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `ticks` – Pointer to where to store the current counter value

Return values

- `0` – If successful.
- `Negative` – error code on failure getting the counter value

int `counter_set_channel_alarm`(const struct *device* *dev, uint8_t chan_id, const struct *counter_alarm_cfg* *alarm_cfg)

Set a single shot alarm on a channel.

After expiration alarm can be set again, disabling is not needed. When alarm expiration handler is called, channel is considered available and can be set again in that context.

> **Note**
>
> API is not thread safe.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `chan_id` – Channel ID.
- `alarm_cfg` – Alarm configuration.

Return values

- `0` – If successful.
- `-ENOTSUP` – if request is not supported (device does not support interrupts or requested channel).
- `-EINVAL` – if alarm settings are invalid.
- `-ETIME` – if absolute alarm was set too late.
- `-EBUSY` – if alarm is already active.

int `counter_cancel_channel_alarm`(const struct *device* *dev, uint8_t chan_id)

Cancel an alarm on a channel.

> **Note**
>
> API is not thread safe.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `chan_id` – Channel ID.

Return values

- `0` – If successful.
- `-ENOTSUP` – if request is not supported or the counter was not started yet.

int `counter_set_top_value`(const struct *device* *dev, const struct *counter_top_cfg* *cfg)

Set counter top value.

Function sets top value and optionally resets the counter to 0 or top value depending on counter direction. On turnaround, counter can be reset and optional callback is periodically called. Top value can only be changed when there is no active channel alarm.

COUNTER_TOP_CFG_DONT_RESET prevents counter reset. When counter is running while top value is updated, it is possible that counter progresses outside the new top value. In that case, error is returned and optionally driver can reset the counter (see *COUNTER_TOP_CFG_RESET_WHEN_LATE*).

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `cfg` – Configuration. Cannot be NULL.

Return values

- `0` – If successful.
- `-ENOTSUP` – if request is not supported (e.g. top value cannot be changed or counter cannot/must be reset during top value update).
- `-EBUSY` – if any alarm is active.
- `-ETIME` – if *COUNTER_TOP_CFG_DONT_RESET* was set and new top value is smaller than current counter value (counter counting up).

int **counter_get_pending_int**(const struct *device* *dev)

Function to get pending interrupts.

The purpose of this function is to return the interrupt status register for the device. This is especially useful when waking up from low power states to check the wake up source.

Parameters

- **dev** – Pointer to the device structure for the driver instance.

Return values

- **1** – if any counter interrupt is pending.
- **0** – if no counter interrupt is pending.

uint32_t **counter_get_top_value**(const struct *device* *dev)

Function to retrieve current top value.

Parameters

- **dev** – **[in]** Pointer to the device structure for the driver instance.

Returns

Top value.

int **counter_set_guard_period**(const struct *device* *dev, uint32_t ticks, uint32_t flags)

Set guard period in counter ticks.

When setting an absolute alarm value close to the current counter value there is a risk that the counter will have counted past the given absolute value before the driver manages to activate the alarm. If this would go unnoticed then the alarm would only expire after the timer has wrapped and reached the given absolute value again after a full timer period. This could take a long time in case of a 32 bit timer. Setting a sufficiently large guard period will help the driver detect unambiguously whether it is late or not.

The guard period should be as many counter ticks as the driver will need at most to actually activate the alarm after the driver API has been called. If the driver finds that the counter has just passed beyond the given absolute tick value but is still close enough to fall within the guard period, it will assume that it is "late", i.e. that the intended expiry time has already passed. Depending on the *COUNTER_ALARM_CFG_EXPIRE_WHEN_LATE* flag the driver will either ignore the alarm or expire it immediately in such a case.

If, however, the counter is past the given absolute tick value but outside the guard period, then the driver will assume that this is intentional and let the counter wrap around to/from zero before it expires.

More precisely:

- When counting upwards (see *COUNTER_CONFIG_INFO_COUNT_UP*) the given absolute tick value must be above (now + guard_period) % top_value to be accepted by the driver.
- When counting downwards, the given absolute tick value must be less than (now + top_value - guard_period) % top_value to be accepted.

Examples:

- counting upwards, now = 4950, top value = 5000, guard period = 100: absolute tick value >= (4950 + 100) % 5000 = 50

- counting downwards, now = 50, top value = 5000, guard period = 100: absolute tick value <= (50 + 5000 - * 100) % 5000 = 4950

If you need only short alarm periods, you can set the guard period very high (e.g. half of the counter top value) which will make it highly unlikely that the counter will ever unintentionally wrap.

The guard period is set to 0 on initialization (no protection).

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `ticks` – Guard period in counter ticks.
- `flags` – See *COUNTER_GUARD_PERIOD_FLAGS*.

Return values

- `0` – if successful.
- `-ENOTSUP` – if function or flags are not supported.
- `-EINVAL` – if ticks value is invalid.

uint32_t **counter_get_guard_period**(const struct *device* *dev, uint32_t flags)

Return guard period.

See also

counter_set_guard_period.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `flags` – See *COUNTER_GUARD_PERIOD_FLAGS*.

Returns

Guard period given in counter ticks or 0 if function or flags are not supported.

struct **counter_alarm_cfg**

#include <counter.h> Alarm callback structure.

Public Members

counter_alarm_callback_t `callback`

Callback called on alarm (cannot be NULL).

uint32_t `ticks`

Number of ticks that triggers the alarm.

It can be relative (to now) or an absolute value (see *COUNTER_ALARM_CFG_ABSOLUTE*). Both, relative and absolute, alarm values can be any value between zero and the current top value (see *counter_get_top_value*). When setting an absolute alarm value close to the current counter value there is a risk that the counter will have counted past the given absolute value before the driver manages to activate the alarm. Therefore a guard period can

be defined that lets the driver decide unambiguously whether it is late or not (see *counter_set_guard_period*). If the counter is clock driven then ticks can be converted to microseconds (see *counter_ticks_to_us*). Alternatively, the counter implementation may count asynchronous events.

void *`user_data`

User data returned in callback.

uint32_t `flags`

Alarm flags (see *COUNTER_ALARM_FLAGS*).

struct `counter_top_cfg`

#include <*counter.h*> Top value configuration structure.

Public Members

uint32_t `ticks`

Top value.

counter_top_callback_t `callback`

Callback function (can be NULL).

void *`user_data`

User data passed to callback function (not valid if callback is NULL).

uint32_t `flags`

Flags (see *COUNTER_TOP_FLAGS*).

struct `counter_config_info`

#include <*counter.h*> Structure with generic counter features.

Public Members

uint32_t `max_top_value`

Maximal (default) top value on which counter is reset (cleared or reloaded).

uint32_t `freq`

Frequency of the source clock if synchronous events are counted.

uint8_t `flags`

Flags (see *COUNTER_FLAGS*).

uint8_t `channels`

Number of channels that can be used for setting alarm.

See also

counter_set_channel_alarm

struct `counter_driver_api`

#include <counter.h>

7.6.12 Digital-to-Analog Converter (DAC)

Overview

The DAC API provides access to Digital-to-Analog Converter (DAC) devices.

Configuration Options

Related configuration options:

- `CONFIG_DAC`

API Reference

Related code samples

Digital-to-Analog Converter (DAC)
Generate an analog sawtooth signal using the DAC driver API.

group `dac_interface`

DAC driver APIs.

Since
2.3

Version
0.8.0

Functions

int `dac_channel_setup`(const struct *device* *dev, const struct *dac_channel_cfg* *channel_cfg)

Configure a DAC channel.

It is required to call this function and configure each channel before it is selected for a write request.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `channel_cfg` – Channel configuration.

Return values

- `0` – On success.
- `-EINVAL` – If a parameter with an invalid value has been provided.
- `-ENOTSUP` – If the requested resolution is not supported.

int `dac_write_value`(const struct *device* *dev, uint8_t channel, uint32_t value)

Write a single value to a DAC channel.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `channel` – Number of the channel to be used.
- `value` – Data to be written to DAC output registers.

Return values

- `0` – On success.
- `-EINVAL` – If a parameter with an invalid value has been provided.

struct `dac_channel_cfg`

#include <*dac.h*> Structure for specifying the configuration of a DAC channel.

Public Members

uint8_t `channel_id`

Channel identifier of the DAC that should be configured.

uint8_t `resolution`

Desired resolution of the DAC (depends on device capabilities).

bool `buffered`

Enable output buffer for this channel.

This is relevant for instance if the output is directly connected to the load, without an amplifierin between. The actual details on this are hardware dependent.

7.6.13 Direct Memory Access (DMA)

Overview

Direct Memory Access (Controller) is a commonly provided type of co-processor that can typically offload transferring data to and from peripherals and memory.

The DMA API is not a portable API and really cannot be as each DMA has unique memory requirements, peripheral interactions, and features. The API in effect provides a union of all useful DMA functionality drivers have needed in the tree. It can still be a good abstraction, with care, for peripheral devices for vendors where the DMA IP might be very similar but have slight variances.

Driver Implementation Expectations

Synchronization and Ownership From an API point of view, a DMA channel is a single-owner object, meaning the drivers should not attempt to wrap a channel with kernel synchronization primitives such as mutexes or semaphores. If DMA channels require mutating shared registers, those register updates should be wrapped in a spin lock.

This enables the entire API to be low-cost and callable from any call context, including ISRs where it may be very useful to start/stop/suspend/resume/reload a channel transfer.

Transfer Descriptor Memory Management Drivers should not attempt to use heap allocations of any kind. If object pools are needed for transfer descriptors then those should be setup in a way that does not break the promise of ISR-allowable calls. Many drivers choose to create a simple static descriptor array per channel with the size of the descriptor array adjustable using Kconfig.

Channel State Machine Expectations DMA channels should be viewed as state machines that the DMA API provides transition events for in the form of API calls. Every driver is expected to maintain its own channel state tracking. The busy state of the channel should be inspectable at any time with *dma_get_status()*.

A diagram showing those expectated possible state transitions and their API calls is provided here for reference.

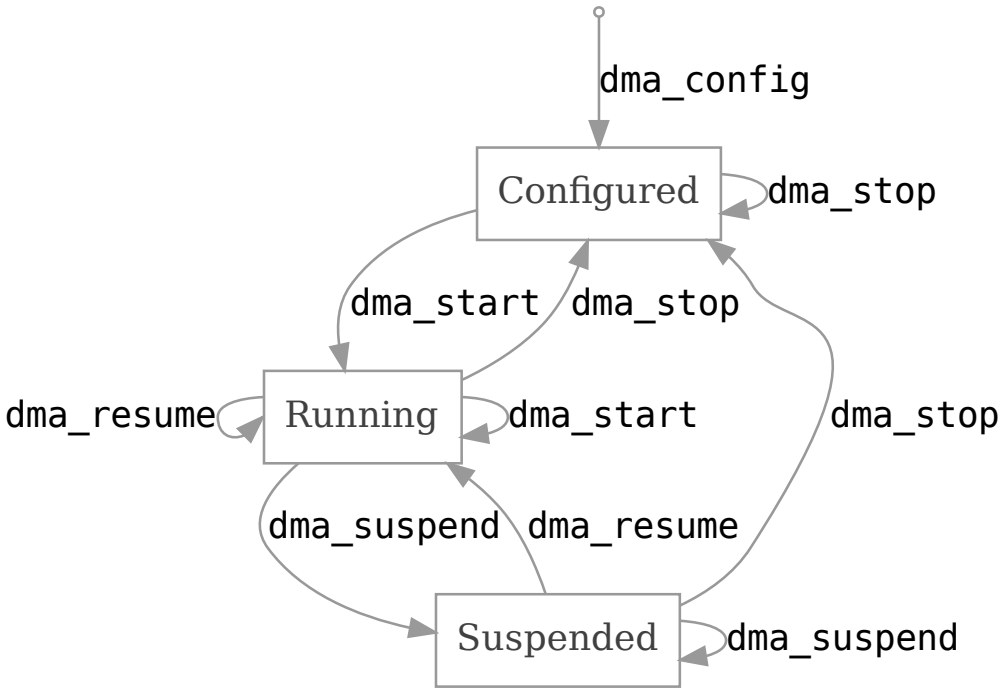


Fig. 2: DMA state finite state machine

API Reference

group `dma_interface`

DMA Interface.

Since
1.5

Version
1.0.0

Defines

`DMA_STATUS_COMPLETE`

The DMA callback event has occurred at the completion of a transfer list.

`DMA_STATUS_BLOCK`

The DMA callback has occurred at the completion of a single transfer block in a transfer list.

`DMA_MAGIC`

Magic code to identify context content.

`DMA_BUF_ADDR_ALIGNMENT`(node)

Get the device tree property describing the buffer address alignment.

Useful when statically defining or allocating buffers for DMA usage where memory alignment often matters.

Parameters

- **node** – Node identifier, e.g. *DT_NODELABEL(dma_0)*

Returns
alignment Memory byte alignment required for DMA buffers

`DMA_BUF_SIZE_ALIGNMENT`(node)

Get the device tree property describing the buffer size alignment.

Useful when statically defining or allocating buffers for DMA usage where memory alignment often matters.

Parameters

- **node** – Node identifier, e.g. *DT_NODELABEL(dma_0)*

Returns
alignment Memory byte alignment required for DMA buffers

`DMA_COPY_ALIGNMENT`(node)

Get the device tree property describing the minimal chunk of data possible to be copied.

Parameters

- **node** – Node identifier, e.g. *DT_NODELABEL(dma_0)*

Returns
minimal Minimal chunk of data possible to be copied

Typedefs

typedef void (*dma_callback_t)(const struct *device* *dev, void *user_data, uint32_t channel, int status)

Callback function for DMA transfer completion.

If enabled, callback function will be invoked at transfer or block completion, or when an error happens. In circular mode, `status` indicates that the DMA device has reached either the end of the buffer (DMA_STATUS_COMPLETE) or a water mark (DMA_STATUS_BLOCK).

Param dev
Pointer to the DMA device calling the callback.

Param user_data
A pointer to some user data or NULL

Param channel
The channel number

Param status
Status of the transfer

- DMA_STATUS_COMPLETE buffer fully consumed
- DMA_STATUS_BLOCK buffer consumption reached a configured block or water mark
- A negative errno otherwise

Enums

enum dma_channel_direction

DMA channel direction.

Values:

enumerator MEMORY_TO_MEMORY = 0x0

Memory to memory.

enumerator MEMORY_TO_PERIPHERAL

Memory to peripheral.

enumerator PERIPHERAL_TO_MEMORY

Peripheral to memory.

enumerator PERIPHERAL_TO_PERIPHERAL

Peripheral to peripheral.

enumerator HOST_TO_MEMORY

Host to memory.

enumerator MEMORY_TO_HOST

Memory to host.

enumerator DMA_CHANNEL_DIRECTION_COMMON_COUNT
Number of all common channel directions.

enumerator DMA_CHANNEL_DIRECTION_PRIV_START = *DMA_CHANNEL_DIRECTION_COMMON_COUNT*
This and higher values are dma controller or soc specific.
Refer to the specified dma driver header file.

enumerator DMA_CHANNEL_DIRECTION_MAX = 0x7
Maximum allowed value (3 bit field!)

enum dma_addr_adj
DMA address adjustment.
Valid values for *source_addr_adj* and *dest_addr_adj*
Values:

enumerator DMA_ADDR_ADJ_INCREMENT
Increment the address.

enumerator DMA_ADDR_ADJ_DECREMENT
Decrement the address.

enumerator DMA_ADDR_ADJ_NO_CHANGE
No change the address.

enum dma_channel_filter
DMA channel attributes.
Values:

enumerator DMA_CHANNEL_NORMAL

enumerator DMA_CHANNEL_PERIODIC

enum dma_attribute_type
DMA attributes.
Values:

enumerator DMA_ATTR_BUFFER_ADDRESS_ALIGNMENT

enumerator DMA_ATTR_BUFFER_SIZE_ALIGNMENT

enumerator DMA_ATTR_COPY_ALIGNMENT

enumerator DMA_ATTR_MAX_BLOCK_COUNT

Functions

static inline int **dma_config**(const struct *device* *dev, uint32_t channel, struct *dma_config* *config)

Configure individual channel for DMA transfer.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **channel** – Numeric identification of the channel to configure
- **config** – Data structure containing the intended configuration for the selected channel

Return values

- **0** – if successful.
- **Negative** – errno code if failure.

static inline int **dma_reload**(const struct *device* *dev, uint32_t channel, uint32_t src, uint32_t dst, size_t size)

Reload buffer(s) for a DMA channel.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **channel** – Numeric identification of the channel to configure selected channel
- **src** – source address for the DMA transfer
- **dst** – destination address for the DMA transfer
- **size** – size of DMA transfer

Return values

- **0** – if successful.
- **Negative** – errno code if failure.

int **dma_start**(const struct *device* *dev, uint32_t channel)

Enables DMA channel and starts the transfer, the channel must be configured beforehand.

Implementations must check the validity of the channel ID passed in and return -EINVAL if it is invalid.

Start is allowed on channels that have already been started and must report success.

Function properties (list may not be complete)

isr-ok

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **channel** – Numeric identification of the channel where the transfer will be processed

Return values

- **0** – if successful.
- **Negative** – errno code if failure.

int `dma_stop`(const struct *device* *dev, uint32_t channel)

Stops the DMA transfer and disables the channel.

Implementations must check the validity of the channel ID passed in and return -EINVAL if it is invalid.

Stop is allowed on channels that have already been stopped and must report success.

Function properties (list may not be complete)

isr-ok

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `channel` – Numeric identification of the channel where the transfer was being processed

Return values

- `0` – if successful.
- `Negative` – errno code if failure.

int `dma_suspend`(const struct *device* *dev, uint32_t channel)

Suspend a DMA channel transfer.

Implementations must check the validity of the channel state and ID passed in and return -EINVAL if either are invalid.

Function properties (list may not be complete)

isr-ok

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `channel` – Numeric identification of the channel to suspend

Return values

- `0` – If successful.
- `-ENOSYS` – If not implemented.
- `-EINVAL` – If invalid channel id or state.
- `-errno` – Other negative errno code failure.

int `dma_resume`(const struct *device* *dev, uint32_t channel)

Resume a DMA channel transfer.

Implementations must check the validity of the channel state and ID passed in and return -EINVAL if either are invalid.

Function properties (list may not be complete)

isr-ok

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `channel` – Numeric identification of the channel to resume

Return values

- `0` – If successful.
- `-ENOSYS` – If not implemented
- `-EINVAL` – If invalid channel id or state.
- `-errno` – Other negative errno code failure.

int `dma_request_channel`(const struct *device* *dev, void *filter_param)

request DMA channel.

request DMA channel resources return -EINVAL if there is no valid channel available.

Function properties (list may not be complete)

isr-ok

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `filter_param` – filter function parameter

Return values

- `dma` – channel if successful.
- `Negative` – errno code if failure.

void `dma_release_channel`(const struct *device* *dev, uint32_t channel)

release DMA channel.

release DMA channel resources

Function properties (list may not be complete)

isr-ok

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `channel` – channel number

int `dma_chan_filter`(const struct *device* *dev, int channel, void *filter_param)

DMA channel filter.

filter channel by attribute

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `channel` – channel number
- `filter_param` – filter attribute

Return values

`Negative` – errno code if not support

static inline int **dma_get_status**(const struct *device* *dev, uint32_t channel, struct *dma_status* *stat)

get current runtime status of DMA transfer

Implementations must check the validity of the channel ID passed in and return -EINVAL if it is invalid or -ENOSYS if not supported.

Function properties (list may not be complete)

isr-ok

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **channel** – Numeric identification of the channel where the transfer was being processed
- **stat** – a non-NULL *dma_status* object for storing DMA status

Return values

- **non-negative** – if successful.
- **Negative** – errno code if failure.

static inline int **dma_get_attribute**(const struct *device* *dev, uint32_t type, uint32_t *value)

get attribute of a dma controller

This function allows to get a device specific static or runtime attribute like required address and size alignment of a buffer. Implementations must check the validity of the type passed in and return -EINVAL if it is invalid or -ENOSYS if not supported.

Function properties (list may not be complete)

isr-ok

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **type** – Numeric identification of the attribute
- **value** – A non-NULL pointer to the variable where the read value is to be placed

Return values

- **non-negative** – if successful.
- **Negative** – errno code if failure.

static inline uint32_t **dma_width_index**(uint32_t size)

Look-up generic width index to be used in registers.

Warning

This look-up works for most controllers, but *may* not work for yours. Ensure your controller expects the most common register bit values before using this convenience function. If your controller does not support these values, you will have to write your own look-up inside the controller driver.

Parameters

- `size` – width of bus (in bytes)

Return values

`common` – DMA index to be placed into registers.

static inline uint32_t `dma_burst_index`(uint32_t burst)

Look-up generic burst index to be used in registers.

> **Warning**
>
> This look-up works for most controllers, but *may* not work for yours. Ensure your controller expects the most common register bit values before using this convenience function. If your controller does not support these values, you will have to write your own look-up inside the controller driver.

Parameters

- `burst` – number of bytes to be sent in a single burst

Return values

`common` – DMA index to be placed into registers.

struct `dma_block_config`

#include <*dma.h*> DMA block configuration structure.

Aside from source address, destination address, and block size many of these options are hardware and driver dependent.

Public Members

uint32_t `source_address`

block starting address at source

uint32_t `dest_address`

block starting address at destination

uint32_t `source_gather_interval`

Address adjustment at gather boundary.

uint32_t `dest_scatter_interval`

Address adjustment at scatter boundary.

uint16_t `dest_scatter_count`

Continuous transfer count between scatter boundaries.

uint16_t `source_gather_count`

Continuous transfer count between gather boundaries.

uint32_t `block_size`

Number of bytes to be transferred for this block.

struct *dma_block_config* *next_block

Pointer to next block in a transfer list.

uint16_t source_gather_en

Enable source gathering when set to 1.

uint16_t dest_scatter_en

Enable destination scattering when set to 1.

uint16_t source_addr_adj

Source address adjustment option.

- 0b00 increment
- 0b01 decrement
- 0b10 no change

uint16_t dest_addr_adj

Destination address adjustment.

- 0b00 increment
- 0b01 decrement
- 0b10 no change

uint16_t source_reload_en

Reload source address at the end of block transfer.

uint16_t dest_reload_en

Reload destination address at the end of block transfer.

uint16_t fifo_mode_control

FIFO fill before starting transfer, HW specific meaning.

uint16_t flow_control_mode

Transfer flow control mode.

- 0b0 source request service upon data availability
- 0b1 source request postponed until destination request happens

struct dma_config

#include <*dma.h*> DMA configuration structure.

Public Members

uint32_t dma_slot

Which peripheral and direction, HW specific.

uint32_t `channel_direction`

Direction the transfers are occurring.

- 0b000 memory to memory,
- 0b001 memory to peripheral,
- 0b010 peripheral to memory,
- 0b011 peripheral to peripheral,
- 0b100 host to memory
- 0b101 memory to host
- others hardware specific

uint32_t `complete_callback_en`

Completion callback enable.

- 0b0 callback invoked at transfer list completion only
- 0b1 callback invoked at completion of each block

uint32_t `error_callback_dis`

Error callback disable.

- 0b0 error callback enabled
- 0b1 error callback disabled

uint32_t `source_handshake`

Source handshake, HW specific.

- 0b0 HW
- 0b1 SW

uint32_t `dest_handshake`

Destination handshake, HW specific.

- 0b0 HW
- 0b1 SW

uint32_t `channel_priority`

Channel priority for arbitration, HW specific.

uint32_t `source_chaining_en`

Source chaining enable, HW specific.

uint32_t `dest_chaining_en`

Destination chaining enable, HW specific.

uint32_t `linked_channel`

Linked channel, HW specific.

uint32_t `cyclic`

Cyclic transfer list, HW specific.

uint32_t **source_data_size**
Width of source data (in bytes)

uint32_t **dest_data_size**
Width of destination data (in bytes)

uint32_t **source_burst_length**
Source burst length in bytes.

uint32_t **dest_burst_length**
Destination burst length in bytes.

uint32_t **block_count**
Number of blocks in transfer list.

struct *dma_block_config* ***head_block**
Pointer to the first block in the transfer list.

void ***user_data**
Optional attached user data for callbacks.

dma_callback_t **dma_callback**
Optional callback for completion and error events.

struct **dma_status**
#include <dma.h> DMA runtime status structure.

Public Members

bool **busy**
Is the current DMA transfer busy or idle.

enum *dma_channel_direction* **dir**
Direction for the transfer.

uint32_t **pending_length**
Pending length to be transferred in bytes, HW specific.

uint32_t **free**
Available buffers space, HW specific.

uint32_t **write_position**
Write position in circular DMA buffer, HW specific.

uint32_t **read_position**
Read position in circular DMA buffer, HW specific.

uint64_t **total_copied**

Total copied, HW specific.

struct **dma_context**

#include <*dma.h*> DMA context structure Note: the *dma_context* shall be the first member of DMA client driver Data, got by dev->data.

Public Members

int32_t **magic**

magic code to identify the context

int **dma_channels**

number of dma channels

atomic_t ***atomic**

atomic holding bit flags for each channel to mark as used/unused

7.6.14 Display Interface

API Reference

Related code samples

Display
Draw basic rectangles on a display device.

LVGL basic sample
Display a ”Hello World” and react to user input using LVGL.

LVGL demos
Run LVGL built-in demos.

LVGL line chart with accelerometer data
Display acceleration data on a real-time chart using LVGL.

Generic Display Interface

group **display_interface**

Display Interface.

Since
1.14

Version
0.8.0

Defines

DISPLAY_BITS_PER_PIXEL(fmt)

Bits required per pixel for display format.

This macro expands to the number of bits required for a given display format. It can be used to allocate a framebuffer based on a given display format type

Typedefs

typedef int (*display_blanking_on_api)(const struct *device* *dev)

Callback API to turn on display blanking See *display_blanking_on()* for argument description.

typedef int (*display_blanking_off_api)(const struct *device* *dev)

Callback API to turn off display blanking See *display_blanking_off()* for argument description.

typedef int (*display_write_api)(const struct *device* *dev, const uint16_t x, const uint16_t y, const struct *display_buffer_descriptor* *desc, const void *buf)

Callback API for writing data to the display See *display_write()* for argument description.

typedef int (*display_read_api)(const struct *device* *dev, const uint16_t x, const uint16_t y, const struct *display_buffer_descriptor* *desc, void *buf)

Callback API for reading data from the display See *display_read()* for argument description.

typedef void *(*display_get_framebuffer_api)(const struct *device* *dev)

Callback API to get framebuffer pointer See *display_get_framebuffer()* for argument description.

typedef int (*display_set_brightness_api)(const struct *device* *dev, const uint8_t brightness)

Callback API to set display brightness See *display_set_brightness()* for argument description.

typedef int (*display_set_contrast_api)(const struct *device* *dev, const uint8_t contrast)

Callback API to set display contrast See *display_set_contrast()* for argument description.

typedef void (*display_get_capabilities_api)(const struct *device* *dev, struct *display_capabilities* *capabilities)

Callback API to get display capabilities See *display_get_capabilities()* for argument description.

typedef int (*display_set_pixel_format_api)(const struct *device* *dev, const enum *display_pixel_format* pixel_format)

Callback API to set pixel format used by the display See *display_set_pixel_format()* for argument description.

typedef int (*`display_set_orientation_api`)(const struct *device* *dev, const enum *display_orientation* orientation)

Callback API to set orientation used by the display See *display_set_orientation()* for argument description.

Enums

enum `display_pixel_format`

Display pixel formats.

Display pixel format enumeration.

In case a pixel format consists out of multiple bytes the byte order is big endian.

Values:

enumerator `PIXEL_FORMAT_RGB_888` = *BIT*(0)

24-bit RGB

enumerator `PIXEL_FORMAT_MONO01` = *BIT*(1)

Monochrome (0=Black 1=White)

enumerator `PIXEL_FORMAT_MONO10` = *BIT*(2)

Monochrome (1=Black 0=White)

enumerator `PIXEL_FORMAT_ARGB_8888` = *BIT*(3)

32-bit ARGB

enumerator `PIXEL_FORMAT_RGB_565` = *BIT*(4)

16-bit RGB

enumerator `PIXEL_FORMAT_BGR_565` = *BIT*(5)

16-bit BGR

enum `display_screen_info`

Display screen information.

Values:

enumerator `SCREEN_INFO_MONO_VTILED` = *BIT*(0)

If selected, one octet represents 8 pixels ordered vertically, otherwise ordered horizontally.

enumerator `SCREEN_INFO_MONO_MSB_FIRST` = *BIT*(1)

If selected, the MSB represents the first pixel, otherwise MSB represents the last pixel.

enumerator `SCREEN_INFO_EPD` = *BIT*(2)

Electrophoretic Display.

enumerator SCREEN_INFO_DOUBLE_BUFFER = *BIT*(3)

Screen has two alternating ram buffers.

enumerator SCREEN_INFO_X_ALIGNMENT_WIDTH = *BIT*(4)

Screen has x alignment constrained to width.

enum display_orientation

Enumeration with possible display orientation.

Values:

enumerator DISPLAY_ORIENTATION_NORMAL

No rotation.

enumerator DISPLAY_ORIENTATION_ROTATED_90

Rotated 90 degrees clockwise.

enumerator DISPLAY_ORIENTATION_ROTATED_180

Rotated 180 degrees clockwise.

enumerator DISPLAY_ORIENTATION_ROTATED_270

Rotated 270 degrees clockwise.

Functions

static inline int display_write(const struct *device* *dev, const uint16_t x, const uint16_t y, const struct *display_buffer_descriptor* *desc, const void *buf)

Write data to display.

Parameters

- dev – Pointer to device structure
- x – x Coordinate of the upper left corner where to write the buffer
- y – y Coordinate of the upper left corner where to write the buffer
- desc – Pointer to a structure describing the buffer layout
- buf – Pointer to buffer array

Return values

0 – on success else negative errno code.

static inline int display_read(const struct *device* *dev, const uint16_t x, const uint16_t y, const struct *display_buffer_descriptor* *desc, void *buf)

Read data from display.

Parameters

- dev – Pointer to device structure
- x – x Coordinate of the upper left corner where to read from
- y – y Coordinate of the upper left corner where to read from
- desc – Pointer to a structure describing the buffer layout
- buf – Pointer to buffer array

Return values

- `0` – on success else negative errno code.
- `-ENOSYS` – if not implemented.

static inline void *`display_get_framebuffer`(const struct *device* *dev)

Get pointer to framebuffer for direct access.

Parameters

- `dev` – Pointer to device structure

Return values

`Pointer` – to frame buffer or NULL if direct framebuffer access is not supported

static inline int `display_blanking_on`(const struct *device* *dev)

Turn display blanking on.

This function blanks the complete display. The content of the frame buffer will be retained while blanking is enabled and the frame buffer will be accessible for read and write operations.

In case backlight control is supported by the driver the backlight is turned off. The backlight configuration is retained and accessible for configuration.

In case the driver supports display blanking the initial state of the driver would be the same as if this function was called.

Parameters

- `dev` – Pointer to device structure

Return values

- `0` – on success else negative errno code.
- `-ENOSYS` – if not implemented.

static inline int `display_blanking_off`(const struct *device* *dev)

Turn display blanking off.

Restore the frame buffer content to the display. In case backlight control is supported by the driver the backlight configuration is restored.

Parameters

- `dev` – Pointer to device structure

Return values

- `0` – on success else negative errno code.
- `-ENOSYS` – if not implemented.

static inline int `display_set_brightness`(const struct *device* *dev, uint8_t brightness)

Set the brightness of the display.

Set the brightness of the display in steps of 1/256, where 255 is full brightness and 0 is minimal.

Parameters

- `dev` – Pointer to device structure
- `brightness` – Brightness in steps of 1/256

Return values

- `0` – on success else negative errno code.

- `-ENOSYS` – if not implemented.

static inline int **display_set_contrast**(const struct *device* *dev, uint8_t contrast)

Set the contrast of the display.

Set the contrast of the display in steps of 1/256, where 255 is maximum difference and 0 is minimal.

Parameters

- `dev` – Pointer to device structure
- `contrast` – Contrast in steps of 1/256

Return values

- `0` – on success else negative errno code.
- `-ENOSYS` – if not implemented.

static inline void **display_get_capabilities**(const struct *device* *dev, struct *display_capabilities* *capabilities)

Get display capabilities.

Parameters

- `dev` – Pointer to device structure
- `capabilities` – Pointer to capabilities structure to populate

static inline int **display_set_pixel_format**(const struct *device* *dev, const enum *display_pixel_format* pixel_format)

Set pixel format used by the display.

Parameters

- `dev` – Pointer to device structure
- `pixel_format` – Pixel format to be used by display

Return values

- `0` – on success else negative errno code.
- `-ENOSYS` – if not implemented.

static inline int **display_set_orientation**(const struct *device* *dev, const enum *display_orientation* orientation)

Set display orientation.

Parameters

- `dev` – Pointer to device structure
- `orientation` – Orientation to be used by display

Return values

- `0` – on success else negative errno code.
- `-ENOSYS` – if not implemented.

struct **display_capabilities**

#include <display.h> Structure holding display capabilities.

Public Members

uint16_t **x_resolution**
Display resolution in the X direction.

uint16_t **y_resolution**
Display resolution in the Y direction.

uint32_t **supported_pixel_formats**
Bitwise or of pixel formats supported by the display.

uint32_t **screen_info**
Information about display panel.

enum *display_pixel_format* **current_pixel_format**
Currently active pixel format for the display.

enum *display_orientation* **current_orientation**
Current display orientation.

struct **display_buffer_descriptor**
#include <display.h> Structure to describe display data buffer layout.

Public Members

uint32_t **buf_size**
Data buffer size in bytes.

uint16_t **width**
Data buffer row width in pixels.

uint16_t **height**
Data buffer column height in pixels.

uint16_t **pitch**
Number of pixels between consecutive rows in the data buffer.

struct **display_driver_api**
#include <display.h> Display driver API API which a display driver should expose.

Related code samples

Grove LCD
Display an incrementing counter and change the backlight color.

Grove LCD Display

group grove_display

Grove display APIs.

Defines

GLCD_DS_DISPLAY_ON

GLCD_DS_DISPLAY_OFF

GLCD_DS_CURSOR_ON

GLCD_DS_CURSOR_OFF

GLCD_DS_BLINK_ON

GLCD_DS_BLINK_OFF

GLCD_IS_SHIFT_INCREMENT

GLCD_IS_SHIFT_DECREMENT

GLCD_IS_ENTRY_LEFT

GLCD_IS_ENTRY_RIGHT

GLCD_FS_8BIT_MODE

GLCD_FS_ROWS_2

GLCD_FS_ROWS_1

GLCD_FS_DOT_SIZE_BIG

GLCD_FS_DOT_SIZE_LITTLE

GROVE_RGB_WHITE

GROVE_RGB_RED

GROVE_RGB_GREEN

GROVE_RGB_BLUE

Functions

void **glcd_print**(const struct *device* *dev, char *data, uint32_t size)

Send text to the screen.

Parameters

- **dev** – Pointer to device structure for driver instance.
- **data** – the ASCII text to display
- **size** – the length of the text in bytes

void **glcd_cursor_pos_set**(const struct *device* *dev, uint8_t col, uint8_t row)

Set text cursor position for next additions.

Parameters

- **dev** – Pointer to device structure for driver instance.
- **col** – the column for the cursor to be moved to (0-15)
- **row** – the row it should be moved to (0 or 1)

void **glcd_clear**(const struct *device* *dev)

Clear the current display.

Parameters

- **dev** – Pointer to device structure for driver instance.

void **glcd_display_state_set**(const struct *device* *dev, uint8_t opt)

Function to change the display state.

This function provides the user the ability to change the state of the display as per needed. Controlling things like powering on or off the screen, the option to display the cursor or not, and the ability to blink the cursor.

Parameters

- **dev** – Pointer to device structure for driver instance.
- **opt** – An 8bit bitmask of GLCD_DS_* options.

uint8_t **glcd_display_state_get**(const struct *device* *dev)

return the display feature set associated with the device

Parameters

- **dev** – the Grove LCD to get the display features set

Returns

the display feature set associated with the device.

void **glcd_input_state_set**(const struct *device* *dev, uint8_t opt)

Function to change the input state.

This function provides the user the ability to change the state of the text input. Controlling things like text entry from the left or right side, and how far to increment on new text

Parameters

- **dev** – Pointer to device structure for driver instance.
- **opt** – A bitmask of GLCD_IS_* options

uint8_t **glcd_input_state_get**(const struct *device* *dev)

return the input set associated with the device

Parameters

- **dev** – the Grove LCD to get the input features set

Returns

the input set associated with the device.

void **glcd_function_set**(const struct *device* *dev, uint8_t opt)

Function to set the functional state of the display.

This function provides the user the ability to change the state of the display as per needed. Controlling things like the number of rows, dot size, and text display quality.

Parameters

- **dev** – Pointer to device structure for driver instance.
- **opt** – A bitmask of GLCD_FS_* options

uint8_t **glcd_function_get**(const struct *device* *dev)

return the function set associated with the device

Parameters

- **dev** – the Grove LCD to get the functions set

Returns

the function features set associated with the device.

void **glcd_color_select**(const struct *device* *dev, uint8_t color)

Set LCD background to a predefined color.

Parameters

- **dev** – Pointer to device structure for driver instance.
- **color** – One of the predefined color options

void **glcd_color_set**(const struct *device* *dev, uint8_t r, uint8_t g, uint8_t b)

Set LCD background to custom RGB color value.

Parameters

- **dev** – Pointer to device structure for driver instance.
- **r** – A numeric value for the red color (max is 255)
- **g** – A numeric value for the green color (max is 255)
- **b** – A numeric value for the blue color (max is 255)

BBC micro:bit Display

group **mb_display**

BBC micro:bit display APIs.

Defines

MB_IMAGE(_rows...)

Generate an image object from a given array rows/columns.

This helper takes an array of 5 rows, each consisting of 5 0/1 values which correspond to the columns of that row. The value 0 means the pixel is disabled whereas a 1 means the pixel is enabled.

The pixels go from left to right and top to bottom, i.e. top-left corner is the first row's first value, top-right is the first rows last value, and bottom-right corner is the last value of the last (5th) row. As an example, the following would create a smiley face image:

Parameters

- **_rows** – Each of the 5 rows represented as a 5-value column array.

Returns

Image bitmap that can be passed e.g. to *mb_display_image()*.

Enums

enum mb_display_mode

Display mode.

First 16 bits are reserved for modes, last 16 for flags.

Values:

enumerator MB_DISPLAY_MODE_DEFAULT

Default mode ("single" for images, "scroll" for text).

enumerator MB_DISPLAY_MODE_SINGLE

Display images sequentially, one at a time.

enumerator MB_DISPLAY_MODE_SCROLL

Display images by scrolling.

enumerator MB_DISPLAY_FLAG_LOOP = *BIT*(16)

Loop back to the beginning when reaching the last image.

Functions

struct mb_display *mb_display_get(void)

Get a pointer to the BBC micro:bit display object.

Returns

Pointer to display object.

void mb_display_image(struct mb_display *disp, uint32_t mode, int32_t duration, const struct *mb_image* *img, uint8_t img_count)

Display one or more images on the BBC micro:bit LED display.

This function takes an array of one or more images and renders them sequentially on the micro:bit display. The call is asynchronous, i.e. the processing of the display happens in the background. If there is another image being displayed it will be canceled and the new one takes over.

Parameters

- **disp** – Display object.
- **mode** – One of the MB_DISPLAY_MODE_* options.
- **duration** – Duration how long to show each image (in milliseconds), or SYS_FOREVER_MS.
- **img** – Array of image bitmaps (struct *mb_image* objects).
- **img_count** – Number of images in 'img' array.

void **mb_display_print**(struct mb_display *disp, uint32_t mode, int32_t duration, const char *fmt, ...)

Print a string of characters on the BBC micro:bit LED display.

This function takes a printf-style format string and outputs it in a scrolling fashion to the display.

The call is asynchronous, i.e. the processing of the display happens in the background. If there is another image or string being displayed it will be canceled and the new one takes over.

Parameters

- **disp** – Display object.
- **mode** – One of the MB_DISPLAY_MODE_* options.
- **duration** – Duration how long to show each character (in milliseconds), or SYS_FOREVER_MS.
- **fmt** – printf-style format string
- **...** – Optional list of format arguments.

void **mb_display_stop**(struct mb_display *disp)

Stop the ongoing display of an image.

Parameters

- **disp** – Display object.

struct **mb_image**

#include <mb_display.h> Representation of a BBC micro:bit display image.

This struct should normally not be used directly, rather created using the *MB_IMAGE()* macro.

Related code samples

Character Framebuffer shell module
Use the CFB shell module to interact with a monochrome display.

Character frame buffer
Display character strings using the Character Frame Buffer (CFB).

Custom fonts
Generate and use a custom font.

Monochrome Character Framebuffer

group **monochrome_character_framebuffer**

Public Monochrome Character Framebuffer API.

Defines

FONT_ENTRY_DEFINE(_name, _width, _height, _caps, _data, _fc, _lc)

Macro for creating a font entry.

Parameters

- **_name** – Name of the font entry.
- **_width** – Width of the font in pixels
- **_height** – Height of the font in pixels.
- **_caps** – Font capabilities.
- **_data** – Raw data of the font.
- **_fc** – Character mapped to first font element.
- **_lc** – Character mapped to last font element.

Enums

enum cfb_display_param

Values:

enumerator CFB_DISPLAY_HEIGH = 0

enumerator CFB_DISPLAY_WIDTH

enumerator CFB_DISPLAY_PPT

enumerator CFB_DISPLAY_ROWS

enumerator CFB_DISPLAY_COLS

enum cfb_font_caps

Values:

enumerator CFB_FONT_MONO_VPACKED = *BIT*(0)

enumerator CFB_FONT_MONO_HPACKED = *BIT*(1)

enumerator CFB_FONT_MSB_FIRST = *BIT*(2)

Functions

int cfb_print(const struct *device* *dev, const char *const str, uint16_t x, uint16_t y)

Print a string into the framebuffer.

Parameters

- **dev** – Pointer to device structure for driver instance
- **str** – String to print

- `x` – Position in X direction of the beginning of the string
- `y` – Position in Y direction of the beginning of the string

Returns
0 on success, negative value otherwise

int `cfb_draw_text`(const struct *device* *dev, const char *const str, int16_t x, int16_t y)

Print a string into the framebuffer.

For compare to cfb_print, cfb_draw_text accept non tile-aligned coords and not line wrapping.

Parameters

- `dev` – Pointer to device structure for driver instance
- `str` – String to print
- `x` – Position in X direction of the beginning of the string
- `y` – Position in Y direction of the beginning of the string

Returns
0 on success, negative value otherwise

int `cfb_draw_point`(const struct *device* *dev, const struct *cfb_position* *pos)

Draw a point.

Parameters

- `dev` – Pointer to device structure for driver instance
- `pos` – position of the point

Returns
0 on success, negative value otherwise

int `cfb_draw_line`(const struct *device* *dev, const struct *cfb_position* *start, const struct *cfb_position* *end)

Draw a line.

Parameters

- `dev` – Pointer to device structure for driver instance
- `start` – start position of the line
- `end` – end position of the line

Returns
0 on success, negative value otherwise

int `cfb_draw_rect`(const struct *device* *dev, const struct *cfb_position* *start, const struct *cfb_position* *end)

Draw a rectangle.

Parameters

- `dev` – Pointer to device structure for driver instance
- `start` – Top-Left position of the rectangle
- `end` – Bottom-Right position of the rectangle

Returns
0 on success, negative value otherwise

int `cfb_framebuffer_clear`(const struct *device* *dev, bool clear_display)

Clear framebuffer.

Parameters

- `dev` – Pointer to device structure for driver instance
- `clear_display` – Clear the display as well

Returns

0 on success, negative value otherwise

int `cfb_framebuffer_invert`(const struct *device* *dev)

Invert Pixels.

Parameters

- `dev` – Pointer to device structure for driver instance

Returns

0 on success, negative value otherwise

int `cfb_invert_area`(const struct *device* *dev, uint16_t x, uint16_t y, uint16_t width, uint16_t height)

Invert Pixels in selected area.

Parameters

- `dev` – Pointer to device structure for driver instance
- `x` – Position in X direction of the beginning of area
- `y` – Position in Y direction of the beginning of area
- `width` – Width of area in pixels
- `height` – Height of area in pixels

Returns

0 on success, negative value otherwise

int `cfb_framebuffer_finalize`(const struct *device* *dev)

Finalize framebuffer and write it to display RAM, invert or reorder pixels if necessary.

Parameters

- `dev` – Pointer to device structure for driver instance

Returns

0 on success, negative value otherwise

int `cfb_get_display_parameter`(const struct *device* *dev, enum *cfb_display_param*)

Get display parameter.

Parameters

- `dev` – Pointer to device structure for driver instance
- `cfb_display_param` – One of the display parameters

Returns

Display parameter value

int `cfb_framebuffer_set_font`(const struct *device* *dev, uint8_t idx)

Set font.

Parameters

- `dev` – Pointer to device structure for driver instance
- `idx` – Font index

Returns
0 on success, negative value otherwise

int **cfb_set_kerning**(const struct *device* *dev, int8_t kerning)
Set font kerning (spacing between individual letters).

Parameters

- **dev** – Pointer to device structure for driver instance
- **kerning** – Font kerning

Returns
0 on success, negative value otherwise

int **cfb_get_font_size**(const struct *device* *dev, uint8_t idx, uint8_t *width, uint8_t *height)
Get font size.

Parameters

- **dev** – Pointer to device structure for driver instance
- **idx** – Font index
- **width** – Pointers to the variable where the font width will be stored.
- **height** – Pointers to the variable where the font height will be stored.

Returns
0 on success, negative value otherwise

int **cfb_get_numof_fonts**(const struct *device* *dev)
Get number of fonts.

Parameters

- **dev** – Pointer to device structure for driver instance

Returns
number of fonts

int **cfb_framebuffer_init**(const struct *device* *dev)
Initialize Character Framebuffer.

Parameters

- **dev** – Pointer to device structure for driver instance

Returns
0 on success, negative value otherwise

void **cfb_framebuffer_deinit**(const struct *device* *dev)
Deinitialize Character Framebuffer.

Parameters

- **dev** – Pointer to device structure for driver instance

struct **cfb_font**
#include <cfb.h>

struct **cfb_position**
#include <cfb.h>

7.6.15 Electrically Erasable Programmable Read-Only Memory (EEPROM)

Overview

EEPROMs have an erase block size of 1 byte, a long lifetime, and allow overwriting data on byte-by-byte access.

EEPROM API

Overview The EEPROM API provides read and write access to Electrically Erasable Programmable Read-Only Memory (EEPROM) devices.

Configuration Options Related configuration options:

- `CONFIG_EEPROM`

Related code samples

EEPROM
Store a boot count value in EEPROM.

API Reference

group `eeprom_interface`
EEPROM Interface.

Since
2.1

Version
1.0.0

Functions

int `eeprom_read`(const struct *device* *dev, off_t offset, void *data, size_t len)
Read data from EEPROM.

Parameters

- `dev` – EEPROM device
- `offset` – Address offset to read from.
- `data` – Buffer to store read data.
- `len` – Number of bytes to read.

Returns
0 on success, negative errno code on failure.

int **eeprom_write**(const struct *device* *dev, off_t offset, const void *data, size_t len)

Write data to EEPROM.

Parameters

- **dev** – EEPROM device
- **offset** – Address offset to write data to.
- **data** – Buffer with data to write.
- **len** – Number of bytes to write.

Returns

0 on success, negative errno code on failure.

size_t **eeprom_get_size**(const struct *device* *dev)

Get the size of the EEPROM in bytes.

Parameters

- **dev** – EEPROM device.

Returns

EEPROM size in bytes.

EEPROM Shell

- *Overview*
- *EEPROM Size*
- *Writing Data*
- *Reading Data*

Overview The EEPROM shell provides an `eeprom` command with a set of subcommands for the *shell* module. It allows testing and exploring the *EEPROM* driver API through an interactive interface without having to write a dedicated application. The EEPROM shell can also be enabled in existing applications to aid in interactive debugging of EEPROM issues.

In order to enable the EEPROM shell, the following *Kconfig* options must be enabled:

- `CONFIG_SHELL`
- `CONFIG_EEPROM`
- `CONFIG_EEPROM_SHELL`

For example, building the hello_world sample for the native_sim with the EEPROM shell:

```
# From the root of the zephyr repository
west build -b native_sim samples/hello_world -- -DCONFIG_SHELL=y -DCONFIG_EEPROM=y -DCONFIG_
↪EEPROM_SHELL=y
```

See the *shell* documentation for general instructions on how to connect and interact with the shell. The EEPROM shell comes with built-in help (unless `CONFIG_SHELL_HELP` is disabled). The built-in help messages can be printed by passing `-h` or `--help` to the `eeprom` command or any of its subcommands. All subcommands also support tab-completion of their arguments.

> **Tip**
>
> All of the EEPROM shell subcommands take the name of an EEPROM peripheral as their first argument, which also supports tab-completion. A list of all devices available can be obtained using the `device list` shell command when `CONFIG_DEVICE_SHELL` is enabled. The examples below all use the device name `eeprom@0`.

EEPROM Size The size of an EEPROM can be inspected using the `eeprom size` subcommand as shown below:

```
uart:~$ eeprom size eeprom@0
32768 bytes
```

Writing Data Data can be written to an EEPROM using the `eeprom write` subcommand. This subcommand takes at least three arguments; the EEPROM device name, the offset to start writing to, and at least one data byte. In the following example, the hexadecimal sequence of bytes `0x0d 0x0e 0x0a 0x0d 0x0b 0x0e 0x0e 0x0f` is written to offset `0x0`:

```
uart:~$ eeprom write eeprom@0 0x0 0x0d 0x0e 0x0a 0x0d 0x0b 0x0e 0x0e 0x0f
Writing 8 bytes to EEPROM...
Verifying...
Verify OK
```

It is also possible to fill a portion of the EEPROM with the same pattern using the `eeprom fill` subcommand. In the following example, the pattern `0xaa` is written to 16 bytes starting at offset `0x8`:

```
uart:~$ eeprom fill eeprom@0 0x8 16 0xaa
Writing 16 bytes of 0xaa to EEPROM...
Verifying...
Verify OK
```

Reading Data Data can be read from an EEPROM using the `eeprom read` subcommand. This subcommand takes three arguments; the EEPROM device name, the offset to start reading from, and the number of bytes to read:

```
uart:~$ eeprom read eeprom@0 0x0 8
Reading 8 bytes from EEPROM, offset 0...
00000000: 0d 0e 0a 0d 0b 0e 0e 0f                         |........        |
```

7.6.16 Enhanced Serial Peripheral Interface (eSPI) Bus

Overview

The eSPI (enhanced serial peripheral interface) is a serial bus that is based on SPI. It also features a four-wire interface (receive, transmit, clock and target select) and three configurations: single IO, dual IO and quad IO.

The technical advancements include lower voltage signal levels (1.8V vs. 3.3V), lower pin count, and the frequency is twice as fast (66MHz vs. 33MHz) Because of its enhancements, the eSPI is used to replace the LPC (lower pin count) interface, SPI, SMBus and sideband signals.

See eSPI interface specification for additional details.

API Reference

Related code samples

Enhanced Serial Peripheral Interface (eSPI)
Use eSPI to connect to a slave device and exchange virtual wire packets.

group `espi_interface`

eSPI Driver APIs

eSPI SAF Driver APIs

Defines

`ESPI_VWIRE_SIGNAL_OCB_0`

`ESPI_VWIRE_SIGNAL_OCB_1`

`ESPI_VWIRE_SIGNAL_OCB_2`

`ESPI_VWIRE_SIGNAL_OCB_3`

`HOST_KBC_EVT_IBF`

`HOST_KBC_EVT_OBE`

Typedefs

typedef void (*`espi_callback_handler_t`)(const struct *device* *dev, struct espi_callback *cb, struct *espi_event* espi_evt)

Define the application callback handler function signature.

Param dev
Device struct for the eSPI device.

Param cb
Original struct espi_callback owning this handler.

Param espi_evt
event details that trigger the callback handler.

Enums

enum `espi_io_mode`

eSPI I/O mode capabilities

Values:

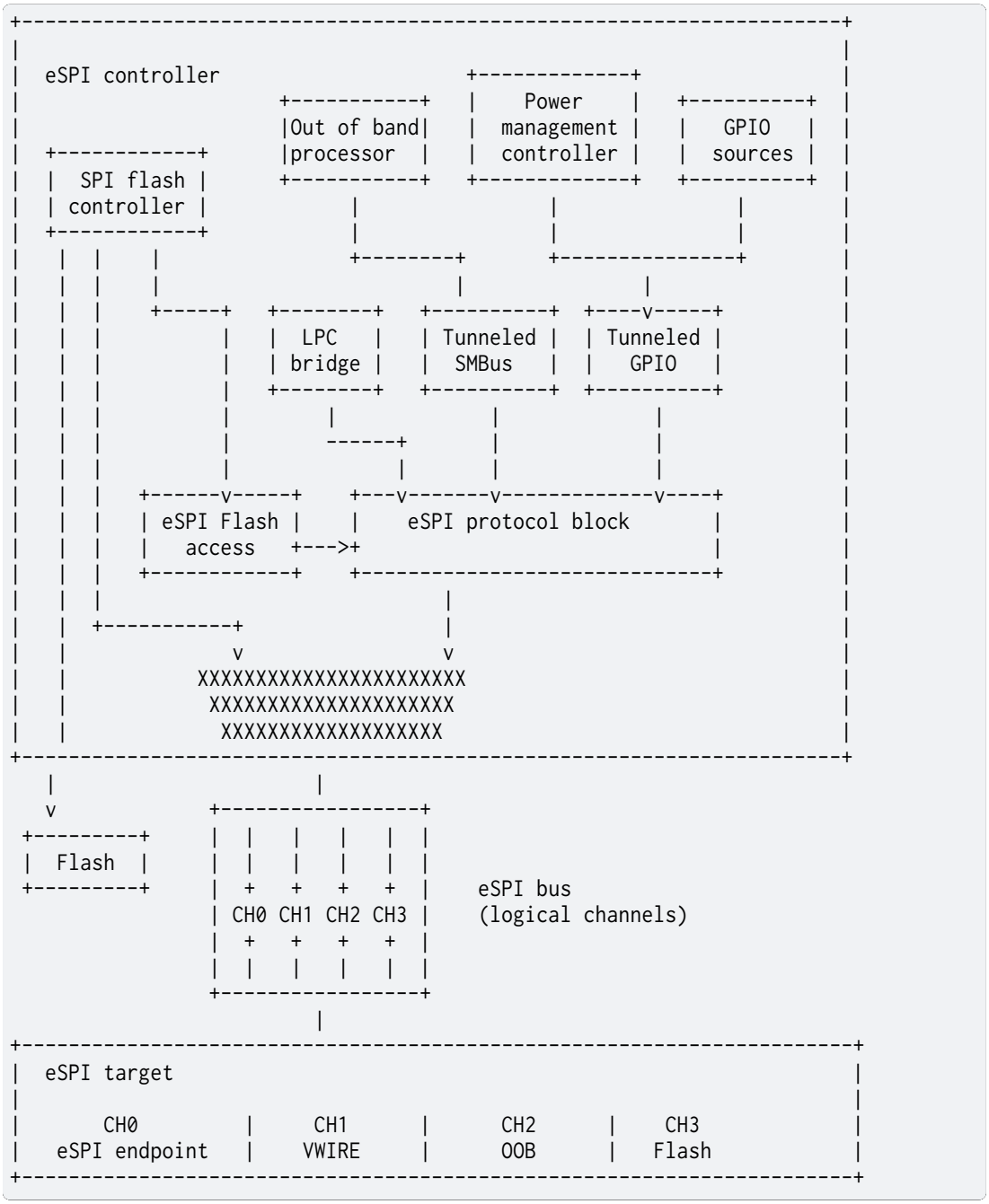
enumerator **ESPI_IO_MODE_SINGLE_LINE** = *BIT*(0)

enumerator **ESPI_IO_MODE_DUAL_LINES** = *BIT*(1)

enumerator **ESPI_IO_MODE_QUAD_LINES** = *BIT*(2)

enum **espi_channel**

eSPI channel.



Identifies each eSPI logical channel supported by eSPI controller Each channel allows independent traffic, but the assignment of channel type to channel number is fixed.

Note that generic commands are not associated with any channel, so traffic over eSPI can occur if all channels are disabled or not ready

Values:

enumerator **ESPI_CHANNEL_PERIPHERAL** = *BIT*(0)

enumerator **ESPI_CHANNEL_VWIRE** = *BIT*(1)

enumerator **ESPI_CHANNEL_OOB** = *BIT*(2)

enumerator **ESPI_CHANNEL_FLASH** = *BIT*(3)

enum **espi_bus_event**

eSPI bus event.

eSPI bus event to indicate events for which user can register callbacks

Values:

enumerator **ESPI_BUS_RESET** = *BIT*(0)

Indicates the eSPI bus was reset either via eSPI reset pin.

eSPI drivers should convey the eSPI reset status to eSPI driver clients following eSPI specification reset pin convention: 0-eSPI bus in reset, 1-eSPI bus out-of-reset

Note: There is no need to send this callback for in-band reset.

enumerator **ESPI_BUS_EVENT_CHANNEL_READY** = *BIT*(1)

Indicates the eSPI HW has received channel enable notification from eSPI host, once the eSPI channel is signal as ready to the eSPI host, eSPI drivers should convey the eSPI channel ready to eSPI driver client via this event.

enumerator **ESPI_BUS_EVENT_VWIRE_RECEIVED** = *BIT*(2)

Indicates the eSPI HW has received a virtual wire message from eSPI host.

eSPI drivers should convey the eSPI virtual wire latest status.

enumerator **ESPI_BUS_EVENT_OOB_RECEIVED** = *BIT*(3)

Indicates the eSPI HW has received a Out-of-band package from eSPI host.

enumerator **ESPI_BUS_PERIPHERAL_NOTIFICATION** = *BIT*(4)

Indicates the eSPI HW has received a peripheral eSPI host event.

eSPI drivers should convey the peripheral type.

enumerator **ESPI_BUS_TAF_NOTIFICATION** = *BIT*(5)

enum **espi_pc_event**

eSPI peripheral channel events.

eSPI peripheral channel event types to indicate users.

Values:

enumerator **ESPI_PC_EVT_BUS_CHANNEL_READY** = *BIT*(0)

enumerator ESPI_PC_EVT_BUS_MASTER_ENABLE = *BIT*(1)

enum espi_virtual_peripheral

eSPI peripheral notification type.

eSPI peripheral notification event details to indicate which peripheral trigger the eSPI callback

Values:

enumerator ESPI_PERIPHERAL_UART

enumerator ESPI_PERIPHERAL_8042_KBC

enumerator ESPI_PERIPHERAL_HOST_IO

enumerator ESPI_PERIPHERAL_DEBUG_PORT80

enumerator ESPI_PERIPHERAL_HOST_IO_PVT

enum espi_cycle_type

eSPI cycle types supported over eSPI peripheral channel

Values:

enumerator ESPI_CYCLE_MEMORY_READ32

enumerator ESPI_CYCLE_MEMORY_READ64

enumerator ESPI_CYCLE_MEMORY_WRITE32

enumerator ESPI_CYCLE_MEMORY_WRITE64

enumerator ESPI_CYCLE_MESSAGE_NODATA

enumerator ESPI_CYCLE_MESSAGE_DATA

enumerator ESPI_CYCLE_OK_COMPLETION_NODATA

enumerator ESPI_CYCLE_OKCOMPLETION_DATA

enumerator ESPI_CYCLE_NOK_COMPLETION_NODATA

enum espi_vwire_signal

eSPI system platform signals that can be send or receive through virtual wire channel

Values:

enumerator ESPI_VWIRE_SIGNAL_SLP_S3

enumerator ESPI_VWIRE_SIGNAL_SLP_S4

enumerator ESPI_VWIRE_SIGNAL_SLP_S5

enumerator ESPI_VWIRE_SIGNAL_OOB_RST_WARN

enumerator ESPI_VWIRE_SIGNAL_PLTRST

enumerator ESPI_VWIRE_SIGNAL_SUS_STAT

enumerator ESPI_VWIRE_SIGNAL_NMIOUT

enumerator ESPI_VWIRE_SIGNAL_SMIOUT

enumerator ESPI_VWIRE_SIGNAL_HOST_RST_WARN

enumerator ESPI_VWIRE_SIGNAL_SLP_A

enumerator ESPI_VWIRE_SIGNAL_SUS_PWRDN_ACK

enumerator ESPI_VWIRE_SIGNAL_SUS_WARN

enumerator ESPI_VWIRE_SIGNAL_SLP_WLAN

enumerator ESPI_VWIRE_SIGNAL_SLP_LAN

enumerator ESPI_VWIRE_SIGNAL_HOST_C10

enumerator ESPI_VWIRE_SIGNAL_DNX_WARN

enumerator ESPI_VWIRE_SIGNAL_PME

enumerator ESPI_VWIRE_SIGNAL_WAKE

enumerator ESPI_VWIRE_SIGNAL_OOB_RST_ACK

enumerator ESPI_VWIRE_SIGNAL_TARGET_BOOT_STS

enumerator ESPI_VWIRE_SIGNAL_ERR_NON_FATAL

enumerator ESPI_VWIRE_SIGNAL_ERR_FATAL

enumerator ESPI_VWIRE_SIGNAL_TARGET_BOOT_DONE

enumerator ESPI_VWIRE_SIGNAL_HOST_RST_ACK

enumerator ESPI_VWIRE_SIGNAL_RST_CPU_INIT

enumerator ESPI_VWIRE_SIGNAL_SMI

enumerator ESPI_VWIRE_SIGNAL_SCI

enumerator ESPI_VWIRE_SIGNAL_DNX_ACK

enumerator ESPI_VWIRE_SIGNAL_SUS_ACK

enumerator ESPI_VWIRE_SIGNAL_TARGET_GPIO_0

enumerator ESPI_VWIRE_SIGNAL_TARGET_GPIO_1

enumerator ESPI_VWIRE_SIGNAL_TARGET_GPIO_2

enumerator ESPI_VWIRE_SIGNAL_TARGET_GPIO_3

enumerator ESPI_VWIRE_SIGNAL_TARGET_GPIO_4

enumerator ESPI_VWIRE_SIGNAL_TARGET_GPIO_5

enumerator ESPI_VWIRE_SIGNAL_TARGET_GPIO_6

enumerator ESPI_VWIRE_SIGNAL_TARGET_GPIO_7

enumerator ESPI_VWIRE_SIGNAL_TARGET_GPIO_8

enumerator ESPI_VWIRE_SIGNAL_TARGET_GPIO_9

enumerator ESPI_VWIRE_SIGNAL_TARGET_GPIO_10

enumerator ESPI_VWIRE_SIGNAL_TARGET_GPIO_11

enumerator ESPI_VWIRE_SIGNAL_COUNT

enum lpc_peripheral_opcode

Values:

enumerator E8042_OBF_HAS_CHAR = 0x50

enumerator E8042_IBF_HAS_CHAR

enumerator E8042_WRITE_KB_CHAR

enumerator **E8042_WRITE_MB_CHAR**

enumerator **E8042_RESUME_IRQ**

enumerator **E8042_PAUSE_IRQ**

enumerator **E8042_CLEAR_OBF**

enumerator **E8042_READ_KB_STS**

enumerator **E8042_SET_FLAG**

enumerator **E8042_CLEAR_FLAG**

enumerator **EACPI_OBF_HAS_CHAR** = EACPI_START_OPCODE

enumerator **EACPI_IBF_HAS_CHAR**

enumerator **EACPI_WRITE_CHAR**

enumerator **EACPI_READ_STS**

enumerator **EACPI_WRITE_STS**

Functions

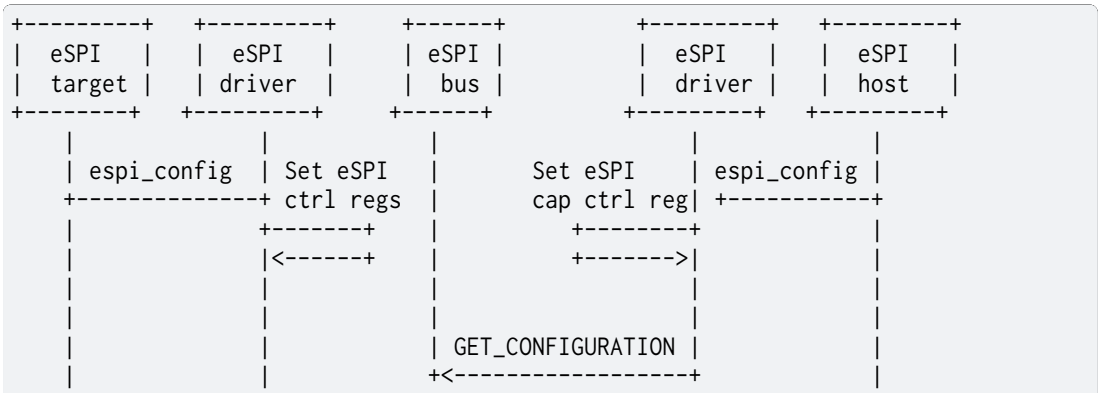
int **espi_config**(const struct *device* *dev, struct *espi_cfg* *cfg)

Configure operation of a eSPI controller.

This routine provides a generic interface to override eSPI controller capabilities.

If this eSPI controller is acting as target, the values set here will be discovered as part through the GET_CONFIGURATION command issued by the eSPI controller during initialization.

If this eSPI controller is acting as controller, the values set here will be used by eSPI controller to determine minimum common capabilities with eSPI target then send via SET_CONFIGURATION command.



(continues on next page)

(continued from previous page)

```
|              |<-----------|                  |             |
|              | eSPI caps  |                  |             |
|              |----------->+    response       |             |
|              |            |----------------->+             |
|              |            |                  |             |
|              |            | SET_CONFIGURATION |             |
|              |            +<-----------------+             |
|              |            |  accept          |             |
|              |            +----------------->+             |
|              |            |                  |             |
+              +            +                  +             +
```

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `cfg` – the device runtime configuration for the eSPI controller.

Return values

- `0` – If successful.
- `-EIO` – General input / output error, failed to configure device.
- `-EINVAL` – invalid capabilities, failed to configure device.
- `-ENOTSUP` – capability not supported by eSPI target.

bool **espi_get_channel_status**(const struct *device* *dev, enum *espi_channel* ch)

Query to see if it a channel is ready.

This routine allows to check if logical channel is ready before use. Note that queries for channels not supported will always return false.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `ch` – the eSPI channel for which status is to be retrieved.

Return values

- `true` – If eSPI channel is ready.
- `false` – otherwise.

int **espi_read_request**(const struct *device* *dev, struct *espi_request_packet* *req)

Sends memory, I/O or message read request over eSPI.

This routines provides a generic interface to send a read request packet.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `req` – Address of structure representing a memory, I/O or message read request.

Return values

- `0` – If successful.
- `-ENOTSUP` – if eSPI controller doesn't support raw packets and instead low memory transactions are handled by controller hardware directly.
- `-EIO` – General input / output error, failed to send over the bus.

int **espi_write_request**(const struct *device* *dev, struct *espi_request_packet* *req)

Sends memory, I/O or message write request over eSPI.

This routines provides a generic interface to send a write request packet.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `req` – Address of structure representing a memory, I/O or message write request.

Return values

- `0` – If successful.
- `-ENOTSUP` – if eSPI controller doesn't support raw packets and instead low memory transactions are handled by controller hardware directly.
- `-EINVAL` – General input / output error, failed to send over the bus.

int **espi_read_lpc_request**(const struct *device* *dev, enum *lpc_peripheral_opcode* op, uint32_t *data)

Reads SOC data from a LPC peripheral with information updated over eSPI.

This routine provides a generic interface to read a block whose information was updated by an eSPI transaction. Reading may trigger a transaction. The eSPI packet is assembled by the HW block.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `op` – Enum representing opcode for peripheral type and read request.
- `data` – Parameter to be read from to the LPC peripheral.

Return values

- `0` – If successful.
- `-ENOTSUP` – if eSPI peripheral is off or not supported.
- `-EINVAL` – for unimplemented lpc opcode, but in range.

int **espi_write_lpc_request**(const struct *device* *dev, enum *lpc_peripheral_opcode* op, uint32_t *data)

Writes data to a LPC peripheral which generates an eSPI transaction.

This routine provides a generic interface to write data to a block which triggers an eSPI transaction. The eSPI packet is assembled by the HW block.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `op` – Enum representing an opcode for peripheral type and write request.
- `data` – Represents the parameter passed to the LPC peripheral.

Return values

- `0` – If successful.
- `-ENOTSUP` – if eSPI peripheral is off or not supported.
- `-EINVAL` – for unimplemented lpc opcode, but in range.

int **espi_send_vwire**(const struct *device* *dev, enum *espi_vwire_signal* signal, uint8_t level)

Sends system/platform signal as a virtual wire packet.

This routines provides a generic interface to send a virtual wire packet from target to controller.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `signal` – The signal to be send to eSPI controller.
- `level` – The level of signal requested LOW or HIGH.

Return values

- `0` – If successful.
- `-EIO` – General input / output error, failed to send over the bus.

int **espi_receive_vwire**(const struct *device* *dev, enum *espi_vwire_signal* signal, uint8_t *level)

Retrieves level status for a signal encapsulated in a virtual wire.

This routines provides a generic interface to request a virtual wire packet from eSPI controller and retrieve the signal level.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `signal` – the signal to be requested from eSPI controller.
- `level` – the level of signal requested 0b LOW, 1b HIGH.

Return values

`-EIO` – General input / output error, failed request to controller.

int **espi_send_oob**(const struct *device* *dev, struct *espi_oob_packet* *pckt)

Sends SMBus transaction (out-of-band) packet over eSPI bus.

This routines provides an interface to encapsulate a SMBus transaction and send into packet over eSPI bus

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `pckt` – Address of the packet representation of SMBus transaction.

Return values

`-EIO` – General input / output error, failed request to controller.

int **espi_receive_oob**(const struct *device* *dev, struct *espi_oob_packet* *pckt)

Receives SMBus transaction (out-of-band) packet from eSPI bus.

This routines provides an interface to receive and decoded a SMBus transaction from eSPI bus

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `pckt` – Address of the packet representation of SMBus transaction.

Return values

`-EIO` – General input / output error, failed request to controller.

int **espi_read_flash**(const struct *device* *dev, struct *espi_flash_packet* *pckt)

Sends a read request packet for shared flash.

This routines provides an interface to send a request to read the flash component shared between the eSPI controller and eSPI targets.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **pckt** – Address of the representation of read flash transaction.

Return values

- **-ENOTSUP** – eSPI flash logical channel transactions not supported.
- **-EBUSY** – eSPI flash channel is not ready or disabled by controller.
- **-EIO** – General input / output error, failed request to controller.

int **espi_write_flash**(const struct *device* *dev, struct *espi_flash_packet* *pckt)

Sends a write request packet for shared flash.

This routines provides an interface to send a request to write to the flash components shared between the eSPI controller and eSPI targets.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **pckt** – Address of the representation of write flash transaction.

Return values

- **-ENOTSUP** – eSPI flash logical channel transactions not supported.
- **-EBUSY** – eSPI flash channel is not ready or disabled by controller.
- **-EIO** – General input / output error, failed request to controller.

int **espi_flash_erase**(const struct *device* *dev, struct *espi_flash_packet* *pckt)

Sends a write request packet for shared flash.

This routines provides an interface to send a request to write to the flash components shared between the eSPI controller and eSPI targets.

Parameters

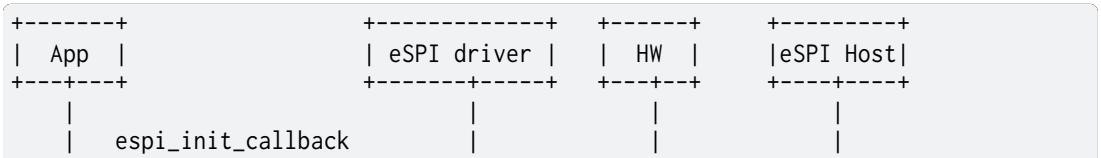
- **dev** – Pointer to the device structure for the driver instance.
- **pckt** – Address of the representation of write flash transaction.

Return values

- **-ENOTSUP** – eSPI flash logical channel transactions not supported.
- **-EBUSY** – eSPI flash channel is not ready or disabled by controller.
- **-EIO** – General input / output error, failed request to controller.

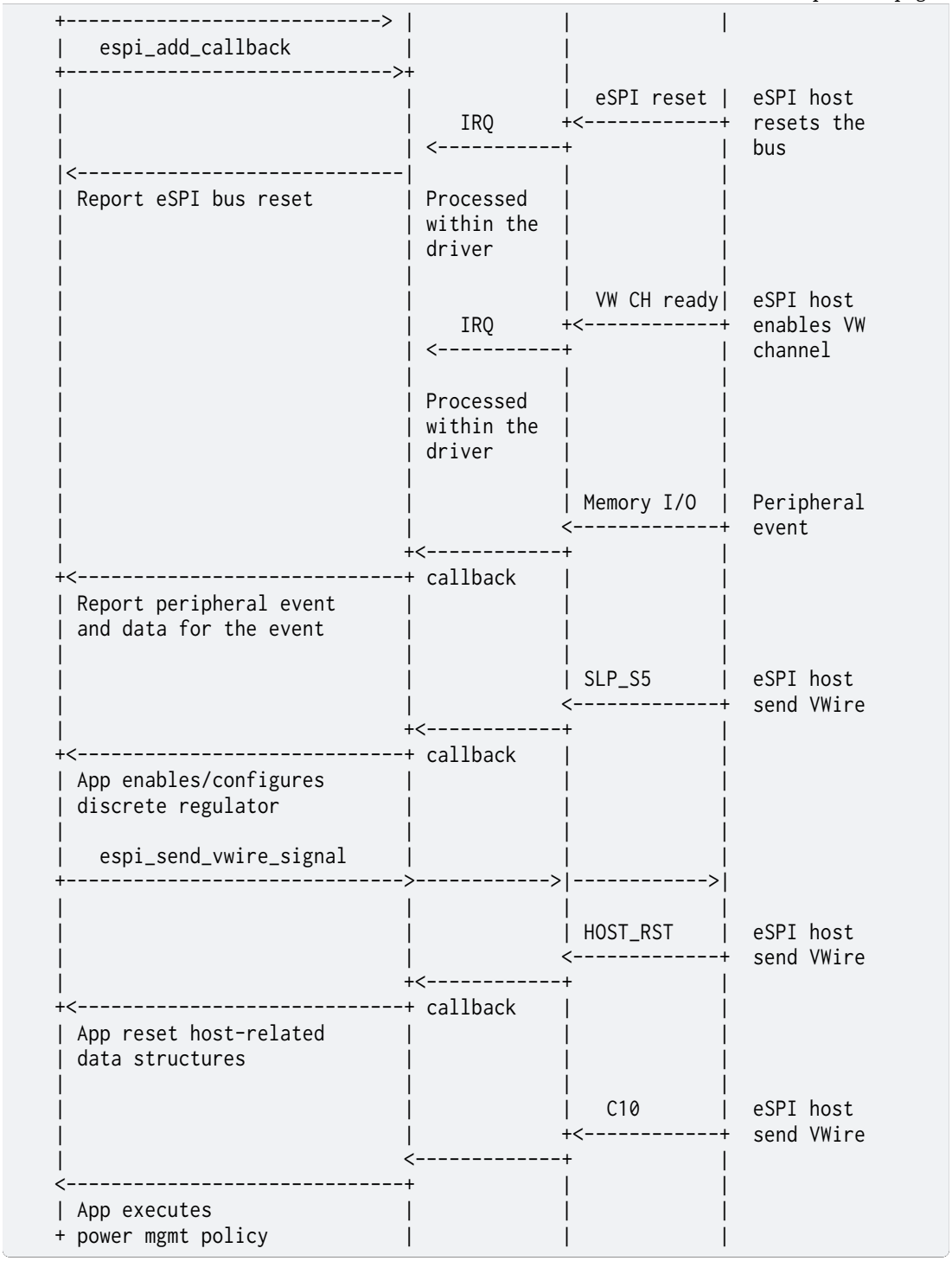
static inline void **espi_init_callback**(struct espi_callback *callback, *espi_callback_handler_t* handler, enum *espi_bus_event* evt_type)

Callback model.



(continues on next page)

(continued from previous page)



Helper to initialize a struct espi_callback properly.

Parameters

- **callback** – A valid Application's callback structure pointer.
- **handler** – A valid handler function pointer.
- **evt_type** – indicates the eSPI event relevant for the handler. for VWIRE_RECEIVED event the data will indicate the new level asserted

static inline int **espi_add_callback**(const struct *device* *dev, struct espi_callback *callback)

Add an application callback.

Note: enables to add as many callback as needed on the same device.

> **Note**
>
> Callbacks may be added to the device from within a callback handler invocation, but whether they are invoked for the current eSPI event is not specified.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `callback` – A valid Application's callback structure pointer.

Returns

0 if successful, negative errno code on failure.

static inline int **espi_remove_callback**(const struct *device* *dev, struct espi_callback *callback)

Remove an application callback.

Note: enables to remove as many callbacks as added through *espi_add_callback()*.

> **Warning**
>
> It is explicitly permitted, within a callback handler, to remove the registration for the callback that is running, i.e. `callback`. Attempts to remove other registrations on the same device may result in undefined behavior, including failure to invoke callbacks that remain registered and unintended invocation of removed callbacks.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `callback` – A valid application's callback structure pointer.

Returns

0 if successful, negative errno code on failure.

int **espi_saf_config**(const struct *device* *dev, const struct *espi_saf_cfg* *cfg)

Configure operation of a eSPI controller.

This routine provides a generic interface to override eSPI controller capabilities.

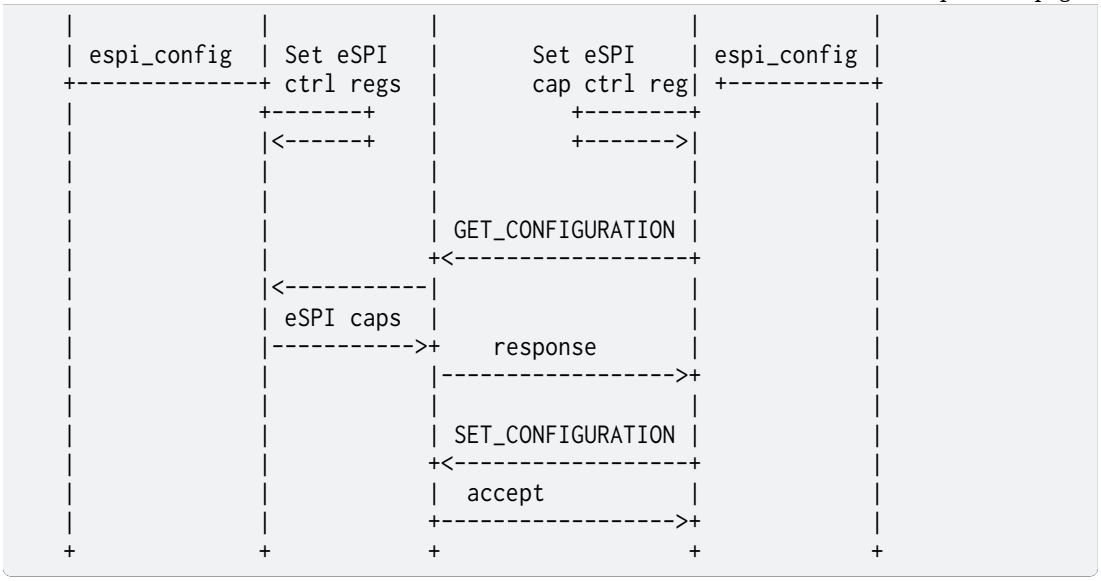
If this eSPI controller is acting as slave, the values set here will be discovered as part through the GET_CONFIGURATION command issued by the eSPI master during initialization.

If this eSPI controller is acting as master, the values set here will be used by eSPI master to determine minimum common capabilities with eSPI slave then send via SET_CONFIGURATION command.

```
+--------+   +---------+     +------+          +---------+   +---------+
|  eSPI  |   |  eSPI   |     | eSPI |          |  eSPI   |   |  eSPI   |
|  slave |   | driver  |     |  bus |          |  driver |   |  host   |
+--------+   +---------+     +------+          +---------+   +---------+
```

(continues on next page)

(continued from previous page)



Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `cfg` – the device runtime configuration for the eSPI controller.

Return values

- `0` – If successful.
- `-EIO` – General input / output error, failed to configure device.
- `-EINVAL` – invalid capabilities, failed to configure device.
- `-ENOTSUP` – capability not supported by eSPI slave.

int **espi_saf_set_protection_regions**(const struct *device* *dev, const struct espi_saf_protection *pr)

Set one or more SAF protection regions.

This routine provides an interface to override the default flash protection regions of the SAF controller.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `pr` – Pointer to the SAF protection region structure.

Return values

- `0` – If successful.
- `-EIO` – General input / output error, failed to configure device.
- `-EINVAL` – invalid capabilities, failed to configure device.
- `-ENOTSUP` – capability not supported by eSPI slave.

int **espi_saf_activate**(const struct *device* *dev)

Activate SAF block.

This routine activates the SAF block and should only be called after SAF has been configured and the eSPI Master has enabled the Flash Channel.

Parameters

- **dev** – Pointer to the device structure for the driver instance.

Return values

- **0** – If successful
- **-EINVAL** – if failed to activate SAF.

bool **espi_saf_get_channel_status**(const struct *device* *dev)

Query to see if SAF is ready.

This routine allows to check if SAF is ready before use.

Parameters

- **dev** – Pointer to the device structure for the driver instance.

Return values

- **true** – If eSPI SAF is ready.
- **false** – otherwise.

int **espi_saf_flash_read**(const struct *device* *dev, struct *espi_saf_packet* *pckt)

Sends a read request packet for slave attached flash.

This routines provides an interface to send a request to read the flash component shared between the eSPI master and eSPI slaves.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **pckt** – Address of the representation of read flash transaction.

Return values

- **-ENOTSUP** – eSPI flash logical channel transactions not supported.
- **-EBUSY** – eSPI flash channel is not ready or disabled by master.
- **-EIO** – General input / output error, failed request to master.

int **espi_saf_flash_write**(const struct *device* *dev, struct *espi_saf_packet* *pckt)

Sends a write request packet for slave attached flash.

This routines provides an interface to send a request to write to the flash components shared between the eSPI master and eSPI slaves.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **pckt** – Address of the representation of write flash transaction.

Return values

- **-ENOTSUP** – eSPI flash logical channel transactions not supported.
- **-EBUSY** – eSPI flash channel is not ready or disabled by master.
- **-EIO** – General input / output error, failed request to master.

int **espi_saf_flash_erase**(const struct *device* *dev, struct *espi_saf_packet* *pckt)

Sends a write request packet for slave attached flash.

This routines provides an interface to send a request to write to the flash components shared between the eSPI master and eSPI slaves.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **pckt** – Address of the representation of erase flash transaction.

Return values

- `-ENOTSUP` – eSPI flash logical channel transactions not supported.
- `-EBUSY` – eSPI flash channel is not ready or disabled by master.
- `-EIO` – General input / output error, failed request to master.

int **espi_saf_flash_unsuccess**(const struct *device* *dev, struct *espi_saf_packet* *pckt)

Response unsuccessful completion for slave attached flash.

This routines provides an interface to response that transaction is invalid and return unsuccessful completion from target to controller.

Parameters

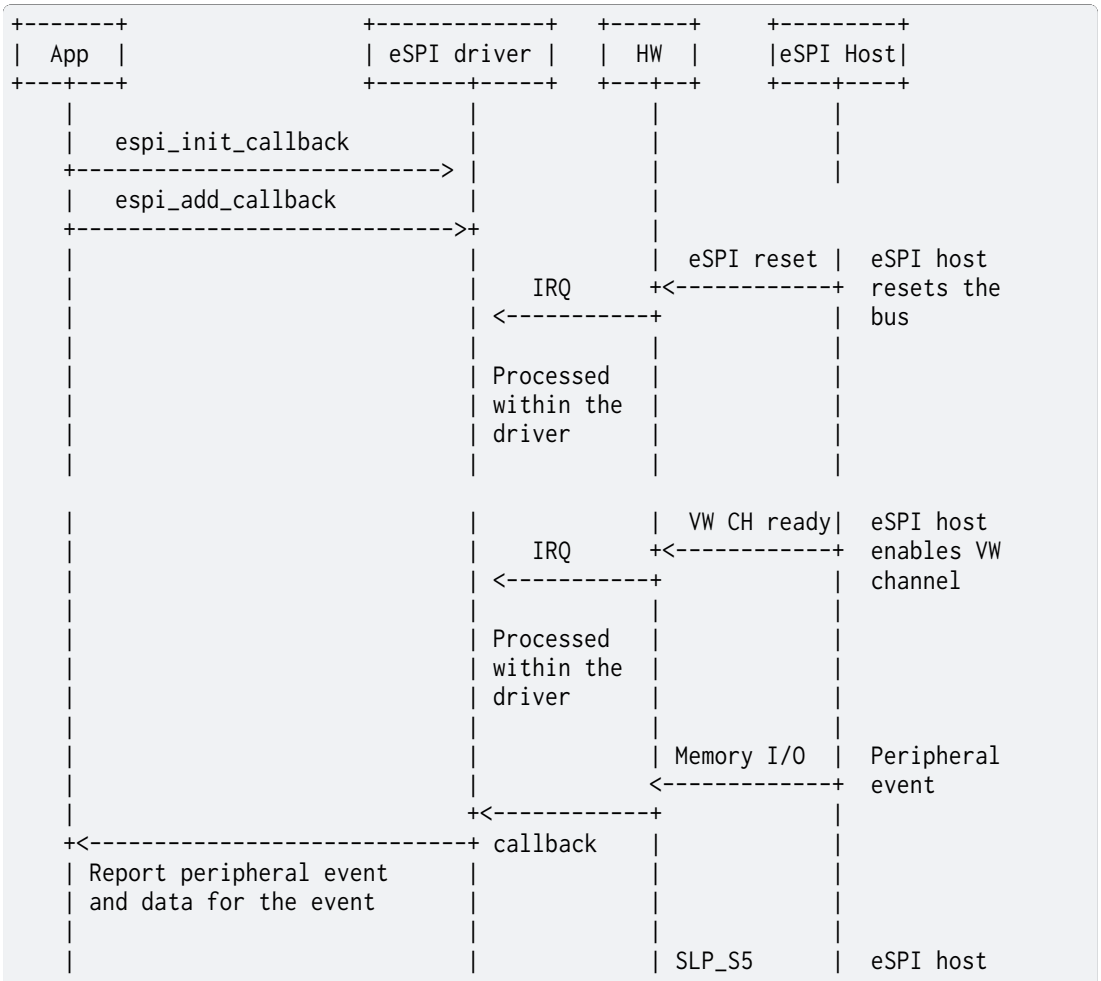
- `dev` – Pointer to the device structure for the driver instance.
- `pckt` – Address of the representation of flash transaction.

Return values

- `-ENOTSUP` – eSPI flash logical channel transactions not supported.
- `-EBUSY` – eSPI flash channel is not ready or disabled by master.
- `-EIO` – General input / output error, failed request to master.

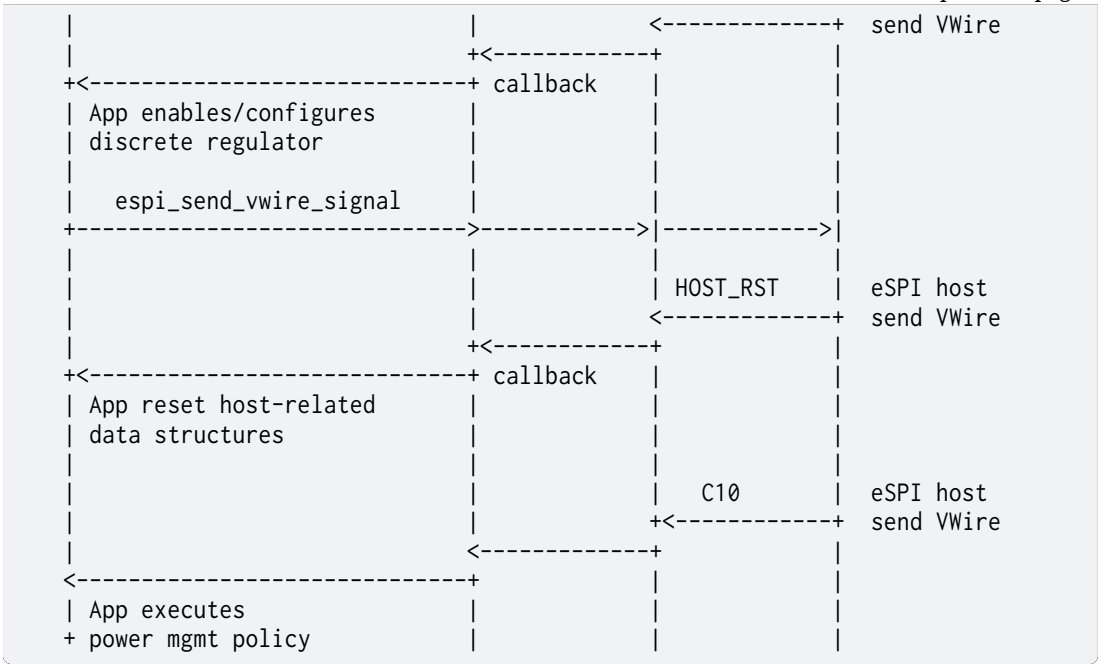
static inline void **espi_saf_init_callback**(struct espi_callback *callback, *espi_callback_handler_t* handler, enum *espi_bus_event* evt_type)

Callback model.



(continues on next page)

(continued from previous page)



Helper to initialize a struct espi_callback properly.

Parameters

- **callback** – A valid Application's callback structure pointer.
- **handler** – A valid handler function pointer.
- **evt_type** – indicates the eSPI event relevant for the handler. for VWIRE_RECEIVED event the data will indicate the new level asserted

static inline int **espi_saf_add_callback**(const struct *device* *dev, struct espi_callback *callback)

Add an application callback.

Note: enables to add as many callback as needed on the same device.

Note

Callbacks may be added to the device from within a callback handler invocation, but whether they are invoked for the current eSPI event is not specified.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **callback** – A valid Application's callback structure pointer.

Returns

0 if successful, negative errno code on failure.

static inline int **espi_saf_remove_callback**(const struct *device* *dev, struct espi_callback *callback)

Remove an application callback.

Note: enables to remove as many callbacks as added through *espi_add_callback()*.

Warning

It is explicitly permitted, within a callback handler, to remove the registration for the callback that is running, i.e. `callback`. Attempts to remove other registrations on the same device may result in undefined behavior, including failure to invoke callbacks that remain registered and unintended invocation of removed callbacks.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `callback` – A valid application's callback structure pointer.

Returns

0 if successful, negative errno code on failure.

struct `espi_evt_data_kbc`

#include <*espi.h*> Bit field definition of evt_data in struct *espi_event* for KBC.

struct `espi_evt_data_acpi`

#include <*espi.h*> Bit field definition of evt_data in struct *espi_event* for ACPI.

struct `espi_event`

#include <*espi.h*> eSPI event

Public Members

enum *espi_bus_event* `evt_type`

Event type.

uint32_t `evt_details`

Additional details for bus event type.

uint32_t `evt_data`

Data associated to the event.

struct `espi_cfg`

#include <*espi.h*> eSPI bus configuration parameters

Public Members

enum *espi_io_mode* `io_caps`

Supported I/O mode.

enum *espi_channel* `channel_caps`

Supported channels.

uint8_t `max_freq`

Maximum supported frequency in MHz.

struct **espi_request_packet**

#include <*espi.h*> eSPI peripheral request packet format

struct **espi_oob_packet**

#include <*espi.h*> eSPI out-of-band transaction packet format

For Tx packet, eSPI driver client shall specify the OOB payload data and its length in bytes. For Rx packet, eSPI driver client shall indicate the maximum number of bytes that can receive, while the eSPI driver should update the length field with the actual data received/available.

In all cases, the length does not include OOB header size 3 bytes.

struct **espi_flash_packet**

#include <*espi.h*> eSPI flash transactions packet format

struct **espi_saf_cfg**

#include <*espi_saf.h*> eSPI SAF configuration parameters

struct **espi_saf_packet**

#include <*espi_saf.h*> eSPI SAF transaction packet format

7.6.17 Entropy

Overview

The entropy API provides functions to retrieve entropy values from entropy hardware present on the platform. The entropy APIs are provided for use by the random subsystem and cryptographic services. They are not suitable to be used as random number generation functions.

API Reference

group **entropy_interface**

Entropy Interface.

Since
1.10

Version
1.0.0

Defines

ENTROPY_BUSYWAIT

Driver is allowed to busy-wait for random data to be ready.

Typedefs

typedef int (*`entropy_get_entropy_t`)(const struct *device* *dev, uint8_t *buffer, uint16_t length)

Callback API to get entropy.

See *entropy_get_entropy()* for argument description

> **Note**
>
> This call has to be thread safe to satisfy requirements of the random subsystem.

typedef int (*`entropy_get_entropy_isr_t`)(const struct *device* *dev, uint8_t *buffer, uint16_t length, uint32_t flags)

Callback API to get entropy from an ISR.

See *entropy_get_entropy_isr()* for argument description

Functions

int `entropy_get_entropy`(const struct *device* *dev, uint8_t *buffer, uint16_t length)

Fills a buffer with entropy.

Blocks if required in order to generate the necessary random data.

Parameters

- `dev` – Pointer to the entropy device.
- `buffer` – Buffer to fill with entropy.
- `length` – Buffer length.

Return values

- `0` – on success.
- `-ERRNO` – errno code on error.

static inline int `entropy_get_entropy_isr`(const struct *device* *dev, uint8_t *buffer, uint16_t length, uint32_t flags)

Fills a buffer with entropy in a non-blocking or busy-wait manner.

Callable from ISRs.

Parameters

- `dev` – Pointer to the device structure.
- `buffer` – Buffer to fill with entropy.
- `length` – Buffer length.
- `flags` – Flags to modify the behavior of the call.

Return values

`number` – of bytes filled with entropy or -error.

struct `entropy_driver_api`

#include <entropy.h> Entropy driver API structure.

This is the mandatory API any Entropy driver needs to expose.

7.6.18 Error Detection And Correction (EDAC)

Error Detection And Correction is a mechanism used to detect and correct errors while storing or reading data.

In Band Error Correction Code (IBECC)

Overview The mechanism initially found in Intel Elkhart Lake SOCs and later boards is an integrated memory controller with IBECC.

The In-Band Error Correction Code (IBECC) improves reliability by providing error detection and correction. IBECC can work for all or for specific regions of physical memory space. The IBECC is useful for memory technologies that do not support the out-of-band ECC.

IBECC adds memory overhead of 1/32 of the memory. This memory is not accessible and used to store ECC syndrome data. IBECC converts read / write transactions to two separate transactions: one for actual data and another for cache line containing ECC value.

There is a debug feature IBECC Error Injection which helps to debug and verify IBECC functionality. ECC errors are injected on the write path and cause ECC errors on the read path.

IBECC Configuration There are three IBECC operation modes which can be selected by Bootloader. They are listed below:

- OPERATION_MODE = 0x0 sets functional mode to protect requests based on address range
- OPERATION_MODE = 0x1 sets functional mode to all requests not be protected and to ignore range checks
- OPERATION_MODE = 0x2 sets functional mode to protect all requests and ignore range checks

IBECC operational mode is configured through BIOS or Bootloader. For operation mode 0 there are more BIOS configuration options such as memory regions.

Due to high security risk Error Injection capability should not be enabled for production. Error Injection is only enabled for tests.

IBECC Logging IBECC logs the following fields:

- Error Address
- Error Syndrome
- Error Type
  - Correctable Error (CE) - error is detected and corrected by IBECC module.
  - Uncorrectable Error (UE) - error is detected by IBECC module and not automatically corrected.

The IBECC driver provides error type for the higher-level application to implement desired policy with respect for handling those memory errors. Error syndrome is not used in the IBECC driver but provided to higher-level application.

Usage notes Exceptional care needs to be taken with Non Maskable Interrupt (NMI). NMI will arrive at any time, even if the local CPU has disabled interrupts. That means that no locking mechanism can protect code against an NMI happening. Zephyr's IPC mechanisms universally use local IRQ locking as the base layer for all higher-level synchronization primitives. So, you cannot share anything that is "protected" by a lock with an NMI, because the protection does not work. The only tool you have available for synchronization in the Zephyr API that works against an NMI is the atomic layer. This also applies to callback function which is called by NMI handler.

Configuration option

Related configuration option:

- `CONFIG_EDAC_IBECC`

Configuration option

Related configuration option:

- `CONFIG_EDAC`

API Reference

Related code samples

EDAC shell
Test error detection and correction (EDAC) using shell commands.

group edac

Since
2.5

Version
0.8.0

Optional interfaces

EDAC Optional Interfaces

static inline int **edac_inject_set_param1**(const struct *device* *dev, uint64_t value)

Set injection parameter param1.

Set first error injection parameter value.

Parameters

- `dev` – Pointer to the device structure
- `value` – First injection parameter

Return values

- `-ENOSYS` – if the optional interface is not implemented
- `0` – on success, other error code otherwise

static inline int **edac_inject_get_param1**(const struct *device* *dev, uint64_t *value)

Get injection parameter param1.

Get first error injection parameter value.

Parameters

- `dev` – Pointer to the device structure
- `value` – Pointer to the first injection parameter

Return values

- `-ENOSYS` – if the optional interface is not implemented
- `0` – on success, error code otherwise

static inline int **edac_inject_set_param2**(const struct *device* *dev, uint64_t value)

Set injection parameter param2.

Set second error injection parameter value.

Parameters

- `dev` – Pointer to the device structure
- `value` – Second injection parameter

Return values

- `-ENOSYS` – if the optional interface is not implemented
- `0` – on success, error code otherwise

static inline int **edac_inject_get_param2**(const struct *device* *dev, uint64_t *value)

Get injection parameter param2.

Parameters

- `dev` – Pointer to the device structure
- `value` – Pointer to the second injection parameter

Return values

- `-ENOSYS` – if the optional interface is not implemented
- `0` – on success, error code otherwise

static inline int **edac_inject_set_error_type**(const struct *device* *dev, uint32_t error_type)

Set error type value.

Set the value of error type to be injected

Parameters

- `dev` – Pointer to the device structure
- `error_type` – Error type value

Return values

- `-ENOSYS` – if the optional interface is not implemented
- `0` – on success, error code otherwise

static inline int **edac_inject_get_error_type**(const struct *device* *dev, uint32_t *error_type)

Get error type value.

Get the value of error type to be injected

Parameters

- `dev` – Pointer to the device structure
- `error_type` – Pointer to error type value

Return values

- `-ENOSYS` – if the optional interface is not implemented
- `0` – on success, error code otherwise

static inline int **edac_inject_error_trigger**(const struct *device* *dev)

Set injection control.

Trigger error injection.

Parameters

- `dev` – Pointer to the device structure

Return values

- `-ENOSYS` – if the optional interface is not implemented
- `0` – on success, error code otherwise

Mandatory interfaces

EDAC Mandatory Interfaces

static inline int **edac_ecc_error_log_get**(const struct *device* *dev, uint64_t *value)

Get ECC Error Log.

Read value of ECC Error Log.

Parameters

- `dev` – Pointer to the device structure
- `value` – Pointer to the ECC Error Log value

Return values

- `0` – on success, error code otherwise
- `-ENOSYS` – if the mandatory interface is not implemented

static inline int **edac_ecc_error_log_clear**(const struct *device* *dev)

Clear ECC Error Log.

Clear value of ECC Error Log.

Parameters

- `dev` – Pointer to the device structure

Return values

- `0` – on success, error code otherwise
- `-ENOSYS` – if the mandatory interface is not implemented

static inline int **edac_parity_error_log_get**(const struct *device* *dev, uint64_t *value)

Get Parity Error Log.

Read value of Parity Error Log.

Parameters

- `dev` – Pointer to the device structure

- `value` – Pointer to the parity Error Log value

Return values

- `0` – on success, error code otherwise
- `-ENOSYS` – if the mandatory interface is not implemented

static inline int **edac_parity_error_log_clear**(const struct *device* *dev)

Clear Parity Error Log.

Clear value of Parity Error Log.

Parameters

- `dev` – Pointer to the device structure

Return values

- `0` – on success, error code otherwise
- `-ENOSYS` – if the mandatory interface is not implemented

static inline int **edac_errors_cor_get**(const struct *device* *dev)

Get number of correctable errors.

Parameters

- `dev` – Pointer to the device structure

Return values

- `num` – Number of correctable errors
- `-ENOSYS` – if the mandatory interface is not implemented

static inline int **edac_errors_uc_get**(const struct *device* *dev)

Get number of uncorrectable errors.

Parameters

- `dev` – Pointer to the device structure

Return values

- `num` – Number of uncorrectable errors
- `-ENOSYS` – if the mandatory interface is not implemented

static inline int **edac_notify_callback_set**(const struct *device* *dev,
edac_notify_callback_f cb)

Register callback function for memory error exception.

This callback runs in interrupt context

Parameters

- `dev` – EDAC driver device to install callback
- `cb` – Callback function pointer

Return values

- `0` – on success, error code otherwise
- `-ENOSYS` – if the mandatory interface is not implemented

Enums

enum `edac_error_type`

EDAC error type.

Values:

enumerator `EDAC_ERROR_TYPE_DRAM_COR` = *BIT*(0)

Correctable error type.

enumerator `EDAC_ERROR_TYPE_DRAM_UC` = *BIT*(1)

Uncorrectable error type.

7.6.19 Flash

Overview

Flash offset concept

Offsets used by the user API are expressed in relation to the flash memory beginning address. This rule shall be applied to all flash controller regular memory that layout is accessible via API for retrieving the layout of pages (see `CONFIG_FLASH_PAGE_LAYOUT`).

An exception from the rule may be applied to a vendor-specific flash dedicated-purpose region (such a region obviously can't be covered under API for retrieving the layout of pages).

User API Reference

Related code samples

AT45 DataFlash driver
Use the AT45 family DataFlash driver to interact with the flash memory over SPI.

ESP32 Flash Memory-Mapped
Write data into scratch area and read it using flash API and memory-mapped pointer.

Flash shell
Explore a flash device using shell commands.

JEDEC MSPI-NOR flash
Use the flash API to interact with a MSPI NOR serial flash memory device.

JEDEC SPI-NOR flash
Use the flash API to interact with an SPI NOR serial flash memory device.

JESD216 flash
Use the JESD216 flash API to extract information from a compatible serial memory device.

nRF SoC Internal Storage
Use the flash API to interact with the SoC flash.

group `flash_interface`

FLASH Interface.

Since
1.2

Version
1.0.0

Defines

`FLASH_ERASE_C_EXPLICIT`
Set for ordinary Flash where erase is needed before write of random data.

`FLASH_ERASE_CAPS_UNSET`
Reserved for users as initializer for variables that will later store capabilities.

`FLASH_ERASE_C_SUPPORTED`

`FLASH_ERASE_C_VAL_BIT`

`FLASH_ERASE_UNIFORM_PAGE`

`FLASH_EX_OP_VENDOR_BASE`

`FLASH_EX_OP_IS_VENDOR(c)`

Typedefs

typedef bool (*`flash_page_cb`)(const struct *flash_pages_info* *info, void *data)
Callback type for iterating over flash pages present on a device.

The callback should return true to continue iterating, and false to halt.

See also

flash_page_foreach()

Param info
Information for current page

Param data
Private data for callback

Return
True to continue iteration, false to halt iteration.

Enums

enum `flash_ex_op_types`

Enumeration for extra flash operations.

Values:

enumerator `FLASH_EX_OP_RESET` = 0

Functions

static inline int `flash_params_get_erase_cap`(const struct *flash_parameters* *p)

int `flash_read`(const struct *device* *dev, off_t offset, void *data, size_t len)

Read data from flash.

All flash drivers support reads without alignment restrictions on the read offset, the read size, or the destination address.

Parameters

- `dev` – : flash dev
- `offset` – : Offset (byte aligned) to read
- `data` – : Buffer to store read data
- `len` – : Number of bytes to read.

Returns

0 on success, negative errno code on fail.

int `flash_write`(const struct *device* *dev, off_t offset, const void *data, size_t len)

Write buffer into flash memory.

All flash drivers support a source buffer located either in RAM or SoC flash, without alignment restrictions on the source address. Write size and offset must be multiples of the minimum write block size supported by the driver.

Any necessary write protection management is performed by the driver write implementation itself.

Parameters

- `dev` – : flash device
- `offset` – : starting offset for the write
- `data` – : data to write
- `len` – : Number of bytes to write

Returns

0 on success, negative errno code on fail.

int `flash_erase`(const struct *device* *dev, off_t offset, size_t size)

Erase part or all of a flash memory.

Acceptable values of erase size and offset are subject to hardware-specific multiples of page size and offset. Please check the API implemented by the underlying sub driver, for example by using *flash_get_page_info_by_offs()* if that is supported by your flash driver.

Any necessary erase protection management is performed by the driver erase implementation itself.

The function should be used only for devices that are really explicit erase devices; in case when code relies on erasing device, i.e. setting it to erase-value, prior to some

operations, but should work with explicit erase and RAM non-volatile devices, then flash_flatten should rather be used.

See also

flash_flatten()

See also

flash_get_page_info_by_offs()

See also

flash_get_page_info_by_idx()

Parameters

- `dev` – : flash device
- `offset` – : erase area starting offset
- `size` – : size of area to be erased

Returns

0 on success, negative errno code on fail.

int `flash_fill`(const struct *device* *dev, uint8_t val, off_t offset, size_t size)

Fill selected range of device with specified value.

Utility function that allows to fill specified range on a device with provided value. The `offset` and `size` of range need to be aligned to a write block size of a device.

Parameters

- `dev` – : flash device
- `val` – : value to use for filling the range
- `offset` – : offset of the range to fill
- `size` – : size of the range

Returns

0 on success, negative errno code on fail.

int `flash_flatten`(const struct *device* *dev, off_t offset, size_t size)

Erase part or all of a flash memory or level it.

If device is explicit erase type device or device driver provides erase callback, the callback of the device is called, in which it behaves the same way as flash_erase. If a device does not require explicit erase, either because it has no erase at all or has auto-erase/erase-on-write, and does not provide erase callback then erase is emulated by leveling selected device memory area with erase_value assigned to device.

Erase page offset and size are constrains of paged, explicit erase devices, but can be relaxed with devices without such requirement, which means that it is up to user code to make sure they are correct as the function will return on, if these constrains are not met, -EINVAL for paged device, but may succeed on non-explicit erase devices. For RAM non-volatile devices the erase pages are emulated, at this point, to allow smooth

transition for code relying on device being paged to function properly; but this is completely software constrain.

Generally: if your code previously required device to be erase prior to some actions to work, replace flash_erase calls with this function; but if your code can work with non-volatile RAM type devices, without emulating erase, you should rather have different path of execution for page-erase, i.e. Flash, devices and call flash_erase for them.

See also

flash_erase()

Parameters

- `dev` – : flash device
- `offset` – : erase area starting offset
- `size` – : size of area to be erased

Returns

0 on success, negative errno code on fail.

int `flash_get_page_info_by_offs`(const struct *device* *dev, off_t offset, struct *flash_pages_info* *info)

Get the size and start offset of flash page at certain flash offset.

Parameters

- `dev` – flash device
- `offset` – Offset within the page
- `info` – Page Info structure to be filled

Returns

0 on success, -EINVAL if page of the offset doesn't exist.

int `flash_get_page_info_by_idx`(const struct *device* *dev, uint32_t page_index, struct *flash_pages_info* *info)

Get the size and start offset of flash page of certain index.

Parameters

- `dev` – flash device
- `page_index` – Index of the page. Index are counted from 0.
- `info` – Page Info structure to be filled

Returns

0 on success, -EINVAL if page of the index doesn't exist.

size_t `flash_get_page_count`(const struct *device* *dev)

Get the total number of flash pages.

Parameters

- `dev` – flash device

Returns

Number of flash pages.

void **flash_page_foreach**(const struct *device* *dev, *flash_page_cb* cb, void *data)

Iterate over all flash pages on a device.

This routine iterates over all flash pages on the given device, ordered by increasing start offset. For each page, it invokes the given callback, passing it the page's information and a private data object.

Parameters

- **dev** – Device whose pages to iterate over
- **cb** – Callback to invoke for each flash page
- **data** – Private data for callback function

int **flash_sfdp_read**(const struct *device* *dev, off_t offset, void *data, size_t len)

Read data from Serial Flash Discoverable Parameters.

This routine reads data from a serial flash device compatible with the JEDEC JESD216 standard for encoding flash memory characteristics.

Availability of this API is conditional on selecting `CONFIG_FLASH_JESD216_API` and support of that functionality in the driver underlying `dev`.

Parameters

- **dev** – device from which parameters will be read
- **offset** – address within the SFDP region containing data of interest
- **data** – where the data to be read will be placed
- **len** – the number of bytes of data to be read

Return values

- **0** – on success
- **-ENOTSUP** – if the flash driver does not support SFDP access
- **negative** – values for other errors.

int **flash_read_jedec_id**(const struct *device* *dev, uint8_t *id)

Read the JEDEC ID from a compatible flash device.

Parameters

- **dev** – device from which id will be read
- **id** – pointer to a buffer of at least 3 bytes into which id will be stored

Return values

- **0** – on successful store of 3-byte JEDEC id
- **-ENOTSUP** – if flash driver doesn't support this function
- **negative** – values for other errors

size_t **flash_get_write_block_size**(const struct *device* *dev)

Get the minimum write block size supported by the driver.

The write block size supported by the driver might differ from the write block size of memory used because the driver might implements write-modify algorithm.

Parameters

- **dev** – flash device

Returns

write block size in bytes.

const struct *flash_parameters* *`flash_get_parameters`(const struct *device* *dev)

Get pointer to *flash_parameters* structure.

Returned pointer points to a structure that should be considered constant through a runtime, regardless if it is defined in RAM or Flash. Developer is free to cache the structure pointer or copy its contents.

Returns
pointer to *flash_parameters* structure characteristic for the device.

int `flash_ex_op`(const struct *device* *dev, uint16_t code, const uintptr_t in, void *out)

Execute flash extended operation on given device.

Besides of standard flash operations like write or erase, flash controllers also support additional features like write protection or readout protection. These features are not available in every flash controller, what's more controllers can implement it in a different way.

It doesn't make sense to add a separate flash API function for every flash controller feature, because it could be unique (supported on small number of flash controllers) or the API won't be able to represent the same feature on every flash controller.

Parameters

- `dev` – Flash device
- `code` – Operation which will be executed on the device.
- `in` – Pointer to input data used by operation. If operation doesn't need any input data it could be NULL.
- `out` – Pointer to operation output data. If operation doesn't produce any output it could be NULL.

Return values

- `0` – on success.
- `-ENOTSUP` – if given device doesn't support extended operation.
- `-ENOSYS` – if support for extended operations is not enabled in Kconfig
- `negative` – value on extended operation errors.

struct `flash_parameters`

#include <flash.h> Flash memory parameters.

Contents of this structure suppose to be filled in during flash device initialization and stay constant through a runtime.

Public Members

const size_t `write_block_size`

Minimal write alignment and size.

uint8_t `erase_value`

Value the device is filled in erased areas.

struct `flash_pages_info`

#include <flash.h>

Implementation interface API Reference

group `flash_internal_interface`

FLASH internal Interface.

Typedefs

typedef int (*`flash_api_read`)(const struct *device* *dev, off_t offset, void *data, size_t len)

typedef int (*`flash_api_write`)(const struct *device* *dev, off_t offset, const void *data, size_t len)

Flash write implementation handler type.

> **Note**
>
> Any necessary write protection management must be performed by the driver, with the driver responsible for ensuring the "write-protect" after the operation completes (successfully or not) matches the write-protect state when the operation was started.

typedef int (*`flash_api_erase`)(const struct *device* *dev, off_t offset, size_t size)

Flash erase implementation handler type.

The callback is optional for RAM non-volatile devices, which do not require erase by design, but may be provided if it allows device to work more effectively, or if device has a support for internal fill operation the erase in driver uses.

> **Note**
>
> Any necessary erase protection management must be performed by the driver, with the driver responsible for ensuring the "erase-protect" after the operation completes (successfully or not) matches the erase-protect state when the operation was started.

typedef const struct *flash_parameters* *(*`flash_api_get_parameters`)(const struct *device* *dev)

typedef void (*`flash_api_pages_layout`)(const struct *device* *dev, const struct *flash_pages_layout* **layout, size_t *layout_size)

Retrieve a flash device's layout.

A flash device layout is a run-length encoded description of the pages on the device. (Here, "page" means the smallest erasable area on the flash device.)

For flash memories which have uniform page sizes, this routine returns an array of length 1, which specifies the page size and number of pages in the memory.

Layouts for flash memories with nonuniform page sizes will be returned as an array with multiple elements, each of which describes a group of pages that all have the same size. In this case, the sequence of array elements specifies the order in which these groups occur on the device.

Param dev
Flash device whose layout to retrieve.

Param layout
The flash layout will be returned in this argument.

Param layout_size
The number of elements in the returned layout.

typedef int (*`flash_api_sfdp_read`)(const struct *device* *dev, off_t offset, void *data, size_t len)

typedef int (*`flash_api_read_jedec_id`)(const struct *device* *dev, uint8_t *id)

typedef int (*`flash_api_ex_op`)(const struct *device* *dev, uint16_t code, const uintptr_t in, void *out)

struct `flash_pages_layout`
#include <flash.h>

struct `flash_driver_api`
#include <flash.h>

7.6.20 Fuel Gauge

The fuel gauge subsystem exposes an API to uniformly access battery fuel gauge devices. Currently, only reading data is supported.

Note: This API is currently experimental and this doc will be significantly changed as new features are added to the API.

Basic Operation

Properties Fundamentally, a property is a quantity that a fuel gauge device can measure.

Fuel gauges typically support multiple properties, such as temperature readings of the battery-pack or present-time current/voltage.

Properties are fetched by the client one at a time using *`fuel_gauge_get_prop()`*, or fetched in a batch using *`fuel_gauge_get_props()`*.

Properties are set by the client one at a time using *`fuel_gauge_set_prop()`*, or set in a batch using *`fuel_gauge_set_props()`*.

Battery Cutoff Many fuel gauges embedded within battery packs expose a register address that when written to with a specific payload will do a battery cutoff. This battery cutoff is often referred to as ship, shelf, or sleep mode due to its utility in reducing battery drain while devices are stored or shipped.

The fuel gauge API exposes battery cutoff with the *`fuel_gauge_battery_cutoff()`* function.

Caching The Fuel Gauge API explicitly provides no caching for its clients.

API Reference

group `fuel_gauge_interface`

Fuel Gauge Interface.

Since
3.3

Version
0.1.0

Defines

`SBS_GAUGE_MANUFACTURER_NAME_MAX_SIZE`

Data structures for reading SBS buffer properties.

`SBS_GAUGE_DEVICE_NAME_MAX_SIZE`

`SBS_GAUGE_DEVICE_CHEMISTRY_MAX_SIZE`

Typedefs

typedef uint16_t **`fuel_gauge_prop_t`**

typedef int (***`fuel_gauge_get_property_t`**)(const struct *device* *dev, *fuel_gauge_prop_t* prop, union *fuel_gauge_prop_val* *val)

Callback API for getting a fuel_gauge property.

See fuel_gauge_get_property() for argument description

typedef int (***`fuel_gauge_set_property_t`**)(const struct *device* *dev, *fuel_gauge_prop_t* prop, union *fuel_gauge_prop_val* val)

Callback API for setting a fuel_gauge property.

See fuel_gauge_set_property() for argument description

typedef int (***`fuel_gauge_get_buffer_property_t`**)(const struct *device* *dev, *fuel_gauge_prop_t* prop_type, void *dst, size_t dst_len)

Callback API for getting a fuel_gauge buffer property.

See fuel_gauge_get_buffer_property() for argument description

typedef int (***`fuel_gauge_battery_cutoff_t`**)(const struct *device* *dev)

Callback API for doing a battery cutoff.

See *fuel_gauge_battery_cutoff()* for argument description

Enums

enum fuel_gauge_prop_type

Values:

enumerator FUEL_GAUGE_AVG_CURRENT = 0

Runtime Dynamic Battery Parameters.

Provide a 1 minute average of the current on the battery. Does not check for flags or whether those values are bad readings. See driver instance header for details on implementation and how the average is calculated. Units in uA negative=discharging

enumerator FUEL_GAUGE_BATTERY_CUTOFF

Used to cutoff the battery from the system - useful for storage/shipping of devices.

enumerator FUEL_GAUGE_CURRENT

Battery current (uA); negative=discharging.

enumerator FUEL_GAUGE_CHARGE_CUTOFF

Whether the battery underlying the fuel-gauge is cut off from charge.

enumerator FUEL_GAUGE_CYCLE_COUNT

Cycle count in 1/100ths (number of charge/discharge cycles)

enumerator FUEL_GAUGE_CONNECT_STATE

Connect state of battery.

enumerator FUEL_GAUGE_FLAGS

General Error/Runtime Flags.

enumerator FUEL_GAUGE_FULL_CHARGE_CAPACITY

Full Charge Capacity in uAh (might change in some implementations to determine wear)

enumerator FUEL_GAUGE_PRESENT_STATE

Is the battery physically present.

enumerator FUEL_GAUGE_REMAINING_CAPACITY

Remaining capacity in uAh.

enumerator FUEL_GAUGE_RUNTIME_TO_EMPTY

Remaining battery life time in minutes.

enumerator FUEL_GAUGE_RUNTIME_TO_FULL

Remaining time in minutes until battery reaches full charge.

enumerator FUEL_GAUGE_SBS_MFR_ACCESS

Retrieve word from SBS1.1 ManufactuerAccess.

enumerator FUEL_GAUGE_ABSOLUTE_STATE_OF_CHARGE
Absolute state of charge (percent, 0-100) - expressed as % of design capacity.

enumerator FUEL_GAUGE_RELATIVE_STATE_OF_CHARGE
Relative state of charge (percent, 0-100) - expressed as % of full charge capacity.

enumerator FUEL_GAUGE_TEMPERATURE
Temperature in 0.1 K.

enumerator FUEL_GAUGE_VOLTAGE
Battery voltage (uV)

enumerator FUEL_GAUGE_SBS_MODE
Battery Mode (flags)

enumerator FUEL_GAUGE_CHARGE_CURRENT
Battery desired Max Charging Current (uA)

enumerator FUEL_GAUGE_CHARGE_VOLTAGE
Battery desired Max Charging Voltage (uV)

enumerator FUEL_GAUGE_STATUS
Alarm, Status and Error codes (flags)

enumerator FUEL_GAUGE_DESIGN_CAPACITY
Design Capacity (mAh or 10mWh)

enumerator FUEL_GAUGE_DESIGN_VOLTAGE
Design Voltage (mV)

enumerator FUEL_GAUGE_SBS_ATRATE
AtRate (mA or 10 mW)

enumerator FUEL_GAUGE_SBS_ATRATE_TIME_TO_FULL
AtRateTimeToFull (minutes)

enumerator FUEL_GAUGE_SBS_ATRATE_TIME_TO_EMPTY
AtRateTimeToEmpty (minutes)

enumerator FUEL_GAUGE_SBS_ATRATE_OK
AtRateOK (boolean)

enumerator FUEL_GAUGE_SBS_REMAINING_CAPACITY_ALARM
Remaining Capacity Alarm (mAh or 10mWh)

enumerator FUEL_GAUGE_SBS_REMAINING_TIME_ALARM
Remaining Time Alarm (minutes)

enumerator FUEL_GAUGE_MANUFACTURER_NAME

Manufacturer of pack (1 byte length + 20 bytes data)

enumerator FUEL_GAUGE_DEVICE_NAME

Name of pack (1 byte length + 20 bytes data)

enumerator FUEL_GAUGE_DEVICE_CHEMISTRY

Chemistry (1 byte length + 4 bytes data)

enumerator FUEL_GAUGE_COMMON_COUNT

Reserved to demark end of common fuel gauge properties.

enumerator FUEL_GAUGE_CUSTOM_BEGIN

Reserved to demark downstream custom properties - use this value as the actual value may change over future versions of this API.

enumerator FUEL_GAUGE_PROP_MAX = UINT16_MAX

Reserved to demark end of valid enum properties.

Functions

int fuel_gauge_get_prop(const struct *device* *dev, *fuel_gauge_prop_t* prop, union *fuel_gauge_prop_val* *val)

Fetch a battery fuel-gauge property.

Parameters

- dev – Pointer to the battery fuel-gauge device
- prop – Type of property to be fetched from device
- val – pointer to a union *fuel_gauge_prop_val* where the property is read into from the fuel gauge device.

Returns

0 if successful, negative errno code if failure.

int fuel_gauge_get_props(const struct *device* *dev, *fuel_gauge_prop_t* *props, union *fuel_gauge_prop_val* *vals, size_t len)

Fetch multiple battery fuel-gauge properties.

The default implementation is the same as calling *fuel_gauge_get_prop()* multiple times. A driver may implement the get_properties field of the fuel gauge driver APIs struct to override this implementation.

Parameters

- dev – Pointer to the battery fuel-gauge device
- props – Array of the type of property to be fetched from device, each index corresponds to the same index of the vals input array.
- vals – Pointer to array of union *fuel_gauge_prop_val* where the property is read into from the fuel gauge device. The vals array is not permuted.
- len – number of properties in props & vals array

Returns

0 if successful, negative errno code of first failing property

int `fuel_gauge_set_prop`(const struct *device* *dev, *fuel_gauge_prop_t* prop, union *fuel_gauge_prop_val* val)

Set a battery fuel-gauge property.

Parameters

- `dev` – Pointer to the battery fuel-gauge device
- `prop` – Type of property that's being set
- `val` – Value to set associated prop property.

Returns

0 if successful, negative errno code of first failing property

int `fuel_gauge_set_props`(const struct *device* *dev, *fuel_gauge_prop_t* *props, union *fuel_gauge_prop_val* *vals, size_t len)

Set a battery fuel-gauge property.

Parameters

- `dev` – Pointer to the battery fuel-gauge device
- `props` – Array of the type of property to be set, each index corresponds to the same index of the vals input array.
- `vals` – Pointer to array of union *fuel_gauge_prop_val* where the property is written the fuel gauge device. The vals array is not permuted.
- `len` – number of properties in props array

Returns

return=0 if successful. Otherwise, return array index of failing property.

int `fuel_gauge_get_buffer_prop`(const struct *device* *dev, *fuel_gauge_prop_t* prop_type, void *dst, size_t dst_len)

Fetch a battery fuel-gauge buffer property.

Parameters

- `dev` – Pointer to the battery fuel-gauge device
- `prop_type` – Type of property to be fetched from device
- `dst` – byte array or struct that will hold the buffer data that is read from the fuel gauge
- `dst_len` – the length of the destination array in bytes

Returns

return=0 if successful, return < 0 if getting property failed, return 0 on success

int `fuel_gauge_battery_cutoff`(const struct *device* *dev)

Have fuel gauge cutoff its associated battery.

Parameters

- `dev` – Pointer to the battery fuel-gauge device

Returns

return=0 if successful and battery cutoff is now in process, return < 0 if failed to do battery cutoff.

union `fuel_gauge_prop_val`

#include <fuel_gauge.h> Property field to value/type union.

Public Members

int **avg_current**
FUEL_GAUGE_AVG_CURRENT.

bool **cutoff**
FUEL_GAUGE_CHARGE_CUTOFF.

int **current**
FUEL_GAUGE_CURRENT.

uint32_t **cycle_count**
FUEL_GAUGE_CYCLE_COUNT.

uint32_t **flags**
FUEL_GAUGE_FLAGS.

uint32_t **full_charge_capacity**
FUEL_GAUGE_FULL_CHARGE_CAPACITY.

uint32_t **remaining_capacity**
FUEL_GAUGE_REMAINING_CAPACITY.

uint32_t **runtime_to_empty**
FUEL_GAUGE_RUNTIME_TO_EMPTY.

uint32_t **runtime_to_full**
FUEL_GAUGE_RUNTIME_TO_FULL.

uint16_t **sbs_mfr_access_word**
FUEL_GAUGE_SBS_MFR_ACCESS.

uint8_t **absolute_state_of_charge**
FUEL_GAUGE_ABSOLUTE_STATE_OF_CHARGE.

uint8_t **relative_state_of_charge**
FUEL_GAUGE_RELATIVE_STATE_OF_CHARGE.

uint16_t **temperature**
FUEL_GAUGE_TEMPERATURE.

int **voltage**
FUEL_GAUGE_VOLTAGE.

uint16_t **sbs_mode**
FUEL_GAUGE_SBS_MODE.

uint32_t **chg_current**
FUEL_GAUGE_CHARGE_CURRENT.

uint32_t **chg_voltage**
FUEL_GAUGE_CHARGE_VOLTAGE.

uint16_t **fg_status**
FUEL_GAUGE_STATUS.

uint16_t **design_cap**
FUEL_GAUGE_DESIGN_CAPACITY.

uint16_t **design_volt**
FUEL_GAUGE_DESIGN_VOLTAGE.

int16_t **sbs_at_rate**
FUEL_GAUGE_SBS_ATRATE.

uint16_t **sbs_at_rate_time_to_full**
FUEL_GAUGE_SBS_ATRATE_TIME_TO_FULL.

uint16_t **sbs_at_rate_time_to_empty**
FUEL_GAUGE_SBS_ATRATE_TIME_TO_EMPTY

bool **sbs_at_rate_ok**
FUEL_GAUGE_SBS_ATRATE_OK.

uint16_t **sbs_remaining_capacity_alarm**
FUEL_GAUGE_SBS_REMAINING_CAPACITY_ALARM.

uint16_t **sbs_remaining_time_alarm**
FUEL_GAUGE_SBS_REMAINING_TIME_ALARM.

struct **sbs_gauge_manufacturer_name**
#include <fuel_gauge.h>

struct **sbs_gauge_device_name**
#include <fuel_gauge.h>

struct **sbs_gauge_device_chemistry**
#include <fuel_gauge.h>

struct **fuel_gauge_driver_api**
#include <fuel_gauge.h>

Public Members

fuel_gauge_get_property_t **get_property**
Note: Historically this API allowed drivers to implement a custom multi-get/set property function, this was added so drivers could potentially optimize batch read with their specific chip.

However, it was removed because of no existing concrete case upstream. If this need is demonstrated, we can add this back in as an API field.

group `fuel_gauge_emulator_backend`

Fuel gauge backend emulator APIs.

Functions

int `emul_fuel_gauge_set_battery_charging`(const struct *emul* *target, uint32_t uV, int uA)

Set charging for fuel gauge associated battery.

Set how much the battery associated with a fuel gauge IC is charging or discharging. Where voltage is always positive and a positive or negative current denotes charging or discharging, respectively.

Parameters

- `target` – Pointer to the emulator structure for the fuel gauge emulator instance.
- `uV` – Microvolts describing the battery voltage.
- `uA` – Microamps describing the battery current where negative is discharging.

Return values

- `0` – If successful.
- `-EINVAL` – if mV or mA are 0.

int `emul_fuel_gauge_is_battery_cutoff`(const struct *emul* *target, bool *cutoff)

Check if the battery has been cut off.

Parameters

- `target` – Pointer to the emulator structure for the fuel gauge emulator instance.
- `cutoff` – Pointer to bool storing variable.

Return values

- `0` – If successful.
- `-ENOTSUP` – if not supported by emulator.

7.6.21 GNSS (Global Navigation Satellite System)

Overview

GNSS is a general term which covers satellite systems used for navigation, like GPS (Global Positioning System). GNSS services are usually accessed through GNSS modems which receive and process GNSS signals to determine their position, or more specifically, their antennas position. They usually additionally provide a precise time synchronization mechanism, commonly named PPS (Pulse-Per-Second).

Subsystem support

The GNSS subsystem is based on the *Modem modules*. The GNSS subsystem covers everything from sending and receiving commands to and from the modem, to parsing, creating and processing NMEA0183 messages.

Adding support for additional NMEA0183 based GNSS modems requires little more than implementing power management and configuration for the specific GNSS modem.

Adding support for GNSS modems which use other protocols and/or buses than the usual NMEA0183 over UART is possible, but will require a bit more work from the driver developer.

Configuration Options

Related configuration options:

- `CONFIG_GNSS`
- `CONFIG_GNSS_SATELLITES`
- `CONFIG_GNSS_DUMP_TO_LOG`

Navigation Reference

Related code samples

GNSS
Connect to a GNSS device to obtain time, navigation data, and satellite information.

group `navigation`

Navigation utilities.

Functions

int `navigation_distance`(uint64_t *distance, const struct *navigation_data* *p1, const struct *navigation_data* *p2)

Calculate the distance between two navigation points along the surface of the sphere they are relative to.

Parameters

- `distance` – Destination for calculated distance in millimeters
- `p1` – First navigation point
- `p2` – Second navigation point

Returns
0 if successful

Returns
-EINVAL if either navigation point is invalid

int `navigation_bearing`(uint32_t *bearing, const struct *navigation_data* *from, const struct *navigation_data* *to)

Calculate the bearing from one navigation point to another.

Parameters

- `bearing` – Destination for calculated bearing angle in millidegrees
- `from` – First navigation point
- `to` – Second navigation point

Returns
0 if successful

Returns
-EINVAL if either navigation point is invalid

struct `navigation_data`

#include <*navigation.h*> Navigation data structure.

The structure describes the momentary navigation details of a point relative to a sphere (commonly Earth)

Public Members

int64_t `latitude`

Latitudal position in nanodegrees (0 to +-180E9)

int64_t `longitude`

Longitudal position in nanodegrees (0 to +-180E9)

uint32_t `bearing`

Bearing angle in millidegrees (0 to 360E3)

uint32_t `speed`

Speed in millimeters per second.

int32_t `altitude`

Altitude in millimeters.

GNSS API Reference

Related code samples

GNSS
Connect to a GNSS device to obtain time, navigation data, and satellite information.

group `gnss_interface`

GNSS Interface.

Since
3.6

Version
0.1.0

Defines

GNSS_DATA_CALLBACK_DEFINE(_dev, _callback)

Register a callback structure for GNSS data published.

Parameters

- **_dev** – Device pointer
- **_callback** – The callback function

GNSS_SATELLITES_CALLBACK_DEFINE(_dev, _callback)

Register a callback structure for GNSS satellites published.

Parameters

- **_dev** – Device pointer
- **_callback** – The callback function

Typedefs

typedef int (*gnss_set_fix_rate_t)(const struct *device* *dev, uint32_t fix_interval_ms)

API for setting fix rate.

typedef int (*gnss_get_fix_rate_t)(const struct *device* *dev, uint32_t *fix_interval_ms)

API for getting fix rate.

typedef int (*gnss_set_periodic_config_t)(const struct *device* *dev, const struct *gnss_periodic_config* *periodic_config)

API for setting periodic tracking configuration.

typedef int (*gnss_get_periodic_config_t)(const struct *device* *dev, struct *gnss_periodic_config* *periodic_config)

API for setting periodic tracking configuration.

typedef int (*gnss_set_navigation_mode_t)(const struct *device* *dev, enum *gnss_navigation_mode* mode)

API for setting navigation mode.

typedef int (*gnss_get_navigation_mode_t)(const struct *device* *dev, enum *gnss_navigation_mode* *mode)

API for getting navigation mode.

typedef uint32_t gnss_systems_t

Type storing bitmask of GNSS systems.

typedef int (*gnss_set_enabled_systems_t)(const struct *device* *dev, *gnss_systems_t* systems)

API for enabling systems.

typedef int (*gnss_get_enabled_systems_t)(const struct *device* *dev, *gnss_systems_t* *systems)

API for getting enabled systems.

typedef int (*gnss_get_supported_systems_t)(const struct *device* *dev, *gnss_systems_t* *systems)

API for getting enabled systems.

typedef void (*gnss_data_callback_t)(const struct *device* *dev, const struct *gnss_data* *data)

Template for GNSS data callback.

typedef void (*gnss_satellites_callback_t)(const struct *device* *dev, const struct *gnss_satellite* *satellites, uint16_t size)

Template for GNSS satellites callback.

Enums

enum gnss_pps_mode

GNSS PPS modes.

Values:

enumerator GNSS_PPS_MODE_DISABLED = 0

PPS output disabled.

enumerator GNSS_PPS_MODE_ENABLED = 1

PPS output always enabled.

enumerator GNSS_PPS_MODE_ENABLED_AFTER_LOCK = 2

PPS output enabled from first lock.

enumerator GNSS_PPS_MODE_ENABLED_WHILE_LOCKED = 3

PPS output enabled while locked.

enum gnss_navigation_mode

GNSS navigation modes.

Values:

enumerator GNSS_NAVIGATION_MODE_ZERO_DYNAMICS = 0

Dynamics have no impact on tracking.

enumerator GNSS_NAVIGATION_MODE_LOW_DYNAMICS = 1

Low dynamics have higher impact on tracking.

enumerator GNSS_NAVIGATION_MODE_BALANCED_DYNAMICS = 2

Low and high dynamics have equal impact on tracking.

enumerator GNSS_NAVIGATION_MODE_HIGH_DYNAMICS = 3

High dynamics have higher impact on tracking.

enum **gnss_system**

Systems contained in gnss_systems_t.

Values:

enumerator **GNSS_SYSTEM_GPS** = *BIT*(0)

Global Positioning System (GPS)

enumerator **GNSS_SYSTEM_GLONASS** = *BIT*(1)

GLObal NAvigation Satellite System (GLONASS)

enumerator **GNSS_SYSTEM_GALILEO** = *BIT*(2)

Galileo.

enumerator **GNSS_SYSTEM_BEIDOU** = *BIT*(3)

BeiDou Navigation Satellite System.

enumerator **GNSS_SYSTEM_QZSS** = *BIT*(4)

Quasi-Zenith Satellite System (QZSS)

enumerator **GNSS_SYSTEM_IRNSS** = *BIT*(5)

Indian Regional Navigation Satellite System (IRNSS)

enumerator **GNSS_SYSTEM_SBAS** = *BIT*(6)

Satellite-Based Augmentation System (SBAS)

enumerator **GNSS_SYSTEM_IMES** = *BIT*(7)

Indoor Messaging System (IMES)

enum **gnss_fix_status**

GNSS fix status.

Values:

enumerator **GNSS_FIX_STATUS_NO_FIX** = 0

No GNSS fix acquired.

enumerator **GNSS_FIX_STATUS_GNSS_FIX** = 1

GNSS fix acquired.

enumerator **GNSS_FIX_STATUS_DGNSS_FIX** = 2

Differential GNSS fix acquired.

enumerator **GNSS_FIX_STATUS_ESTIMATED_FIX** = 3

Estimated fix acquired.

enum **gnss_fix_quality**

GNSS fix quality.

Values:

enumerator GNSS_FIX_QUALITY_INVALID = 0
Invalid fix.

enumerator GNSS_FIX_QUALITY_GNSS_SPS = 1
Standard positioning service.

enumerator GNSS_FIX_QUALITY_DGNSS = 2
Differential GNSS.

enumerator GNSS_FIX_QUALITY_GNSS_PPS = 3
Precise positioning service.

enumerator GNSS_FIX_QUALITY_RTK = 4
Real-time kinematic.

enumerator GNSS_FIX_QUALITY_FLOAT_RTK = 5
Floating real-time kinematic.

enumerator GNSS_FIX_QUALITY_ESTIMATED = 6
Estimated fix.

Functions

int gnss_set_fix_rate(const struct *device* *dev, uint32_t fix_interval_ms)
Set the GNSS fix rate.

Parameters

- dev – Device instance
- fix_interval_ms – Fix interval to set in milliseconds

Returns
0 if successful

Returns
-errno negative errno code on failure

int gnss_get_fix_rate(const struct *device* *dev, uint32_t *fix_interval_ms)
Get the GNSS fix rate.

Parameters

- dev – Device instance
- fix_interval_ms – Destination for fix interval in milliseconds

Returns
0 if successful

Returns
-errno negative errno code on failure

int gnss_set_periodic_config(const struct *device* *dev, const struct *gnss_periodic_config* *config)
Set the GNSS periodic tracking configuration.

Parameters

- **dev** – Device instance
- **config** – Periodic tracking configuration to set

Returns
0 if successful

Returns
-errno negative errno code on failure

int **gnss_get_periodic_config**(const struct *device* *dev, struct *gnss_periodic_config* *config)

Get the GNSS periodic tracking configuration.

Parameters

- **dev** – Device instance
- **config** – Destination for periodic tracking configuration

Returns
0 if successful

Returns
-errno negative errno code on failure

int **gnss_set_navigation_mode**(const struct *device* *dev, enum *gnss_navigation_mode* mode)

Set the GNSS navigation mode.

Parameters

- **dev** – Device instance
- **mode** – Navigation mode to set

Returns
0 if successful

Returns
-errno negative errno code on failure

int **gnss_get_navigation_mode**(const struct *device* *dev, enum *gnss_navigation_mode* *mode)

Get the GNSS navigation mode.

Parameters

- **dev** – Device instance
- **mode** – Destination for navigation mode

Returns
0 if successful

Returns
-errno negative errno code on failure

int **gnss_set_enabled_systems**(const struct *device* *dev, *gnss_systems_t* systems)

Set enabled GNSS systems.

Parameters

- **dev** – Device instance
- **systems** – Systems to enable

Returns
0 if successful

Returns
-errno negative errno code on failure

int **gnss_get_enabled_systems**(const struct *device* *dev, *gnss_systems_t* *systems)

Get enabled GNSS systems.

Parameters

- **dev** – Device instance
- **systems** – Destination for enabled systems

Returns
0 if successful

Returns
-errno negative errno code on failure

int **gnss_get_supported_systems**(const struct *device* *dev, *gnss_systems_t* *systems)

Get supported GNSS systems.

Parameters

- **dev** – Device instance
- **systems** – Destination for supported systems

Returns
0 if successful

Returns
-errno negative errno code on failure

struct **gnss_periodic_config**

#include <*gnss.h*> GNSS periodic tracking configuration.

> **Note**
>
> Setting either active_time or inactive_time to 0 will disable periodic function.

Public Members

uint32_t **active_time_ms**

The time the GNSS will spend in the active state in ms.

uint32_t **inactive_time_ms**

The time the GNSS will spend in the inactive state in ms.

struct **gnss_info**

#include <*gnss.h*> GNSS info data structure.

Public Members

uint16_t **satellites_cnt**

Number of satellites being tracked.

uint32_t **hdop**
Horizontal dilution of precision in 1/1000.

enum *gnss_fix_status* **fix_status**
The fix status.

enum *gnss_fix_quality* **fix_quality**
The fix quality.

struct **gnss_time**
#include <*gnss.h*> GNSS time data structure.

Public Members

uint8_t **hour**
Hour [0, 23].

uint8_t **minute**
Minute [0, 59].

uint16_t **millisecond**
Millisecond [0, 59999].

uint8_t **month_day**
Day of month [1, 31].

uint8_t **month**
Month [1, 12].

uint8_t **century_year**
Year [0, 99].

struct **gnss_driver_api**
#include <*gnss.h*> GNSS API structure.

struct **gnss_data**
#include <*gnss.h*> GNSS data structure.

Public Members

struct *navigation_data* **nav_data**
Navigation data acquired.

struct *gnss_info* **info**
GNSS info when navigation data was acquired.

struct *gnss_time* **utc**
UTC time when data was acquired.

struct **gnss_data_callback**
#include <*gnss.h*> GNSS callback structure.

Public Members

const struct *device* ***dev**
Filter callback to GNSS data from this device if not NULL.

gnss_data_callback_t **callback**
Callback called when GNSS data is published.

struct **gnss_satellite**
#include <*gnss.h*> GNSS satellite structure.

Public Members

uint8_t **prn**
Pseudo-random noise sequence.

uint8_t **snr**
Signal-to-noise ratio in dB.

uint8_t **elevation**
Elevation in degrees [0, 90].

uint16_t **azimuth**
Azimuth relative to True North in degrees [0, 359].

enum *gnss_system* **system**
System of satellite.

uint8_t **is_tracked**
True if satellite is being tracked.

struct **gnss_satellites_callback**
#include <*gnss.h*> GNSS callback structure.

Public Members

const struct *device* ***dev**
Filter callback to GNSS data from this device if not NULL.

gnss_satellites_callback_t `callback`
Callback called when GNSS satellites is published.

7.6.22 General-Purpose Input/Output (GPIO)

Overview

Configuration Options

Related configuration options:

- `CONFIG_GPIO`

API Reference

Related code samples

Basic thread manipulation
Spawn multiple threads that blink LEDs and print information to the console.

Blinky
Blink an LED forever using the GPIO API.

Button
Handle GPIO inputs with interrupts.

GPIO with custom Devicetree binding
Use custom Devicetree binding to control a GPIO.

HD44780 LCD controller
Control an HD44780-based LCD display using GPIO pins.

X-NUCLEO-53L0A1 shield
Interact with the 7-segment display and VL53L0X ranging sensor of an X-NUCLEO-53L0A1 shield.

group `gpio_interface`
GPIO Driver APIs.

Since
1.0

Version
1.0.0

GPIO input/output configuration flags

`GPIO_INPUT`
Enables pin as input.

`GPIO_OUTPUT`
Enables pin as output, no change to the output state.

`GPIO_DISCONNECTED`
: Disables pin for both input and output.

`GPIO_OUTPUT_LOW`
: Configures GPIO pin as output and initializes it to a low state.

`GPIO_OUTPUT_HIGH`
: Configures GPIO pin as output and initializes it to a high state.

`GPIO_OUTPUT_INACTIVE`
: Configures GPIO pin as output and initializes it to a logic 0.

`GPIO_OUTPUT_ACTIVE`
: Configures GPIO pin as output and initializes it to a logic 1.

GPIO interrupt configuration flags

The `GPIO_INT_*` flags are used to specify how input GPIO pins will trigger interrupts.

The interrupts can be sensitive to pin physical or logical level. Interrupts sensitive to pin logical level take into account GPIO_ACTIVE_LOW flag. If a pin was configured as Active Low, physical level low will be considered as logical level 1 (an active state), physical level high will be considered as logical level 0 (an inactive state). The GPIO controller should reset the interrupt status, such as clearing the pending bit, etc, when configuring the interrupt triggering properties. Applications should use the `GPIO_INT_MODE_ENABLE_ONLY` and `GPIO_INT_MODE_DISABLE_ONLY` flags to enable and disable interrupts on the pin without changing any GPIO settings.

`GPIO_INT_DISABLE`
: Disables GPIO pin interrupt.

`GPIO_INT_EDGE_RISING`
: Configures GPIO interrupt to be triggered on pin rising edge and enables it.

`GPIO_INT_EDGE_FALLING`
: Configures GPIO interrupt to be triggered on pin falling edge and enables it.

`GPIO_INT_EDGE_BOTH`
: Configures GPIO interrupt to be triggered on pin rising or falling edge and enables it.

`GPIO_INT_LEVEL_LOW`
: Configures GPIO interrupt to be triggered on pin physical level low and enables it.

`GPIO_INT_LEVEL_HIGH`
: Configures GPIO interrupt to be triggered on pin physical level high and enables it.

`GPIO_INT_EDGE_TO_INACTIVE`
: Configures GPIO interrupt to be triggered on pin state change to logical level 0 and enables it.

`GPIO_INT_EDGE_TO_ACTIVE`
: Configures GPIO interrupt to be triggered on pin state change to logical level 1 and enables it.

`GPIO_INT_LEVEL_INACTIVE`
: Configures GPIO interrupt to be triggered on pin logical level 0 and enables it.

`GPIO_INT_LEVEL_ACTIVE`
: Configures GPIO interrupt to be triggered on pin logical level 1 and enables it.

GPIO pin active level flags

`GPIO_ACTIVE_LOW`
: GPIO pin is active (has logical value '1') in low state.

`GPIO_ACTIVE_HIGH`
: GPIO pin is active (has logical value '1') in high state.

GPIO pin drive flags

`GPIO_OPEN_DRAIN`
: Configures GPIO output in open drain mode (wired AND).

> **Note**
>
> 'Open Drain' mode also known as 'Open Collector' is an output configuration which behaves like a switch that is either connected to ground or disconnected.

`GPIO_OPEN_SOURCE`
: Configures GPIO output in open source mode (wired OR).

> **Note**
>
> 'Open Source' is a term used by software engineers to describe output mode opposite to 'Open Drain'. It behaves like a switch that is either connected to power supply or disconnected. There exist no corresponding hardware schematic and the term is generally unknown to hardware engineers.

GPIO pin bias flags

`GPIO_PULL_UP`
: Enables GPIO pin pull-up.

`GPIO_PULL_DOWN`
: Enable GPIO pin pull-down.

Unnamed Group

`STM32_GPIO_WKUP`

STM32 GPIO specific flags.

The driver flags are encoded in the 8 upper bits of *gpio_dt_flags_t* as follows:

- Bit 8: Configure a GPIO pin to power on the system after Poweroff. Configures a GPIO pin to power on the system after Poweroff. This flag is reserved to GPIO pins that are associated with wake-up pins in STM32 PWR devicetree node, through the property “wkup-gpios”.

Defines

`GPIO_DT_SPEC_GET_BY_IDX`(node_id, prop, idx)

Static initializer for a *gpio_dt_spec*.

This returns a static initializer for a *gpio_dt_spec* structure given a devicetree node identifier, a property specifying a GPIO and an index.

Example devicetree fragment:

```
n: node {
   foo-gpios = <&gpio0 1 GPIO_ACTIVE_LOW>,
               <&gpio1 2 GPIO_ACTIVE_LOW>;
}
```

Example usage:

```
const struct gpio_dt_spec spec = GPIO_DT_SPEC_GET_BY_IDX(DT_NODELABEL(n),
                                                         foo_gpios, 1);
// Initializes 'spec' to:
// {
//         .port = DEVICE_DT_GET(DT_NODELABEL(gpio1)),
//         .pin = 2,
//         .dt_flags = GPIO_ACTIVE_LOW
// }
```

The ‘gpio’ field must still be checked for readiness, e.g. using *device_is_ready()*. It is an error to use this macro unless the node exists, has the given property, and that property specifies a GPIO controller, pin number, and flags as shown above.

Parameters

- `node_id` – devicetree node identifier
- `prop` – lowercase-and-underscores property name
- `idx` – logical index into “prop”

Returns

static initializer for a struct *gpio_dt_spec* for the property

`GPIO_DT_SPEC_GET_BY_IDX_OR`(node_id, prop, idx, default_value)

Like *GPIO_DT_SPEC_GET_BY_IDX()*, with a fallback to a default value.

If the devicetree node identifier ‘node_id’ refers to a node with a property ‘prop’, this expands to *GPIO_DT_SPEC_GET_BY_IDX(node_id, prop, idx)*. The `default_value` parameter is not expanded in this case.

Otherwise, this expands to `default_value`.

Parameters

- `node_id` – devicetree node identifier
- `prop` – lowercase-and-underscores property name
- `idx` – logical index into “prop”
- `default_value` – fallback value to expand to

Returns
static initializer for a struct *gpio_dt_spec* for the property, or default_value if the node or property do not exist

`GPIO_DT_SPEC_GET`(node_id, prop)

Equivalent to *GPIO_DT_SPEC_GET_BY_IDX(node_id, prop, 0)*.

See also

GPIO_DT_SPEC_GET_BY_IDX()

Parameters

- `node_id` – devicetree node identifier
- `prop` – lowercase-and-underscores property name

Returns
static initializer for a struct *gpio_dt_spec* for the property

`GPIO_DT_SPEC_GET_OR`(node_id, prop, default_value)

Equivalent to *GPIO_DT_SPEC_GET_BY_IDX_OR(node_id, prop, 0, default_value)*.

See also

GPIO_DT_SPEC_GET_BY_IDX_OR()

Parameters

- `node_id` – devicetree node identifier
- `prop` – lowercase-and-underscores property name
- `default_value` – fallback value to expand to

Returns
static initializer for a struct *gpio_dt_spec* for the property

`GPIO_DT_SPEC_INST_GET_BY_IDX`(inst, prop, idx)

Static initializer for a *gpio_dt_spec* from a DT_DRV_COMPAT instance’s GPIO property at an index.

See also

GPIO_DT_SPEC_GET_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `prop` – lowercase-and-underscores property name
- `idx` – logical index into “prop”

Returns
static initializer for a struct *gpio_dt_spec* for the property

`GPIO_DT_SPEC_INST_GET_BY_IDX_OR`(inst, prop, idx, default_value)

Static initializer for a *gpio_dt_spec* from a DT_DRV_COMPAT instance’s GPIO property at an index, with fallback.

See also

GPIO_DT_SPEC_GET_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `prop` – lowercase-and-underscores property name
- `idx` – logical index into “prop”
- `default_value` – fallback value to expand to

Returns
static initializer for a struct *gpio_dt_spec* for the property

`GPIO_DT_SPEC_INST_GET`(inst, prop)

Equivalent to *GPIO_DT_SPEC_INST_GET_BY_IDX(inst, prop, 0)*.

See also

GPIO_DT_SPEC_INST_GET_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `prop` – lowercase-and-underscores property name

Returns
static initializer for a struct *gpio_dt_spec* for the property

`GPIO_DT_SPEC_INST_GET_OR`(inst, prop, default_value)

Equivalent to *GPIO_DT_SPEC_INST_GET_BY_IDX_OR(inst, prop, 0, default_value)*.

See also

GPIO_DT_SPEC_INST_GET_BY_IDX()

Parameters

- **inst** – DT_DRV_COMPAT instance number
- **prop** – lowercase-and-underscores property name
- **default_value** – fallback value to expand to

Returns
static initializer for a struct *gpio_dt_spec* for the property

GPIO_DT_RESERVED_RANGES_NGPIOS(node_id, ngpios)

Makes a bitmask of reserved GPIOs from DT "gpio-reserved-ranges" property and "ngpios" argument.

This macro returns the value as a bitmask of the "gpio-reserved-ranges" property. This property defines the disabled (or 'reserved') GPIOs in the range 0...ngpios-1 and is specified as an array of value's pairs that define the start offset and size of the reserved ranges.

For example, setting "gpio-reserved-ranges = <3 2>, <10 1>;" means that GPIO offsets 3, 4 and 10 cannot be used even if ngpios = <18>.

The implementation constraint is inherited from common DT limitations: a maximum of 64 pairs can be used (with result limited to bitsize of gpio_port_pins_t type).

NB: Due to the nature of C macros, some incorrect tuple definitions (for example, overlapping or out of range) will produce undefined results.

Also be aware that if ngpios is less than 32 (bit size of DT int type), then all unused MSBs outside the range defined by ngpios will be marked as reserved too.

Example devicetree fragment:

```
a {
        compatible = "some,gpio-controller";
        ngpios = <32>;
        gpio-reserved-ranges = <0  4>, <5  3>, <9  5>, <11 2>, <15 2>,
                               <18 2>, <21 1>, <23 1>, <25 4>, <30 2>;
};

b {
        compatible = "some,gpio-controller";
        ngpios = <18>;
        gpio-reserved-ranges = <3 2>, <10 1>;
};
```

Example usage:

```
struct some_config {
        uint32_t ngpios;
        uint32_t gpios_reserved;
};

static const struct some_config dev_cfg_a = {
        .ngpios = DT_PROP_OR(DT_LABEL(a), ngpios, 0),
        .gpios_reserved = GPIO_DT_RESERVED_RANGES_NGPIOS(DT_LABEL(a),
                                DT_PROP(DT_LABEL(a), ngpios)),
};

static const struct some_config dev_cfg_b = {
        .ngpios = DT_PROP_OR(DT_LABEL(b), ngpios, 0),
        .gpios_reserved = GPIO_DT_RESERVED_RANGES_NGPIOS(DT_LABEL(b),
                                DT_PROP(DT_LABEL(b), ngpios)),
};
```

This expands to:

```
struct some_config {
        uint32_t ngpios;
        uint32_t gpios_reserved;
};

static const struct some_config dev_cfg_a = {
        .ngpios = 32,
        .gpios_reserved = 0xdeadbeef,
                        // 0b1101 1110 1010 1101 1011 1110 1110 1111

static const struct some_config dev_cfg_b = {
        .ngpios = 18,
        .gpios_reserved = 0xfffc0418,
                        // 0b1111 1111 1111 1100 0000 0100 0001 1000
                        // unused MSBs were marked as reserved too
};
```

Parameters

- **node_id** – GPIO controller node identifier.
- **ngpios** – number of GPIOs.

Returns
the bitmask of reserved gpios

GPIO_DT_RESERVED_RANGES(node_id)

Makes a bitmask of reserved GPIOs from the "gpio-reserved-ranges" and "ngpios" DT properties values.

Parameters

- **node_id** – GPIO controller node identifier.

Returns
the bitmask of reserved gpios

GPIO_DT_INST_RESERVED_RANGES_NGPIOS(inst, ngpios)

Makes a bitmask of reserved GPIOs from a DT_DRV_COMPAT instance's "gpio-reserved-ranges" property and "ngpios" argument.

See also

GPIO_DT_RESERVED_RANGES()

Parameters

- **inst** – DT_DRV_COMPAT instance number
- **ngpios** – number of GPIOs

Returns
the bitmask of reserved gpios

GPIO_DT_INST_RESERVED_RANGES(inst)

Make a bitmask of reserved GPIOs from a DT_DRV_COMPAT instance's GPIO "gpio-reserved-ranges" and "ngpios" properties.

See also

GPIO_DT_RESERVED_RANGES()

Parameters

- **inst** – DT_DRV_COMPAT instance number

Returns
the bitmask of reserved gpios

GPIO_DT_PORT_PIN_MASK_NGPIOS_EXC(node_id, ngpios)

Makes a bitmask of allowed GPIOs from DT "gpio-reserved-ranges" property and "ngpios" argument.

This macro is paired with *GPIO_DT_RESERVED_RANGES_NGPIOS()*, however unlike the latter, it returns a bitmask of ALLOWED gpios.

Example devicetree fragment:

```
a {
   compatible = "some,gpio-controller";
   ngpios = <32>;
   gpio-reserved-ranges = <0 8>, <9 5>, <15 16>;
};
```

Example usage:

```
struct some_config {
   uint32_t port_pin_mask;
};

static const struct some_config dev_cfg = {
   .port_pin_mask = GPIO_DT_PORT_PIN_MASK_NGPIOS_EXC(
                           DT_LABEL(a), 32),
};
```

This expands to:

```
struct some_config {
   uint32_t port_pin_mask;
};

static const struct some_config dev_cfg = {
   .port_pin_mask = 0x80004100,
                   // 0b1000 0000 0000 0000 0100 0001 00000 000
};
```

Parameters

- **node_id** – GPIO controller node identifier.
- **ngpios** – number of GPIOs

Returns
the bitmask of allowed gpios

GPIO_DT_INST_PORT_PIN_MASK_NGPIOS_EXC(inst, ngpios)

Makes a bitmask of allowed GPIOs from a DT_DRV_COMPAT instance's "gpio-reserved-ranges" property and "ngpios" argument.

See also

GPIO_DT_NGPIOS_PORT_PIN_MASK_EXC()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `ngpios` – number of GPIOs

Returns
the bitmask of allowed gpios

`GPIO_MAX_PINS_PER_PORT`

Maximum number of pins that are supported by `gpio_port_pins_t`.

`GPIO_DT_FLAGS_MASK`

Mask for DT GPIO flags.

`GPIO_INT_WAKEUP`

Configures GPIO interrupt to wakeup the system from low power mode.

Typedefs

typedef uint32_t **gpio_port_pins_t**

Identifies a set of pins associated with a port.

The pin with index n is present in the set if and only if the bit identified by (1U « n) is set.

typedef uint32_t **gpio_port_value_t**

Provides values for a set of pins associated with a port.

The value for a pin with index n is high (physical mode) or active (logical mode) if and only if the bit identified by (1U « n) is set. Otherwise the value for the pin is low (physical mode) or inactive (logical mode).

Values of this type are often paired with a `gpio_port_pins_t` value that specifies which encoded pin values are valid for the operation.

typedef uint8_t **gpio_pin_t**

Provides a type to hold a GPIO pin index.

This reduced-size type is sufficient to record a pin number, e.g. from a devicetree GPIOS property.

typedef uint16_t **gpio_dt_flags_t**

Provides a type to hold GPIO devicetree flags.

All GPIO flags that can be expressed in devicetree fit in the low 16 bits of the full flags field, so use a reduced-size type to record that part of a GPIOS property.

The lower 8 bits are used for standard flags. The upper 8 bits are reserved for SoC specific flags.

typedef uint32_t **gpio_flags_t**

Provides a type to hold GPIO configuration flags.

This type is sufficient to hold all flags used to control GPIO configuration, whether pin or interrupt.

typedef void (***gpio_callback_handler_t**)(const struct *device* *port, struct *gpio_callback* *cb, *gpio_port_pins_t* pins)

Define the application callback handler function signature.

Note: cb pointer can be used to retrieve private data through *CONTAINER_OF()* if original struct *gpio_callback* is stored in another private structure.

Param port
Device struct for the GPIO device.

Param cb
Original struct *gpio_callback* owning this handler

Param pins
Mask of pins that triggers the callback handler

Functions

static inline bool **gpio_is_ready_dt**(const struct *gpio_dt_spec* *spec)

Validate that GPIO port is ready.

Parameters

- **spec** – GPIO specification from devicetree

Return values

- **true** – if the GPIO spec is ready for use.
- **false** – if the GPIO spec is not ready for use.

int **gpio_pin_interrupt_configure**(const struct *device* *port, *gpio_pin_t* pin, *gpio_flags_t* flags)

Configure pin interrupt.

> **Note**
>
> This function can also be used to configure interrupts on pins not controlled directly by the GPIO module. That is, pins which are routed to other modules such as I2C, SPI, UART.

Parameters

- **port** – Pointer to device structure for the driver instance.
- **pin** – Pin number.
- **flags** – Interrupt configuration flags as defined by GPIO_INT_*.

Return values

- **0** – If successful.
- **-ENOSYS** – If the operation is not implemented by the driver.

- **-ENOTSUP** – If any of the configuration options is not supported (unless otherwise directed by flag documentation).
- **-EINVAL** – Invalid argument.
- **-EBUSY** – Interrupt line required to configure pin interrupt is already in use.
- **-EIO** – I/O error when accessing an external GPIO chip.
- **-EWOULDBLOCK** – if operation would block.

static inline int **gpio_pin_interrupt_configure_dt**(const struct *gpio_dt_spec* *spec, *gpio_flags_t* flags)

Configure pin interrupts from a *gpio_dt_spec*.

This is equivalent to:

```
gpio_pin_interrupt_configure(spec->port, spec->pin, flags);
```

The spec->dt_flags value is not used.

Parameters

- **spec** – GPIO specification from devicetree
- **flags** – interrupt configuration flags

Returns

a value from *gpio_pin_interrupt_configure()*

int **gpio_pin_configure**(const struct *device* *port, *gpio_pin_t* pin, *gpio_flags_t* flags)

Configure a single pin.

Parameters

- **port** – Pointer to device structure for the driver instance.
- **pin** – Pin number to configure.
- **flags** – Flags for pin configuration: 'GPIO input/output configuration flags', 'GPIO pin drive flags', 'GPIO pin bias flags'.

Return values

- **0** – If successful.
- **-ENOTSUP** – if any of the configuration options is not supported (unless otherwise directed by flag documentation).
- **-EINVAL** – Invalid argument.
- **-EIO** – I/O error when accessing an external GPIO chip.
- **-EWOULDBLOCK** – if operation would block.

static inline int **gpio_pin_configure_dt**(const struct *gpio_dt_spec* *spec, *gpio_flags_t* extra_flags)

Configure a single pin from a *gpio_dt_spec* and some extra flags.

This is equivalent to:

```
gpio_pin_configure(spec->port, spec->pin, spec->dt_flags | extra_flags);
```

Parameters

- **spec** – GPIO specification from devicetree
- **extra_flags** – additional flags

Returns
a value from *gpio_pin_configure()*

int **gpio_port_get_direction**(const struct *device* *port, *gpio_port_pins_t* map, *gpio_port_pins_t* *inputs, *gpio_port_pins_t* *outputs)

Get direction of select pins in a port.

Retrieve direction of each pin specified in `map`.

If `inputs` or `outputs` is NULL, then this function does not get the respective input or output direction information.

Parameters

- **port** – Pointer to the device structure for the driver instance.
- **map** – Bitmap of pin directions to query.
- **inputs** – Pointer to a variable where input directions will be stored.
- **outputs** – Pointer to a variable where output directions will be stored.

Return values

- **0** – If successful.
- **-ENOSYS** – if the underlying driver does not support this call.
- **-EIO** – I/O error when accessing an external GPIO chip.
- **-EWOULDBLOCK** – if operation would block.

static inline int **gpio_pin_is_input**(const struct *device* *port, *gpio_pin_t* pin)

Check if `pin` is configured for input.

Parameters

- **port** – Pointer to device structure for the driver instance.
- **pin** – Pin number to query the direction of

Return values

- **1** – if `pin` is configured as *GPIO_INPUT*.
- **0** – if `pin` is not configured as *GPIO_INPUT*.
- **-ENOSYS** – if the underlying driver does not support this call.
- **-EIO** – I/O error when accessing an external GPIO chip.
- **-EWOULDBLOCK** – if operation would block.

static inline int **gpio_pin_is_input_dt**(const struct *gpio_dt_spec* *spec)

Check if a single pin from *gpio_dt_spec* is configured for input.

This is equivalent to:

```
gpio_pin_is_input(spec->port, spec->pin);
```

Parameters

- **spec** – GPIO specification from devicetree.

Returns
A value from *gpio_pin_is_input()*.

static inline int **gpio_pin_is_output**(const struct *device* *port, *gpio_pin_t* pin)

Check if `pin` is configured for output.

Parameters

- **port** – Pointer to device structure for the driver instance.
- **pin** – Pin number to query the direction of

Return values

- **1** – if `pin` is configured as *GPIO_OUTPUT*.
- **0** – if `pin` is not configured as *GPIO_OUTPUT*.
- **-ENOSYS** – if the underlying driver does not support this call.
- **-EIO** – I/O error when accessing an external GPIO chip.
- **-EWOULDBLOCK** – if operation would block.

static inline int **gpio_pin_is_output_dt**(const struct *gpio_dt_spec* *spec)

Check if a single pin from *gpio_dt_spec* is configured for output.

This is equivalent to:

```
gpio_pin_is_output(spec->port, spec->pin);
```

Parameters

- **spec** – GPIO specification from devicetree.

Returns

A value from *gpio_pin_is_output()*.

int **gpio_pin_get_config**(const struct *device* *port, *gpio_pin_t* pin, *gpio_flags_t* *flags)

Get a configuration of a single pin.

Parameters

- **port** – Pointer to device structure for the driver instance.
- **pin** – Pin number which configuration is get.
- **flags** – Pointer to variable in which the current configuration will be stored if function is successful.

Return values

- **0** – If successful.
- **-ENOSYS** – if getting current pin configuration is not implemented by the driver.
- **-EINVAL** – Invalid argument.
- **-EIO** – I/O error when accessing an external GPIO chip.
- **-EWOULDBLOCK** – if operation would block.

static inline int **gpio_pin_get_config_dt**(const struct *gpio_dt_spec* *spec, *gpio_flags_t* *flags)

Get a configuration of a single pin from a *gpio_dt_spec*.

This is equivalent to:

```
gpio_pin_get_config(spec->port, spec->pin, flags);
```

Parameters

- **spec** – GPIO specification from devicetree
- **flags** – Pointer to variable in which the current configuration will be stored if function is successful.

Returns
a value from *gpio_pin_configure()*

int **gpio_port_get_raw**(const struct *device* *port, *gpio_port_value_t* *value)

Get physical level of all input pins in a port.

A low physical level on the pin will be interpreted as value 0. A high physical level will be interpreted as value 1. This function ignores GPIO_ACTIVE_LOW flag.

Value of a pin with index n will be represented by bit n in the returned port value.

Parameters

- **port** – Pointer to the device structure for the driver instance.
- **value** – Pointer to a variable where pin values will be stored.

Return values

- **0** – If successful.
- **-EIO** – I/O error when accessing an external GPIO chip.
- **-EWOULDBLOCK** – if operation would block.

static inline int **gpio_port_get**(const struct *device* *port, *gpio_port_value_t* *value)

Get logical level of all input pins in a port.

Get logical level of an input pin taking into account GPIO_ACTIVE_LOW flag. If pin is configured as Active High, a low physical level will be interpreted as logical value 0. If pin is configured as Active Low, a low physical level will be interpreted as logical value 1.

Value of a pin with index n will be represented by bit n in the returned port value.

Parameters

- **port** – Pointer to the device structure for the driver instance.
- **value** – Pointer to a variable where pin values will be stored.

Return values

- **0** – If successful.
- **-EIO** – I/O error when accessing an external GPIO chip.
- **-EWOULDBLOCK** – if operation would block.

int **gpio_port_set_masked_raw**(const struct *device* *port, *gpio_port_pins_t* mask, *gpio_port_value_t* value)

Set physical level of output pins in a port.

Writing value 0 to the pin will set it to a low physical level. Writing value 1 will set it to a high physical level. This function ignores GPIO_ACTIVE_LOW flag.

Pin with index n is represented by bit n in mask and value parameter.

Parameters

- **port** – Pointer to the device structure for the driver instance.
- **mask** – Mask indicating which pins will be modified.
- **value** – Value assigned to the output pins.

Return values

- `0` – If successful.
- `-EIO` – I/O error when accessing an external GPIO chip.
- `-EWOULDBLOCK` – if operation would block.

static inline int **gpio_port_set_masked**(const struct *device* *port, *gpio_port_pins_t* mask, *gpio_port_value_t* value)

Set logical level of output pins in a port.

Set logical level of an output pin taking into account GPIO_ACTIVE_LOW flag. Value 0 sets the pin in logical 0 / inactive state. Value 1 sets the pin in logical 1 / active state. If pin is configured as Active High, the default, setting it in inactive state will force the pin to a low physical level. If pin is configured as Active Low, setting it in inactive state will force the pin to a high physical level.

Pin with index n is represented by bit n in mask and value parameter.

Parameters

- `port` – Pointer to the device structure for the driver instance.
- `mask` – Mask indicating which pins will be modified.
- `value` – Value assigned to the output pins.

Return values

- `0` – If successful.
- `-EIO` – I/O error when accessing an external GPIO chip.
- `-EWOULDBLOCK` – if operation would block.

int **gpio_port_set_bits_raw**(const struct *device* *port, *gpio_port_pins_t* pins)

Set physical level of selected output pins to high.

Parameters

- `port` – Pointer to the device structure for the driver instance.
- `pins` – Value indicating which pins will be modified.

Return values

- `0` – If successful.
- `-EIO` – I/O error when accessing an external GPIO chip.
- `-EWOULDBLOCK` – if operation would block.

static inline int **gpio_port_set_bits**(const struct *device* *port, *gpio_port_pins_t* pins)

Set logical level of selected output pins to active.

Parameters

- `port` – Pointer to the device structure for the driver instance.
- `pins` – Value indicating which pins will be modified.

Return values

- `0` – If successful.
- `-EIO` – I/O error when accessing an external GPIO chip.
- `-EWOULDBLOCK` – if operation would block.

int **gpio_port_clear_bits_raw**(const struct *device* *port, *gpio_port_pins_t* pins)

Set physical level of selected output pins to low.

Parameters

- **port** – Pointer to the device structure for the driver instance.
- **pins** – Value indicating which pins will be modified.

Return values

- **0** – If successful.
- **-EIO** – I/O error when accessing an external GPIO chip.
- **-EWOULDBLOCK** – if operation would block.

static inline int **gpio_port_clear_bits**(const struct *device* *port, *gpio_port_pins_t* pins)

Set logical level of selected output pins to inactive.

Parameters

- **port** – Pointer to the device structure for the driver instance.
- **pins** – Value indicating which pins will be modified.

Return values

- **0** – If successful.
- **-EIO** – I/O error when accessing an external GPIO chip.
- **-EWOULDBLOCK** – if operation would block.

int **gpio_port_toggle_bits**(const struct *device* *port, *gpio_port_pins_t* pins)

Toggle level of selected output pins.

Parameters

- **port** – Pointer to the device structure for the driver instance.
- **pins** – Value indicating which pins will be modified.

Return values

- **0** – If successful.
- **-EIO** – I/O error when accessing an external GPIO chip.
- **-EWOULDBLOCK** – if operation would block.

static inline int **gpio_port_set_clr_bits_raw**(const struct *device* *port, *gpio_port_pins_t* set_pins, *gpio_port_pins_t* clear_pins)

Set physical level of selected output pins.

Parameters

- **port** – Pointer to the device structure for the driver instance.
- **set_pins** – Value indicating which pins will be set to high.
- **clear_pins** – Value indicating which pins will be set to low.

Return values

- **0** – If successful.
- **-EIO** – I/O error when accessing an external GPIO chip.
- **-EWOULDBLOCK** – if operation would block.

static inline int **gpio_port_set_clr_bits**(const struct *device* *port, *gpio_port_pins_t* set_pins, *gpio_port_pins_t* clear_pins)

Set logical level of selected output pins.

Parameters

- **port** – Pointer to the device structure for the driver instance.

- **set_pins** – Value indicating which pins will be set to active.
- **clear_pins** – Value indicating which pins will be set to inactive.

Return values

- **0** – If successful.
- **-EIO** – I/O error when accessing an external GPIO chip.
- **-EWOULDBLOCK** – if operation would block.

static inline int **gpio_pin_get_raw**(const struct *device* *port, *gpio_pin_t* pin)

Get physical level of an input pin.

A low physical level on the pin will be interpreted as value 0. A high physical level will be interpreted as value 1. This function ignores GPIO_ACTIVE_LOW flag.

Parameters

- **port** – Pointer to the device structure for the driver instance.
- **pin** – Pin number.

Return values

- **1** – If pin physical level is high.
- **0** – If pin physical level is low.
- **-EIO** – I/O error when accessing an external GPIO chip.
- **-EWOULDBLOCK** – if operation would block.

static inline int **gpio_pin_get**(const struct *device* *port, *gpio_pin_t* pin)

Get logical level of an input pin.

Get logical level of an input pin taking into account GPIO_ACTIVE_LOW flag. If pin is configured as Active High, a low physical level will be interpreted as logical value 0. If pin is configured as Active Low, a low physical level will be interpreted as logical value 1.

Note: If pin is configured as Active High, the default, *gpio_pin_get()* function is equivalent to *gpio_pin_get_raw()*.

Parameters

- **port** – Pointer to the device structure for the driver instance.
- **pin** – Pin number.

Return values

- **1** – If pin logical value is 1 / active.
- **0** – If pin logical value is 0 / inactive.
- **-EIO** – I/O error when accessing an external GPIO chip.
- **-EWOULDBLOCK** – if operation would block.

static inline int **gpio_pin_get_dt**(const struct *gpio_dt_spec* *spec)

Get logical level of an input pin from a *gpio_dt_spec*.

This is equivalent to:

```
gpio_pin_get(spec->port, spec->pin);
```

Parameters

- **spec** – GPIO specification from devicetree

Returns
a value from *gpio_pin_get()*

static inline int **gpio_pin_set_raw**(const struct *device* *port, *gpio_pin_t* pin, int value)

Set physical level of an output pin.

Writing value 0 to the pin will set it to a low physical level. Writing any value other than 0 will set it to a high physical level. This function ignores GPIO_ACTIVE_LOW flag.

Parameters

- **port** – Pointer to the device structure for the driver instance.
- **pin** – Pin number.
- **value** – Value assigned to the pin.

Return values

- **0** – If successful.
- **-EIO** – I/O error when accessing an external GPIO chip.
- **-EWOULDBLOCK** – if operation would block.

static inline int **gpio_pin_set**(const struct *device* *port, *gpio_pin_t* pin, int value)

Set logical level of an output pin.

Set logical level of an output pin taking into account GPIO_ACTIVE_LOW flag. Value 0 sets the pin in logical 0 / inactive state. Any value other than 0 sets the pin in logical 1 / active state. If pin is configured as Active High, the default, setting it in inactive state will force the pin to a low physical level. If pin is configured as Active Low, setting it in inactive state will force the pin to a high physical level.

Note: If pin is configured as Active High, *gpio_pin_set()* function is equivalent to *gpio_pin_set_raw()*.

Parameters

- **port** – Pointer to the device structure for the driver instance.
- **pin** – Pin number.
- **value** – Value assigned to the pin.

Return values

- **0** – If successful.
- **-EIO** – I/O error when accessing an external GPIO chip.
- **-EWOULDBLOCK** – if operation would block.

static inline int **gpio_pin_set_dt**(const struct *gpio_dt_spec* *spec, int value)

Set logical level of a output pin from a *gpio_dt_spec*.

This is equivalent to:

```
gpio_pin_set(spec->port, spec->pin, value);
```

Parameters

- **spec** – GPIO specification from devicetree
- **value** – Value assigned to the pin.

Returns
a value from *gpio_pin_set()*

static inline int **gpio_pin_toggle**(const struct *device* *port, *gpio_pin_t* pin)

Toggle pin level.

Parameters

- **port** – Pointer to the device structure for the driver instance.
- **pin** – Pin number.

Return values

- **0** – If successful.
- **-EIO** – I/O error when accessing an external GPIO chip.
- **-EWOULDBLOCK** – if operation would block.

static inline int **gpio_pin_toggle_dt**(const struct *gpio_dt_spec* *spec)

Toggle pin level from a *gpio_dt_spec*.

This is equivalent to:

```
gpio_pin_toggle(spec->port, spec->pin);
```

Parameters

- **spec** – GPIO specification from devicetree

Returns

a value from *gpio_pin_toggle()*

static inline void **gpio_init_callback**(struct *gpio_callback* *callback, *gpio_callback_handler_t* handler, *gpio_port_pins_t* pin_mask)

Helper to initialize a struct *gpio_callback* properly.

Parameters

- **callback** – A valid Application's callback structure pointer.
- **handler** – A valid handler function pointer.
- **pin_mask** – A bit mask of relevant pins for the handler

static inline int **gpio_add_callback**(const struct *device* *port, struct *gpio_callback* *callback)

Add an application callback.

Note: enables to add as many callback as needed on the same port.

Note

Callbacks may be added to the device from within a callback handler invocation, but whether they are invoked for the current GPIO event is not specified.

Parameters

- **port** – Pointer to the device structure for the driver instance.
- **callback** – A valid Application's callback structure pointer.

Return values

- **0** – If successful

- **-ENOSYS** – If driver does not implement the operation
- **-errno** – Other negative errno code on failure.

static inline int **gpio_add_callback_dt**(const struct *gpio_dt_spec* *spec, struct *gpio_callback* *callback)

Add an application callback.

This is equivalent to:

```
gpio_add_callback(spec->port, callback);
```

Parameters

- **spec** – GPIO specification from devicetree.
- **callback** – A valid application's callback structure pointer.

Returns
a value from *gpio_add_callback()*.

static inline int **gpio_remove_callback**(const struct *device* *port, struct *gpio_callback* *callback)

Remove an application callback.

Note: enables to remove as many callbacks as added through *gpio_add_callback()*.

Warning

It is explicitly permitted, within a callback handler, to remove the registration for the callback that is running, i.e. `callback`. Attempts to remove other registrations on the same device may result in undefined behavior, including failure to invoke callbacks that remain registered and unintended invocation of removed callbacks.

Parameters

- **port** – Pointer to the device structure for the driver instance.
- **callback** – A valid application's callback structure pointer.

Return values

- **0** – If successful
- **-ENOSYS** – If driver does not implement the operation
- **-errno** – Other negative errno code on failure.

static inline int **gpio_remove_callback_dt**(const struct *gpio_dt_spec* *spec, struct *gpio_callback* *callback)

Remove an application callback.

This is equivalent to:

```
gpio_remove_callback(spec->port, callback);
```

Parameters

- **spec** – GPIO specification from devicetree.
- **callback** – A valid application's callback structure pointer.

Returns
a value from *gpio_remove_callback()*.

int **gpio_get_pending_int**(const struct *device* *dev)

Function to get pending interrupts.

The purpose of this function is to return the interrupt status register for the device. This is especially useful when waking up from low power states to check the wake up source.

Parameters

- **dev** – Pointer to the device structure for the driver instance.

Return values

- **status** – != 0 if at least one gpio interrupt is pending.
- **0** – if no gpio interrupt is pending.
- **-ENOSYS** – If driver does not implement the operation

struct **gpio_dt_spec**

#include <gpio.h> Container for GPIO pin information specified in devicetree.

This type contains a pointer to a GPIO device, pin number for a pin controlled by that device, and the subset of pin configuration flags which may be given in devicetree.

See also

GPIO_DT_SPEC_GET_BY_IDX

See also

GPIO_DT_SPEC_GET_BY_IDX_OR

See also

GPIO_DT_SPEC_GET

See also

GPIO_DT_SPEC_GET_OR

Public Members

const struct *device* ***port**

GPIO device controlling the pin.

gpio_pin_t **pin**

The pin's number on the device.

gpio_dt_flags_t **dt_flags**

The pin's configuration flags as specified in devicetree.

struct **gpio_driver_config**

#include <*gpio.h*> This structure is common to all GPIO drivers and is expected to be the first element in the object pointed to by the config field in the device structure.

Public Members

gpio_port_pins_t **port_pin_mask**

Mask identifying pins supported by the controller.

Initialization of this mask is the responsibility of device instance generation in the driver.

struct **gpio_driver_data**

#include <*gpio.h*> This structure is common to all GPIO drivers and is expected to be the first element in the driver's struct driver_data declaration.

Public Members

gpio_port_pins_t **invert**

Mask identifying pins that are configured as active low.

Management of this mask is the responsibility of the wrapper functions in this header.

struct **gpio_callback**

#include <*gpio.h*> GPIO callback structure.

Used to register a callback in the driver instance callback list. As many callbacks as needed can be added as long as each of them are unique pointers of struct *gpio_callback*. Beware such structure should not be allocated on stack.

Note: To help setting it, see *gpio_init_callback()* below

Public Members

sys_snode_t **node**

This is meant to be used in the driver and the user should not mess with it (see drivers/gpio/gpio_utils.h)

gpio_callback_handler_t **handler**

Actual callback function being called when relevant.

gpio_port_pins_t **pin_mask**

A mask of pins the callback is interested in, if 0 the callback will never be called.

Such pin_mask can be modified whenever necessary by the owner, and thus will affect the handler being called or not. The selected pins must be configured to trigger an interrupt.

7.6.23 Hardware Information

Overview

The HW Info API provides access to hardware information such as device identifiers and reset cause flags.

Reset cause flags can be used to determine why the device was reset; for example due to a watchdog timeout or due to power cycling. Different devices support different subset of flags. Use *hwinfo_get_supported_reset_cause()* to retrieve the flags that are supported by that device.

Configuration Options

Related configuration options:

- CONFIG_HWINFO

API Reference

group hwinfo_interface

Hardware Information Interface.

Since
1.14

Version
1.0.0

Reset cause flags

RESET_PIN
External pin.

RESET_SOFTWARE
Software reset.

RESET_BROWNOUT
Brownout (drop in voltage)

RESET_POR
Power-on reset (POR)

RESET_WATCHDOG
Watchdog timer expiration.

RESET_DEBUG
Debug event.

`RESET_SECURITY`
: Security violation.

`RESET_LOW_POWER_WAKE`
: Waking up from low power mode.

`RESET_CPU_LOCKUP`
: CPU lock-up detected.

`RESET_PARITY`
: Parity error.

`RESET_PLL`
: PLL error.

`RESET_CLOCK`
: Clock error.

`RESET_HARDWARE`
: Hardware reset.

`RESET_USER`
: User reset.

`RESET_TEMPERATURE`
: Temperature reset.

Functions

ssize_t **hwinfo_get_device_id**(uint8_t *buffer, size_t length)

Copy the device id to a buffer.

This routine copies "length" number of bytes of the device ID to the buffer. If the device ID is smaller than length, the rest of the buffer is left unchanged. The ID depends on the hardware and is not guaranteed unique.

Drivers are responsible for ensuring that the ID data structure is a sequence of bytes. The returned ID value is not supposed to be interpreted based on vendor-specific assumptions of byte order. It should express the identifier as a raw byte sequence, doing any endian conversion necessary so that a hex representation of the bytes produces the intended serial number.

Parameters

- `buffer` – Buffer to write the ID to.
- `length` – Max length of the buffer.

Return values

- `size` – of the device ID copied.
- `-ENOSYS` – if there is no implementation for the particular device.
- `any` – negative value on driver specific errors.

int **hwinfo_get_device_eui64**(uint8_t *buffer)

Copy the device EUI64 to a buffer.

This routine copies the device EUI64 (8 bytes) to the buffer. The EUI64 depends on the hardware and is guaranteed unique.

Parameters

- **buffer** – Buffer of 8 bytes to write the ID to.

Return values

- **zero** – if successful.
- **-ENOSYS** – if there is no implementation for the particular device.
- **any** – negative value on driver specific errors.

int **hwinfo_get_reset_cause**(uint32_t *cause)

Retrieve cause of device reset.

This routine retrieves the flags that indicate why the device was reset.

On some platforms the reset cause flags accumulate between successive resets and this routine may return multiple flags indicating all reset causes since the device was powered on. If you need to retrieve the cause only for the most recent reset call `hwinfo_clear_reset_cause` after calling this routine to clear the hardware flags before the next reset event.

Successive calls to this routine will return the same value, unless `hwinfo_clear_reset_cause` has been called.

Parameters

- **cause** – OR'd *reset cause* flags

Return values

- **zero** – if successful.
- **-ENOSYS** – if there is no implementation for the particular device.
- **any** – negative value on driver specific errors.

int **hwinfo_clear_reset_cause**(void)

Clear cause of device reset.

Clears reset cause flags.

Return values

- **zero** – if successful.
- **-ENOSYS** – if there is no implementation for the particular device.
- **any** – negative value on driver specific errors.

int **hwinfo_get_supported_reset_cause**(uint32_t *supported)

Get supported reset cause flags.

Retrieves all `reset_cause` flags that are supported by this device.

Parameters

- **supported** – OR'd *reset cause* flags that are supported

Return values

- **zero** – if successful.

- `-ENOSYS` – if there is no implementation for the particular device.
- `any` – negative value on driver specific errors.

7.6.24 I2C EEPROM Target

Overview

API Reference

group `i2c_eeprom_target_api`

I2C EEPROM Target Driver API.

Since
1.13

Version
1.0.0

Functions

int `eeprom_target_program`(const struct *device* *dev, const uint8_t *eeprom_data, unsigned int length)

Program memory of the virtual EEPROM.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `eeprom_data` – Pointer of data to program into the virtual eeprom memory
- `length` – Length of data to program into the virtual eeprom memory

Return values

- `0` – If successful.
- `-EINVAL` – Invalid data size

int `eeprom_target_read`(const struct *device* *dev, uint8_t *eeprom_data, unsigned int offset)

Read single byte of virtual EEPROM memory.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `eeprom_data` – Pointer of byte where to store the virtual eeprom memory
- `offset` – Offset into EEPROM memory where to read the byte

Return values

- `0` – If successful.
- `-EINVAL` – Invalid data pointer or offset

int **eeprom_target_set_addr**(const struct *device* *dev, uint8_t addr)

Change the address of eeprom target at runtime.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **addr** – New address to assign to the eeprom target device

Return values

- **0** – Is successful
- **-EINVAL** – If parameters are invalid
- **-EIO** – General input / output error during i2c_taget_register
- **-ENOSYS** – If target mode is not implemented

7.6.25 Improved Inter-Integrated Circuit (I3C) Bus

I3C (Improved Inter-Integrated Circuit) is a two-signal shared peripheral interface bus. Devices on the bus can operate in two roles: as a "controller" that initiates transactions and controls the clock, or as a "target" that responds to transaction commands.

Currently, the API is based on I3C Specification version 1.1.1.

- *I3C Controller API*
  - *In-Band Interrupt (IBI)*
  - *Device Tree*
  - *Device Drivers for I3C Devices*
  - *I^2C Devices under I3C Bus*
- *Configuration Options*
- *API Reference*

I3C Controller API

Zephyr's I3C controller API is used when an I3C controller controls the bus, in particularly the start and stop conditions and the clock. This is the most common mode, used to interact with I3C target devices such as sensors.

Due to the nature of the I3C, there are devices on the bus where they may not have addresses when powered on. Therefore, an additional dynamic address assignment needs to be carried out by the I3C controller. Because of this, the controller needs to maintain separate structures to keep track of device status. This can be done at build time, for example, by creating arrays of device descriptors for both I3C and I^2C devices:

```
static struct i3c_device_desc i3c_device_array[] = I3C_DEVICE_ARRAY_DT_INST(inst);
static struct i3c_i2c_device_desc i2c_device_array[] = I3C_I2C_DEVICE_ARRAY_DT_INST(inst);
```

The macros `I3C_DEVICE_ARRAY_DT_INST` and `I3C_I2C_DEVICE_ARRAY_DT_INST` are helper macros to aid in create arrays of device descriptors corresponding to the devicetree nodes under the I3C controller.

Here is a list of generic steps for initializing the I3C controller and the I3C bus inside the device driver initialization function:

1. Initialize the data structure of the I3C controller device driver instance. The usual device defining macros such as *DEVICE_DT_INST_DEFINE* can be used, and the initialization function provided as a parameter to the macro.
    - The *i3c_addr_slots* and *i3c_dev_list* are structures to aid in address assignments and device list management. If this is being used, this struct needs to be initialized by calling *i3c_addr_slots_init()*. These two structures can also be used with various helper functions.
    - Initialize the device descriptors if needed by the controller driver.
2. Initialize the hardware, including but not limited to:
    - Setup pin mux and directions.
    - Setup the clock for the controller.
    - Power on the hardware.
    - Configure the hardware (e.g. SCL clock frequency).
3. Perform bus initialization. There is a generic helper function, *i3c_bus_init()*, which performs the following steps. This function can be used if the controller does not require any special handling during bus initialization.
    1. Do RSTDAA to reset dynamic addresses of connected devices. If any connected devices have already been assigned an address, the bookkeeping data structures do not have records of these, for example, at power-on. So it is a good idea to reset and assign them new addresses.
    2. Do DISEC to disable any events from devices.
    3. Do SETDASA to use static addresses as dynamic address if so desired.
        - SETAASA may not be supported for all connected devices to assign static addresses as dynamic addresses.
        - BCR and DCR need to be obtained separately to populate the relevant fields in the I3C target device descriptor struct.
    4. Do ENTDAA to start dynamic address assignment, if there are still devices without addresses.
        - If there is a device waiting for address, it will send its Provisioned ID, BCR, and DCR back. Match the received Provisioned ID to the list of registered I3C devices.
            - If there is a match, assign an address (either from the stated static address if SETDASA has not been done, or use a free address).
                * Also, set the BCR and DCR fields in the device descriptor struct.
            - If there is no match, depending on policy, it can be assigned a free address, or the device driver can stop the assignment process and errors out.
                * Note that the I3C API requires device descriptor to function. A device without a device descriptor cannot be accessed through the API.
        - This step can be skipped if there is no connected devices requiring DAA.
    5. These are optional but highly recommended:
        - Do GETMRL and GETMWL to get maximum read/write length.
        - Do GETMXDS to get maximum read/write speed and maximum read turnaround time.
        - The helper function, *i3c_bus_init()*, would retrieve basic device information such as BCR, DCR, MRL and MWL.
    6. Do ENEC to re-enable events from devices.

- The helper function, *i3c_bus_init()*, only re-enables hot-join events. IBI event should only be enabled when enabling IBI of a device.

In-Band Interrupt (IBI) If a target device can generate In-Band Interrupt (IBI), the controller needs to be made aware of it.

- `i3c_ibi_enable()` to enable IBI of a target device.
  - Some controller hardware have IBI slots which need to be programmed so that the controller can recognize incoming IBIs from a particular target device.
    - If the hardware has IBI slots, `i3c_ibi_enable()` needs to program those IBI slots.
    - Note that there are usually limited IBI slots on the controller so this operation may fail.
  - The implementation in driver should also send the `ENEC` command to enable interrupt of this target device.
- `i3c_ibi_disable()` to disable IBI of a target device.
  - If controller hardware makes use of IBI slots, this will remove description of the target device from the slots.
  - The implementation in driver should also send the `DISEC` command to disable interrupt of this target device.

Device Tree Here is an example for defining a I3C controller in device tree:

```
i3c0: i3c@10000 {
        compatible = "vendor,i3c";

        #address-cells = < 0x3 >;
        #size-cells = < 0x0 >;

        reg = < 0x10000 0x1000 >;
        interrupts = < 0x1F 0x0 >;

        pinctrl-0 = < &pinmux-i3c >;
        pinctrl-names = "default";

        i2c-scl-hz = < 400000 >;

        i3c-scl-hz = < 12000000 >;

        status = "okay";

        i3c-dev0: i3c-dev0@420000ABCD12345678 {
                compatible = "vendor,i3c-dev";

                reg = < 0x42 0xABCD 0x12345678 >;

                status = "okay";
        };

        i2c-dev0: i2c-dev0@380000000000000050 {
                compatible = "vendor-i2c-dev";

                reg = < 0x38 0x0 0x50 >;

                status = "okay";
        };
};
```

I3C Devices For I3C devices, the `reg` property has 3 elements:

- The first one is the static address of the device.
  - Can be zero if static address is not used. Address will be assigned during DAA (Dynamic Address Assignment).
  - If non-zero and property `assigned-address` is not set, this will be the address of the device after SETDASA (Set Dynamic Address from Static Address) is issued.
- Second element is the upper 16-bit of the Provisioned ID (PID) which contains the manufacturer ID left-shifted by 1. This is the bits 33-47 (zero-based) of the 48-bit Provisioned ID.
- Third element contains the lower 32-bit of the Provisioned ID which is a combination of the part ID (left-shifted by 16, bits 16-31 of the PID) and the instance ID (left-shifted by 12, bits 12-15 of the PID).

Note that the unit-address (the part after `@`) must match the `reg` property fully where each element is treated as 32-bit integer, combining to form a 96-bit integer. This is required for properly generating device tree macros.

I^{2}C Devices For I^{2}C devices where the device driver has support for working under I3C bus, the device node can be described as a child of the I3C controller. If the device driver is written to only work with I^{2}C controllers, define the node under the I^{2}C virtual controller as described below. Otherwise, the `reg` property, similar to I3C devices, has 3 elements:

- The first one is the static address of the device. This must be a valid address as I^{2}C devices do not support dynamic address assignment.
- Second element is always zero.
  - This is used by various helper macros to determine whether the device tree entry corresponds to a I^{2}C device.
- Third element is the LVR (Legacy Virtual Register):
  - bit[31:8] are unused.
  - bit[7:5] are the I^{2}C device index:
    - Index `0`
      - I3C device has a 50 ns spike filter where it is not affected by high frequency on SCL.
    - Index `1`
      - I^{2}C device does not have a 50 ns spike filter but can work with high frequency on SCL.
    - Index `2`
      - I3C device does not have a 50 ns spike filter and cannot work with high frequency on SCL.
  - bit[4] is the I^{2}C mode indicator:
    - `0` is FM+ mode.
    - `1` is FM mode.

Similar to I3C devices, the unit-address must match the `reg` property fully where each element is treated as 32-bit integer, combining to form a 96-bit integer.

Device Drivers for I3C Devices All of the transfer functions of I3C controller API require the use of device descriptors, *i3c_device_desc*. This struct contains runtime information about a I3C device, such as, its dynamic address, BCR, DCR, MRL and MWL. Therefore, the device driver of a I3C device should grab a pointer to this device descriptor from the controller using *i3c_device_find()*. This function takes an ID parameter of type *i3c_device_id* for matching. The returned pointer can then be used in subsequent API calls to the controller.

I²C Devices under I3C Bus Since I3C is backward compatible with I²C, the I3C controller API can accommodate I2C API calls without modifications if the controller device driver implements the I2C API. This has the advantage of using existing I2C devices without any modifications to their device drivers. However, since the I3C controller API works on device descriptors, any calls to I2C API will need to look up the corresponding device descriptor from the I2C device address. This adds a bit of processing cost to any I2C API calls.

On the other hand, a device driver can be extended to utilize native I2C device support via the I3C controller API. During device initialization, `i3c_i2c_device_find()` needs to be called to retrieve the pointer to the device descriptor. This pointer can be used in subsequent API calls.

Note that, with either methods mentioned above, the devicetree node of the I2C device must be declared according to I3C standard:

The I²C virtual controller device driver provides a way to interface I²C devices on the I3C bus where the associated device drivers can be used as-is without modifications. This requires adding an intermediate node in the device tree:

```
i3c0: i3c@10000 {
        <... I3C controller related properties ...>
        <... Nodes of I3C devices, if any ...>

        i2c-dev0: i2c-dev0@42000000000000050 {
                compatible = "vendor-i2c-dev";

                reg = < 0x42 0x0 0x50 >;

                status = "okay";
        };
};
```

Configuration Options

Related configuration options:

- CONFIG_I3C
- CONFIG_I3C_USE_GROUP_ADDR
- CONFIG_I3C_USE_IBI
- CONFIG_I3C_IBI_MAX_PAYLOAD_SIZE
- CONFIG_I3C_CONTROLLER_INIT_PRIORITY

API Reference

group `i3c_interface`

I3C Interface.

Since
3.2

Version
0.1.0

Bus Characteristic Register (BCR)

- BCR[7:6]: Device Role
  - 0: I3C Target
  - 1: I3C Controller capable
  - 2: Reserved
  - 3: Reserved
- BCR[5]: Advanced Capabilities
  - 0: Does not support optional advanced capabilities.
  - 1: Supports optional advanced capabilities which can be viewed via GETCAPS CCC.
- BCR[4]: Virtual Target Support
  - 0: Is not a virtual target.
  - 1: Is a virtual target.
- BCR[3]: Offline Capable
  - 0: Will always response to I3C commands.
  - 1: Will not always response to I3C commands.
- BCR[2]: IBI Payload
  - 0: No data bytes following the accepted IBI.
  - 1: One data byte (MDB, Mandatory Data Byte) follows the accepted IBI. Additional data bytes may also follows.
- BCR[1]: IBI Request Capable
  - 0: Not capable
  - 1: Capable
- BCR[0]: Max Data Speed Limitation
  - 0: No Limitation
  - 1: Limitation obtained via GETMXDS CCC.

`I3C_BCR_MAX_DATA_SPEED_LIMIT`
Max Data Speed Limitation bit.

0 - No Limitation. 1 - Limitation obtained via GETMXDS CCC.

`I3C_BCR_IBI_REQUEST_CAPABLE`
IBI Request Capable bit.

I3C_BCR_IBI_PAYLOAD_HAS_DATA_BYTE

IBI Payload bit.

0 - No data bytes following the accepted IBI. 1 - One data byte (MDB, Mandatory Data Byte) follows the accepted IBI. Additional data bytes may also follows.

I3C_BCR_OFFLINE_CAPABLE

Offline Capable bit.

0 - Will always respond to I3C commands. 1 - Will not always respond to I3C commands.

I3C_BCR_VIRTUAL_TARGET

Virtual Target Support bit.

0 - Is not a virtual target. 1 - Is a virtual target.

I3C_BCR_ADV_CAPABILITIES

Advanced Capabilities bit.

0 - Does not support optional advanced capabilities. 1 - Supports optional advanced capabilities which can be viewed via GETCAPS CCC.

I3C_BCR_DEVICE_ROLE_I3C_TARGET

Device Role - I3C Target.

I3C_BCR_DEVICE_ROLE_I3C_CONTROLLER_CAPABLE

Device Role - I3C Controller Capable.

I3C_BCR_DEVICE_ROLE_SHIFT

Device Role bit shift value.

I3C_BCR_DEVICE_ROLE_MASK

Device Role bit shift mask.

I3C_BCR_DEVICE_ROLE(bcr)

Device Role.

Obtain Device Role value from the BCR value obtained via GETBCR.

Parameters

- bcr – BCR value

Legacy Virtual Register (LVR)

Legacy Virtual Register (LVR)

- LVR[7:5]: I2C device index:
  - 0: I2C device has a 50 ns spike filter where it is not affected by high frequency on SCL.
  - 1: I2C device does not have a 50 ns spike filter but can work with high frequency on SCL.
  - 2: I2C device does not have a 50 ns spike filter and cannot work with high frequency on SCL.

- LVR[4]: I2C mode indicator:
  - 0: FM+ mode
  - 1: FM mode
- LVR[3:0]: Reserved.

`I3C_LVR_I2C_FM_PLUS_MODE`
I2C FM+ Mode.

`I3C_LVR_I2C_FM_MODE`
I2C FM Mode.

`I3C_LVR_I2C_MODE_SHIFT`
I2C Mode Indicator bit shift value.

`I3C_LVR_I2C_MODE_MASK`
I2C Mode Indicator bitmask.

`I3C_LVR_I2C_MODE`(lvr)
I2C Mode.

Obtain I2C Mode value from the LVR value.

Parameters

- `lvr` – LVR value

`I3C_LVR_I2C_DEV_IDX_0`
I2C Device Index 0.

I2C device has a 50 ns spike filter where it is not affected by high frequency on SCL.

`I3C_LVR_I2C_DEV_IDX_1`
I2C Device Index 1.

I2C device does not have a 50 ns spike filter but can work with high frequency on SCL.

`I3C_LVR_I2C_DEV_IDX_2`
I2C Device Index 2.

I2C device does not have a 50 ns spike filter and cannot work with high frequency on SCL.

`I3C_LVR_I2C_DEV_IDX_SHIFT`
I2C Device Index bit shift value.

`I3C_LVR_I2C_DEV_IDX_MASK`
I2C Device Index bitmask.

`I3C_LVR_I2C_DEV_IDX`(lvr)
I2C Device Index.

Obtain I2C Device Index value from the LVR value.

Parameters

- `lvr` – LVR value

Defines

I3C_DEVICE_ID(pid)

Structure initializer for *i3c_device_id* from PID.

This helper macro expands to a static initializer for a `struct i3c_device_id` by populating the PID (Provisioned ID) field.

Parameters

- `pid` – Provisioned ID.

Enums

enum `i3c_bus_mode`

I3C bus mode.

Values:

enumerator `I3C_BUS_MODE_PURE`

Only I3C devices are on the bus.

enumerator `I3C_BUS_MODE_MIXED_FAST`

Both I3C and legacy I2C devices are on the bus.

The I2C devices have 50ns spike filter on SCL.

enumerator `I3C_BUS_MODE_MIXED_LIMITED`

Both I3C and legacy I2C devices are on the bus.

The I2C devices do not have 50ns spike filter on SCL and can tolerate maximum SDR SCL clock frequency.

enumerator `I3C_BUS_MODE_MIXED_SLOW`

Both I3C and legacy I2C devices are on the bus.

The I2C devices do not have 50ns spike filter on SCL but cannot tolerate maximum SDR SCL clock frequency.

enumerator `I3C_BUS_MODE_MAX` = *I3C_BUS_MODE_MIXED_SLOW*

enumerator `I3C_BUS_MODE_INVALID`

enum `i3c_i2c_speed_type`

I2C bus speed under I3C bus.

Only FM and FM+ modes are supported for I2C devices under I3C bus.

Values:

enumerator `I3C_I2C_SPEED_FM`

I2C FM mode.

enumerator `I3C_I2C_SPEED_FMPLUS`

I2C FM+ mode.

enumerator **I3C_I2C_SPEED_MAX** = *I3C_I2C_SPEED_FMPLUS*

enumerator **I3C_I2C_SPEED_INVALID**

enum **i3c_data_rate**

I3C data rate.

I3C data transfer rate defined by the I3C specification.

Values:

enumerator **I3C_DATA_RATE_SDR**

Single Data Rate messaging.

enumerator **I3C_DATA_RATE_HDR_DDR**

High Data Rate - Double Data Rate messaging.

enumerator **I3C_DATA_RATE_HDR_TSL**

High Data Rate - Ternary Symbol Legacy-inclusive-Bus.

enumerator **I3C_DATA_RATE_HDR_TSP**

High Data Rate - Ternary Symbol for Pure Bus.

enumerator **I3C_DATA_RATE_HDR_BT**

High Data Rate - Bulk Transport.

enumerator **I3C_DATA_RATE_MAX** = *I3C_DATA_RATE_HDR_BT*

enumerator **I3C_DATA_RATE_INVALID**

enum **i3c_sdr_controller_error_codes**

I3C SDR Controller Error Codes.

These are error codes defined by the I3C specification.

`I3C_ERROR_CE_UNKNOWN` and `I3C_ERROR_CE_NONE` are not official error codes according to the specification. These are there simply to aid in error handling during interactions with the I3C drivers and subsystem.

Values:

enumerator **I3C_ERROR_CE0**

Transaction after sending CCC.

enumerator **I3C_ERROR_CE1**

Monitoring Error.

enumerator **I3C_ERROR_CE2**

No response to broadcast address (0x7E)

enumerator **I3C_ERROR_CE3**

Failed Controller Handoff.

enumerator I3C_ERROR_CE_UNKNOWN

Unknown error (not official error code)

enumerator I3C_ERROR_CE_NONE

No error (not official error code)

enumerator I3C_ERROR_CE_MAX = *I3C_ERROR_CE_UNKNOWN*

enumerator I3C_ERROR_CE_INVALID

enum i3c_sdr_target_error_codes

I3C SDR Target Error Codes.

These are error codes defined by the I3C specification.

I3C_ERROR_TE_UNKNOWN and I3C_ERROR_TE_NONE are not official error codes according to the specification. These are there simply to aid in error handling during interactions with the I3C drivers and subsystem.

Values:

enumerator I3C_ERROR_TE0

Invalid Broadcast Address or Dynamic Address after DA assignment.

enumerator I3C_ERROR_TE1

CCC Code.

enumerator I3C_ERROR_TE2

Write Data.

enumerator I3C_ERROR_TE3

Assigned Address during Dynamic Address Arbitration.

enumerator I3C_ERROR_TE4

0x7E/R missing after RESTART during Dynamic Address Arbitration

enumerator I3C_ERROR_TE5

Transaction after detecting CCC.

enumerator I3C_ERROR_TE6

Monitoring Error.

enumerator I3C_ERROR_DBR

Dead Bus Recovery.

enumerator I3C_ERROR_TE_UNKNOWN

Unknown error (not official error code)

enumerator I3C_ERROR_TE_NONE

No error (not official error code)

enumerator I3C_ERROR_TE_MAX = *I3C_ERROR_TE_UNKNOWN*

enumerator I3C_ERROR_TE_INVALID

enum i3c_config_type

Type of configuration being passed to configure function.

Values:

enumerator I3C_CONFIG_CONTROLLER

enumerator I3C_CONFIG_TARGET

enumerator I3C_CONFIG_CUSTOM

Functions

struct *i3c_device_desc* *i3c_dev_list_find(const struct *i3c_dev_list* *dev_list, const struct *i3c_device_id* *id)

Find a I3C target device descriptor by ID.

This finds the I3C target device descriptor in the device list matching the provided ID struct (id).

Parameters

- dev_list – Pointer to the device list struct.
- id – Pointer to I3C device ID struct.

Returns

Pointer to the I3C target device descriptor, or NULL if none is found.

struct *i3c_device_desc* *i3c_dev_list_i3c_addr_find(struct *i3c_dev_attached_list* *dev_list, uint8_t addr)

Find a I3C target device descriptor by dynamic address.

This finds the I3C target device descriptor in the attached device list matching the dynamic address (addr)

Parameters

- dev_list – Pointer to the device list struct.
- addr – Dynamic address to be matched.

Returns

Pointer to the I3C target device descriptor, or NULL if none is found.

struct *i3c_i2c_device_desc* *i3c_dev_list_i2c_addr_find(struct *i3c_dev_attached_list* *dev_list, uint16_t addr)

Find a I2C target device descriptor by address.

This finds the I2C target device descriptor in the attached device list matching the address (addr)

Parameters

- dev_list – Pointer to the device list struct.
- addr – Address to be matched.

Returns
Pointer to the I2C target device descriptor, or NULL if none is found.

int `i3c_determine_default_addr`(struct *i3c_device_desc* *target, uint8_t *addr)

Helper function to find the default address an i3c device is attached with.

This is a helper function to find the default address the device will be loaded with. This could be either it's static address, a requested dynamic address, or just a dynamic address that is available

Parameters

- `target` – **[in]** The pointer of the device descriptor
- `addr` – **[out]** Address to be assigned to target device.

Return values

- `0` – if successful.
- `-EINVAL` – if the expected default address is already in use

int `i3c_dev_list_daa_addr_helper`(struct *i3c_addr_slots* *addr_slots, const struct *i3c_dev_list* *dev_list, uint64_t pid, bool must_match, bool assigned_okay, struct *i3c_device_desc* **target, uint8_t *addr)

Helper function to find a usable address during ENTDAA.

This is a helper function to find a usable address during Dynamic Address Assignment. Given the PID (`pid`), it will search through the device list for the matching device descriptor. If the device descriptor indicates that there is a preferred address (i.e. assigned-address in device tree,

If `must_match` is true, the PID (`pid`) must match one of the device in the device list.

See also

i3c_device_desc::init_dynamic_addr), this preferred address will be returned if this address is still available. If it is not available, another free address will be returned.

If `must_match` is false, this will return an arbitrary address. This is useful when not all devices are described in device tree. Or else, the DAA process cannot proceed since there is no address to be assigned.

If `assigned_okay` is true, it will return the same address already assigned to the device (

If `assigned_okay` is false, the device cannot have an address assigned already (that

See also

i3c_device_desc::dynamic_addr). If no address has been assigned, it behaves as if `assigned_okay` is false. This is useful for assigning the same address to the same *device* (for example, hot-join after *device* coming back from suspend).

See also

i3c_device_desc::dynamic_addr is not zero). This is mainly used during the initial DAA.

Parameters

- **addr_slots** – **[in]** Pointer to address slots struct.
- **dev_list** – **[in]** Pointer to the device list struct.
- **pid** – **[in]** Provisioned ID of device to be assigned address.
- **must_match** – **[in]** True if PID must match devices in the device list. False otherwise.
- **assigned_okay** – **[in]** True if it is okay to return the address already assigned to the target matching the PID (pid).
- **target** – **[out]** Store the pointer of the device descriptor if it matches the incoming PID (pid).
- **addr** – **[out]** Address to be assigned to target device.

Return values

- **0** – if successful.
- **-ENODEV** – if no device matches the PID (pid) in the device list and must_match is true.
- **-EINVAL** – if the device matching PID (pid) already has an address assigned or invalid function arguments.

static inline int **i3c_configure**(const struct *device* *dev, enum *i3c_config_type* type, void *config)

Configure the I3C hardware.

Parameters

- **dev** – Pointer to controller device driver instance.
- **type** – Type of configuration parameters being passed in config.
- **config** – Pointer to the configuration parameters.

Return values

- **0** – If successful.
- **-EINVAL** – If invalid configure parameters.
- **-EIO** – General Input/Output errors.
- **-ENOSYS** – If not implemented.

static inline int **i3c_config_get**(const struct *device* *dev, enum *i3c_config_type* type, void *config)

Get configuration of the I3C hardware.

This provides a way to get the current configuration of the I3C hardware.

This can return cached config or probed hardware parameters, but it has to be up to date with current configuration.

Note that if type is I3C_CONFIG_CUSTOM, config must contain the ID of the parameter to be retrieved.

Parameters

- **dev** – **[in]** Pointer to controller device driver instance.
- **type** – **[in]** Type of configuration parameters being passed in config.
- **config** – **[inout]** Pointer to the configuration parameters.

Return values

- `0` – If successful.
- `-EIO` – General Input/Output errors.
- `-ENOSYS` – If not implemented.

static inline int `i3c_recover_bus`(const struct *device* *dev)

Attempt bus recovery on the I3C bus.

This routine asks the controller to attempt bus recovery.

Return values

- `0` – If successful.
- `-EBUSY` – If bus recovery fails.
- `-EIO` – General input / output error.
- `-ENOSYS` – Bus recovery is not supported by the controller driver.

int `i3c_attach_i3c_device`(struct *i3c_device_desc* *target)

Attach an I3C device.

Called to attach a I3C device to the addresses. This is typically called before a SETDASA or ENTDAA to reserve the addresses. This will also call the optional api to update any registers within the driver if implemented.

Warning

Use cases involving multiple writers to the i3c/i2c devices must prevent concurrent write operations, either by preventing all writers from being preempted or by using a mutex to govern writes to the i3c/i2c devices.

Parameters

- `target` – Pointer to the target device descriptor

Return values

- `0` – If successful.
- `-EINVAL` – If address is not available or if the device has already been attached before

int `i3c_reattach_i3c_device`(struct *i3c_device_desc* *target, uint8_t old_dyn_addr)

Reattach I3C device.

called after every time an I3C device has its address changed. It can be because the device has been powered down and has lost its address, or it can happen when a device had a static address and has been assigned a dynamic address with SETDASA or a dynamic address has been updated with SETNEWDA. This will also call the optional api to update any registers within the driver if implemented.

Warning

Use cases involving multiple writers to the i3c/i2c devices must prevent concurrent write operations, either by preventing all writers from being preempted or by using a mutex to govern writes to the i3c/i2c devices.

Parameters

- `target` – Pointer to the target device descriptor

- `old_dyn_addr` – The old dynamic address of target device, 0 if there was no old dynamic address

Return values

- `0` – If successful.
- `-EINVAL` – If address is not available

int `i3c_detach_i3c_device`(struct *i3c_device_desc* *target)

Detach I3C Device.

called to remove an I3C device and to free up the address that it used. If it's dynamic address was not set, then it assumed that SETDASA failed and will free it's static addr. This will also call the optional api to update any registers within the driver if implemented.

Warning

Use cases involving multiple writers to the i3c/i2c devices must prevent concurrent write operations, either by preventing all writers from being preempted or by using a mutex to govern writes to the i3c/i2c devices.

Parameters

- `target` – Pointer to the target device descriptor

Return values

- `0` – If successful.
- `-EINVAL` – If device is already detached

int `i3c_attach_i2c_device`(struct *i3c_i2c_device_desc* *target)

Attach an I2C device.

Called to attach a I2C device to the addresses. This will also call the optional api to update any registers within the driver if implemented.

Warning

Use cases involving multiple writers to the i3c/i2c devices must prevent concurrent write operations, either by preventing all writers from being preempted or by using a mutex to govern writes to the i3c/i2c devices.

Parameters

- `target` – Pointer to the target device descriptor

Return values

- `0` – If successful.
- `-EINVAL` – If address is not available or if the device has already been attached before

int `i3c_detach_i2c_device`(struct *i3c_i2c_device_desc* *target)

Detach I2C Device.

called to remove an I2C device and to free up the address that it used. This will also call the optional api to update any registers within the driver if implemented.

Warning

Use cases involving multiple writers to the i3c/i2c devices must prevent concurrent write operations, either by preventing all writers from being preempted or by using a mutex to govern writes to the i3c/i2c devices.

Parameters

- `target` – Pointer to the target device descriptor

Return values

- `0` – If successful.
- `-EINVAL` – If device is already detached

static inline int `i3c_do_daa`(const struct *device* *dev)

Perform Dynamic Address Assignment on the I3C bus.

This routine asks the controller to perform dynamic address assignment where the controller belongs. Only the active controller of the bus should do this.

Note

For controller driver implementation, the controller should perform SETDASA to allow static addresses to be the dynamic addresses before actually doing ENTDAA.

Parameters

- `dev` – Pointer to the device structure for the controller driver instance.

Return values

- `0` – If successful.
- `-EBUSY` – Bus is busy.
- `-EIO` – General input / output error.
- `-ENODEV` – If a provisioned ID does not match to any target devices in the registered device list.
- `-ENOSPC` – No more free addresses can be assigned to target.
- `-ENOSYS` – Dynamic address assignment is not supported by the controller driver.

int `i3c_do_ccc`(const struct *device* *dev, struct *i3c_ccc_payload* *payload)

Send CCC to the bus.

Parameters

- `dev` – Pointer to the device structure for the controller driver instance.
- `payload` – Pointer to the structure describing the CCC payload.

Return values

- `0` – If successful.
- `-EBUSY` – Bus is busy.
- `-EIO` – General Input / output error.
- `-EINVAL` – Invalid valid set in the payload structure.

- `-ENOSYS` – Not implemented.

static inline struct *i3c_device_desc* *`i3c_device_find`(const struct *device* *dev, const struct *i3c_device_id* *id)

Find a registered I3C target device.

Controller only API.

This returns the I3C device descriptor of the I3C device matching the incoming `id`.

Parameters

- `dev` – Pointer to controller device driver instance.
- `id` – Pointer to I3C device ID.

Returns

Pointer to I3C device descriptor, or NULL if no I3C device found matching incoming `id`.

int `i3c_bus_init`(const struct *device* *dev, const struct *i3c_dev_list* *i3c_dev_list)

Generic helper function to perform bus initialization.

Parameters

- `dev` – Pointer to controller device driver instance.
- `i3c_dev_list` – Pointer to I3C device list.

Return values

- `0` – If successful.
- `-EBUSY` – Bus is busy.
- `-EIO` – General input / output error.
- `-ENODEV` – If a provisioned ID does not match to any target devices in the registered device list.
- `-ENOSPC` – No more free addresses can be assigned to target.
- `-ENOSYS` – Dynamic address assignment is not supported by the controller driver.

int `i3c_device_basic_info_get`(struct *i3c_device_desc* *target)

Get basic information from device and update device descriptor.

This retrieves some basic information:

- Bus Characteristics Register (GETBCR)
- Device Characteristics Register (GETDCR)
- Max Read Length (GETMRL)
- Max Write Length (GETMWL) from the device and update the corresponding fields of the device descriptor.

This only updates the field(s) in device descriptor only if CCC operations succeed.

Parameters

- `target` – **[inout]** I3C target device descriptor.

Return values

- `0` – if successful.
- `-EIO` – General Input/Output error.

struct `i3c_config_controller`

#include <*i3c.h*> Configuration parameters for I3C hardware to act as controller.

Public Members

bool `is_secondary`

True if the controller is to be the secondary controller of the bus.

False to be the primary controller.

uint32_t `i3c`

SCL frequency (in Hz) for I3C transfers.

uint32_t `i2c`

SCL frequency (in Hz) for I2C transfers.

uint8_t `supported_hdr`

Bit mask of supported HDR modes (0 - 7).

This can be used to enable or disable HDR mode supported by the hardware at runtime.

struct `i3c_config_custom`

#include <*i3c.h*> Custom I3C configuration parameters.

This can be used to configure the I3C hardware on parameters not covered by

See also

i3c_config_controller or

See also

i3c_config_target. Mostly used to configure vendor specific parameters of the I3C hardware.

Public Members

uint32_t `id`

ID of the configuration parameter.

uintptr_t `val`

Value of configuration parameter.

void *`ptr`

Pointer to configuration parameter.

Mainly used to pointer to a struct that the device driver understands.

struct **i3c_device_id**

#include <*i3c.h*> Structure used for matching I3C devices.

Public Members

const uint64_t **pid**

Device Provisioned ID.

struct **i3c_device_desc**

#include <*i3c.h*> Structure describing a I3C target device.

Instances of this are passed to the I3C controller device APIs, for example:

- i3c_device_register() to tell the controller of a target device.
- i3c_transfers() to initiate data transfers between controller and target device.

Fields `bus`, `pid` and `static_addr` must be initialized by the module that implements the target device behavior prior to passing the object reference to I3C controller device APIs. `static_addr` can be zero if target device does not have static address.

Field `node` should not be initialized or modified manually.

Public Members

const struct *device* *const **bus**

Used to attach this node onto a linked list.

I3C bus to which this target device is attached

const struct *device* *const **dev**

Device driver instance of the I3C device.

const uint64_t **pid**

Device Provisioned ID.

const uint8_t **static_addr**

Static address for this target device.

0 if static address is not being used, and only dynamic address is used. This means that the target device must go through ENTDAA (Dynamic Address Assignment) to get a dynamic address before it can communicate with the controller. This means SETAASA and SETDASA CCC cannot be used to set dynamic address on the target device (as both are to tell target device to use static address as dynamic address).

const uint8_t **init_dynamic_addr**

Initial dynamic address.

This is specified in the device tree property “assigned-address” to indicate the desired dynamic address during address assignment (SETDASA and ENTDAA).

0 if there is no preference.

uint8_t `dynamic_addr`

Dynamic Address for this target device used for communication.

This is to be set by the controller driver in one of the following situations:
- During Dynamic Address Assignment (during ENTDAA)
- Reset Dynamic Address Assignment (RSTDAA)
- Set All Addresses to Static Addresses (SETAASA)
- Set New Dynamic Address (SETNEWDA)
- Set Dynamic Address from Static Address (SETDASA)

0 if address has not been assigned.

uint8_t `group_addr`

Group address for this target device.

Set during:
- Reset Group Address(es) (RSTGRPA)
- Set Group Address (SETGRPA)

0 if group address has not been assigned.

uint8_t `bcr`

Bus Characteristic Register (BCR)

- BCR[7:6]: Device Role
  - 0: I3C Target
  - 1: I3C Controller capable
  - 2: Reserved
  - 3: Reserved
- BCR[5]: Advanced Capabilities
  - 0: Does not support optional advanced capabilities.
  - 1: Supports optional advanced capabilities which can be viewed via GETCAPS CCC.
- BCR[4}: Virtual Target Support
  - 0: Is not a virtual target.
  - 1: Is a virtual target.
- BCR[3]: Offline Capable
  - 0: Will always response to I3C commands.
  - 1: Will not always response to I3C commands.
- BCR[2]: IBI Payload
  - 0: No data bytes following the accepted IBI.
  - 1: One data byte (MDB, Mandatory Data Byte) follows the accepted IBI. Additional data bytes may also follows.
- BCR[1]: IBI Request Capable
  - 0: Not capable
  - 1: Capable
- BCR[0]: Max Data Speed Limitation
  - 0: No Limitation
  - 1: Limitation obtained via GETMXDS CCC.

uint8_t `dcr`

Device Characteristic Register (DCR)

Describes the type of device. Refer to official documentation on what this number means.

uint8_t `maxrd`

Maximum Read Speed.

uint8_t **maxwr**

Maximum Write Speed.

uint32_t **max_read_turnaround**

Maximum Read turnaround time in microseconds.

uint16_t **mrl**

Maximum Read Length.

uint16_t **mwl**

Maximum Write Length.

uint8_t **max_ibi**

Maximum IBI Payload Size.

Valid only if BCR[2] is 1.

uint8_t **gethdrcap**

I3C v1.0 HDR Capabilities (`I3C_CCC_GETCAPS1_*`)

- Bit[0]: HDR-DDR
- Bit[1]: HDR-TSP
- Bit[2]: HDR-TSL
- Bit[7:3]: Reserved

uint8_t **getcap1**

I3C v1.1+ GETCAPS1 (`I3C_CCC_GETCAPS1_*`)

- Bit[0]: HDR-DDR
- Bit[1]: HDR-TSP
- Bit[2]: HDR-TSL
- Bit[3]: HDR-BT
- Bit[7:4]: Reserved

uint8_t **getcap2**

GETCAPS2 (`I3C_CCC_GETCAPS2_*`)

- Bit[3:0]: I3C 1.x Specification Version
- Bit[5:4]: Group Address Capabilities
- Bit[6]: HDR-DDR Write Abort
- Bit[7]: HDR-DDR Abort CRC

uint8_t **getcap3**

GETCAPS3 (`I3C_CCC_GETCAPS3_*`)

- Bit[0]: Multi-Lane (ML) Data Transfer Support
- Bit[1]: Device to Device Transfer (D2DXFER) Support
- Bit[2]: Device to Device Transfer (D2DXFER) IBI Capable
- Bit[3]: Defining Byte Support in GETCAPS
- Bit[4]: Defining Byte Support in GETSTATUS
- Bit[5]: HDR-BT CRC-32 Support

- Bit[6]: IBI MDB Support for Pending Read Notification
- Bit[7]: Reserved

uint8_t **getcap4**

GETCAPS4.

- Bit[7:0]: Reserved

struct *i3c_device_desc* **getcaps**

Describes advanced (Target) capabilities and features.

i3c_target_ibi_cb_t **ibi_cb**

Private data by the controller to aid in transactions.

Do not modify. In-Band Interrupt (IBI) callback.

struct **i3c_i2c_device_desc**

#include <*i3c.h*> Structure describing a I2C device on I3C bus.

Instances of this are passed to the I3C controller device APIs, for example: () i3c_i2c_device_register() to tell the controller of an I2C device. () i3c_i2c_transfers() to initiate data transfers between controller and I2C device.

Fields other than node must be initialized by the module that implements the device behavior prior to passing the object reference to I3C controller device APIs.

Public Members

const struct *device* ***bus**

Used to attach this node onto a linked list.

I3C bus to which this I2C device is attached

const uint16_t **addr**

Static address for this I2C device.

const uint8_t **lvr**

Legacy Virtual Register (LVR)

- LVR[7:5]: I2C device index:
  - 0: I2C device has a 50 ns spike filter where it is not affected by high frequency on SCL.
  - 1: I2C device does not have a 50 ns spike filter but can work with high frequency on SCL.
  - 2: I2C device does not have a 50 ns spike filter and cannot work with high frequency on SCL.
- LVR[4]: I2C mode indicator:
  - 0: FM+ mode
  - 1: FM mode
- LVR[3:0]: Reserved.

struct **i3c_dev_attached_list**

#include <*i3c.h*> Structure for describing attached devices for a controller.

This contains slists of attached I3C and I2C devices.

This is a helper struct that can be used by controller device driver to aid in device management.

Public Members

struct *i3c_addr_slots* **addr_slots**

Address slots:

- Aid in dynamic address assignment.
- Quick way to find out if a target address is a I3C or I2C device.

sys_slist_t **i3c**

Linked list of attached I3C devices.

sys_slist_t **i2c**

Linked list of attached I2C devices.

struct **i3c_dev_list**

#include <*i3c.h*> Structure for describing known devices for a controller.

This contains arrays of known I3C and I2C devices.

This is a helper struct that can be used by controller device driver to aid in device management.

Public Members

struct *i3c_device_desc* *const **i3c**

Pointer to array of known I3C devices.

struct *i3c_i2c_device_desc* *const **i2c**

Pointer to array of known I2C devices.

const uint8_t **num_i3c**

Number of I3C devices in array.

const uint8_t **num_i2c**

Number of I2C devices in array.

struct **i3c_driver_config**

#include <*i3c.h*> This structure is common to all I3C drivers and is expected to be the first element in the object pointed to by the config field in the device structure.

Public Members

struct *i3c_dev_list* `dev_list`
I3C/I2C device list struct.

struct `i3c_driver_data`
#include <*i3c.h*> This structure is common to all I3C drivers and is expected to be the first element in the driver's struct driver_data declaration.

Public Members

struct *i3c_config_controller* `ctrl_config`
Controller Configuration.

struct *i3c_dev_attached_list* `attached_dev`
Attached I3C/I2C devices and addresses.

group `i3c_ccc`
I3C Common Command Codes.

Defines

`I3C_CCC_BROADCAST_MAX_ID`
Maximum CCC ID for broadcast.

`I3C_CCC_ENEC`(broadcast)
Enable Events Command.

Parameters

- `broadcast` – True if broadcast, false if direct.

`I3C_CCC_DISEC`(broadcast)
Disable Events Command.

Parameters

- `broadcast` – True if broadcast, false if direct.

`I3C_CCC_ENTAS`(as, broadcast)
Enter Activity State.

Parameters

- `as` – Desired activity state
- `broadcast` – True if broadcast, false if direct.

`I3C_CCC_ENTAS0`(broadcast)
Enter Activity State 0.

Parameters

- `broadcast` – True if broadcast, false if direct.

I3C_CCC_ENTAS1(broadcast)
Enter Activity State 1.

Parameters

- broadcast – True if broadcast, false if direct.

I3C_CCC_ENTAS2(broadcast)
Enter Activity State 2.

Parameters

- broadcast – True if broadcast, false if direct.

I3C_CCC_ENTAS3(broadcast)
Enter Activity State 3.

Parameters

- broadcast – True if broadcast, false if direct.

I3C_CCC_RSTDAA
Reset Dynamic Address Assignment (Broadcast)

I3C_CCC_ENTDAA
Enter Dynamic Address Assignment (Broadcast)

I3C_CCC_DEFTGTS
Define List of Targets (Broadcast)

I3C_CCC_SETMWL(broadcast)
Set Max Write Length (Broadcast or Direct)

Parameters

- broadcast – True if broadcast, false if direct.

I3C_CCC_SETMRL(broadcast)
Set Max Read Length (Broadcast or Direct)

Parameters

- broadcast – True if broadcast, false if direct.

I3C_CCC_ENTTM
Enter Test Mode (Broadcast)

I3C_CCC_SETBUSCON
Set Bus Context (Broadcast)

I3C_CCC_ENDXFER(broadcast)
Data Transfer Ending Procedure Control.

Parameters

- broadcast – True if broadcast, false if direct.

I3C_CCC_ENTHDR(x)
Enter HDR Mode (HDR-DDR) (Broadcast)

I3C_CCC_ENTHDR0
Enter HDR Mode 0 (HDR-DDR) (Broadcast)

I3C_CCC_ENTHDR1
: Enter HDR Mode 1 (HDR-TSP) (Broadcast)

I3C_CCC_ENTHDR2
: Enter HDR Mode 2 (HDR-TSL) (Broadcast)

I3C_CCC_ENTHDR3
: Enter HDR Mode 3 (HDR-BT) (Broadcast)

I3C_CCC_ENTHDR4
: Enter HDR Mode 4 (Broadcast)

I3C_CCC_ENTHDR5
: Enter HDR Mode 5 (Broadcast)

I3C_CCC_ENTHDR6
: Enter HDR Mode 6 (Broadcast)

I3C_CCC_ENTHDR7
: Enter HDR Mode 7 (Broadcast)

I3C_CCC_SETXTIME(broadcast)
: Exchange Timing Information (Broadcast or Direct)

  Parameters

  - broadcast – True if broadcast, false if direct.

I3C_CCC_SETAASA
: Set All Addresses to Static Addresses (Broadcast)

I3C_CCC_RSTACT(broadcast)
: Target Reset Action.

  Parameters

  - broadcast – True if broadcast, false if direct.

I3C_CCC_DEFGRPA
: Define List of Group Address (Broadcast)

I3C_CCC_RSTGRPA(broadcast)
: Reset Group Address.

  Parameters

  - broadcast – True if broadcast, false if direct.

I3C_CCC_MLANE(broadcast)
: Multi-Lane Data Transfer Control (Broadcast)

I3C_CCC_VENDOR(broadcast, id)
: Vendor/Standard Extension.

  Parameters

  - broadcast – True if broadcast, false if direct.
  - id – Extension ID.

`I3C_CCC_SETDASA`
: Set Dynamic Address from Static Address (Direct)

`I3C_CCC_SETNEWDA`
: Set New Dynamic Address (Direct)

`I3C_CCC_GETMWL`
: Get Max Write Length (Direct)

`I3C_CCC_GETMRL`
: Get Max Read Length (Direct)

`I3C_CCC_GETPID`
: Get Provisioned ID (Direct)

`I3C_CCC_GETBCR`
: Get Bus Characteristics Register (Direct)

`I3C_CCC_GETDCR`
: Get Device Characteristics Register (Direct)

`I3C_CCC_GETSTATUS`
: Get Device Status (Direct)

`I3C_CCC_GETACCCR`
: Get Accept Controller Role (Direct)

`I3C_CCC_SETBRGTGT`
: Set Bridge Targets (Direct)

`I3C_CCC_GETMXDS`
: Get Max Data Speed (Direct)

`I3C_CCC_GETCAPS`
: Get Optional Feature Capabilities (Direct)

`I3C_CCC_SETROUTE`
: Set Route (Direct)

`I3C_CCC_D2DXFER`
: Device to Device(s) Tunneling Control (Direct)

`I3C_CCC_GETXTIME`
: Get Exchange Timing Information (Direct)

`I3C_CCC_SETGRPA`
: Set Group Address (Direct)

I3C_CCC_ENEC_EVT_ENINTR
: Enable Events (ENEC) - Target Interrupt Requests.

I3C_CCC_ENEC_EVT_ENCR
: Enable Events (ENEC) - Controller Role Requests.

I3C_CCC_ENEC_EVT_ENHJ
: Enable Events (ENEC) - Hot-Join Event.

I3C_CCC_ENEC_EVT_ALL

I3C_CCC_DISEC_EVT_DISINTR
: Disable Events (DISEC) - Target Interrupt Requests.

I3C_CCC_DISEC_EVT_DISCR
: Disable Events (DISEC) - Controller Role Requests.

I3C_CCC_DISEC_EVT_DISHJ
: Disable Events (DISEC) - Hot-Join Event.

I3C_CCC_DISEC_EVT_ALL

I3C_CCC_EVT_INTR
: Events - Target Interrupt Requests.

I3C_CCC_EVT_CR
: Events - Controller Role Requests.

I3C_CCC_EVT_HJ
: Events - Hot-Join Event.

I3C_CCC_EVT_ALL
: Bitmask for all events.

I3C_CCC_GETSTATUS_PROTOCOL_ERR
: GETSTATUS Format 1 - Protocol Error bit.

I3C_CCC_GETSTATUS_ACTIVITY_MODE_SHIFT
: GETSTATUS Format 1 - Activity Mode bit shift value.

I3C_CCC_GETSTATUS_ACTIVITY_MODE_MASK
: GETSTATUS Format 1 - Activity Mode bitmask.

I3C_CCC_GETSTATUS_ACTIVITY_MODE(status)
: GETSTATUS Format 1 - Activity Mode.

  Obtain Activity Mode from GETSTATUS Format 1 value obtained via GETSTATUS.

  Parameters

  - **status** – GETSTATUS Format 1 value

I3C_CCC_GETSTATUS_NUM_INT_SHIFT

GETSTATUS Format 1 - Number of Pending Interrupts bit shift value.

I3C_CCC_GETSTATUS_NUM_INT_MASK

GETSTATUS Format 1 - Number of Pending Interrupts bitmask.

I3C_CCC_GETSTATUS_NUM_INT(status)

GETSTATUS Format 1 - Number of Pending Interrupts.

Obtain Number of Pending Interrupts from GETSTATUS Format 1 value obtained via GETSTATUS.

Parameters

- **status** – GETSTATUS Format 1 value

I3C_CCC_GETSTATUS_PRECR_DEEP_SLEEP_DETECTED

GETSTATUS Format 2 - PERCR - Deep Sleep Detected bit.

I3C_CCC_GETSTATUS_PRECR_HANDOFF_DELAY_NACK

GETSTATUS Format 2 - PERCR - Handoff Delay NACK.

I3C_CCC_GETMXDS_MAX_SDR_FSCL_MAX

Get Max Data Speed (GETMXDS) - Default Max Sustained Data Rate.

I3C_CCC_GETMXDS_MAX_SDR_FSCL_8MHZ

Get Max Data Speed (GETMXDS) - 8MHz Max Sustained Data Rate.

I3C_CCC_GETMXDS_MAX_SDR_FSCL_6MHZ

Get Max Data Speed (GETMXDS) - 6MHz Max Sustained Data Rate.

I3C_CCC_GETMXDS_MAX_SDR_FSCL_4MHZ

Get Max Data Speed (GETMXDS) - 4MHz Max Sustained Data Rate.

I3C_CCC_GETMXDS_MAX_SDR_FSCL_2MHZ

Get Max Data Speed (GETMXDS) - 2MHz Max Sustained Data Rate.

I3C_CCC_GETMXDS_TSCO_8NS

Get Max Data Speed (GETMXDS) - Clock to Data Turnaround <= 8ns.

I3C_CCC_GETMXDS_TSCO_9NS

Get Max Data Speed (GETMXDS) - Clock to Data Turnaround <= 9ns.

I3C_CCC_GETMXDS_TSCO_10NS

Get Max Data Speed (GETMXDS) - Clock to Data Turnaround <= 10ns.

I3C_CCC_GETMXDS_TSCO_11NS

Get Max Data Speed (GETMXDS) - Clock to Data Turnaround <= 11ns.

I3C_CCC_GETMXDS_TSCO_12NS

Get Max Data Speed (GETMXDS) - Clock to Data Turnaround <= 12ns.

I3C_CCC_GETMXDS_TSCO_GT_12NS

Get Max Data Speed (GETMXDS) - Clock to Data Turnaround > 12ns.

I3C_CCC_GETMXDS_MAXWR_DEFINING_BYTE_SUPPORT

Get Max Data Speed (GETMXDS) - maxWr - Optional Defining Byte Support.

I3C_CCC_GETMXDS_MAXWR_MAX_SDR_FSCL_SHIFT

Get Max Data Speed (GETMXDS) - Max Sustained Data Rate bit shift value.

I3C_CCC_GETMXDS_MAXWR_MAX_SDR_FSCL_MASK

Get Max Data Speed (GETMXDS) - Max Sustained Data Rate bitmask.

I3C_CCC_GETMXDS_MAXWR_MAX_SDR_FSCL(maxwr)

Get Max Data Speed (GETMXDS) - maxWr - Max Sustained Data Rate.

Obtain Max Sustained Data Rate value from GETMXDS maxWr value obtained via GETMXDS.

Parameters

- `maxwr` – GETMXDS maxWr value.

I3C_CCC_GETMXDS_MAXRD_W2R_PERMITS_STOP_BETWEEN

Get Max Data Speed (GETMXDS) - maxRd - Write-to-Read Permits Stop Between.

I3C_CCC_GETMXDS_MAXRD_TSCO_SHIFT

Get Max Data Speed (GETMXDS) - maxRd - Clock to Data Turnaround bit shift value.

I3C_CCC_GETMXDS_MAXRD_TSCO_MASK

Get Max Data Speed (GETMXDS) - maxRd - Clock to Data Turnaround bitmask.

I3C_CCC_GETMXDS_MAXRD_TSCO(maxrd)

Get Max Data Speed (GETMXDS) - maxRd - Clock to Data Turnaround.

Obtain Clock to Data Turnaround value from GETMXDS maxRd value obtained via GETMXDS.

Parameters

- `maxrd` – GETMXDS maxRd value.

I3C_CCC_GETMXDS_MAXRD_MAX_SDR_FSCL_SHIFT

Get Max Data Speed (GETMXDS) - maxRd - Max Sustained Data Rate bit shift value.

I3C_CCC_GETMXDS_MAXRD_MAX_SDR_FSCL_MASK

Get Max Data Speed (GETMXDS) - maxRd - Max Sustained Data Rate bitmask.

I3C_CCC_GETMXDS_MAXRD_MAX_SDR_FSCL(maxrd)

Get Max Data Speed (GETMXDS) - maxRd - Max Sustained Data Rate.

Obtain Max Sustained Data Rate value from GETMXDS maxRd value obtained via GETMXDS.

Parameters

- `maxrd` – GETMXDS maxRd value.

I3C_CCC_GETMXDS_CRDHLY1_SET_BUS_ACT_STATE

Get Max Data Speed (GETMXDS) - CRDHLY1 - Set Bus Activity State bit shift value.

I3C_CCC_GETMXDS_CRDHLY1_CTRL_HANDOFF_ACT_STATE_SHIFT

Get Max Data Speed (GETMXDS) - CRDHLY1 - Controller Handoff Activity State bit shift value.

I3C_CCC_GETMXDS_CRDHLY1_CTRL_HANDOFF_ACT_STATE_MASK

Get Max Data Speed (GETMXDS) - CRDHLY1 - Controller Handoff Activity State bitmask.

I3C_CCC_GETMXDS_CRDHLY1_CTRL_HANDOFF_ACT_STATE(crhdly1)

Get Max Data Speed (GETMXDS) - CRDHLY1 - Controller Handoff Activity State.

Obtain Controller Handoff Activity State value from GETMXDS value obtained via GETMXDS.

Parameters

- crhdly1 – GETMXDS value.

I3C_CCC_GETCAPS1_HDR_DDR

Get Optional Feature Capabilities Byte 1 (GETCAPS) Format 1 - HDR-DDR mode bit.

I3C_CCC_GETCAPS1_HDR_TSP

Get Optional Feature Capabilities Byte 1 (GETCAPS) Format 1 - HDR-TSP mode bit.

I3C_CCC_GETCAPS1_HDR_TSL

Get Optional Feature Capabilities Byte 1 (GETCAPS) Format 1 - HDR-TSL mode bit.

I3C_CCC_GETCAPS1_HDR_BT

Get Optional Feature Capabilities Byte 1 (GETCAPS) Format 1 - HDR-BT mode bit.

I3C_CCC_GETCAPS1_HDR_MODE(x)

Get Optional Feature Capabilities Byte 1 (GETCAPS) - HDR Mode.

Get the bit corresponding to HDR mode.

Parameters

- x – HDR mode

I3C_CCC_GETCAPS1_HDR_MODE0

Get Optional Feature Capabilities Byte 1 (GETCAPS) Format 1 - HDR Mode 0.

I3C_CCC_GETCAPS1_HDR_MODE1

Get Optional Feature Capabilities Byte 1 (GETCAPS) Format 1 - HDR Mode 1.

I3C_CCC_GETCAPS1_HDR_MODE2

Get Optional Feature Capabilities Byte 1 (GETCAPS) Format 1 - HDR Mode 2.

I3C_CCC_GETCAPS1_HDR_MODE3

Get Optional Feature Capabilities Byte 1 (GETCAPS) Format 1 - HDR Mode 3.

I3C_CCC_GETCAPS1_HDR_MODE4

Get Optional Feature Capabilities Byte 1 (GETCAPS) Format 1 - HDR Mode 4.

I3C_CCC_GETCAPS1_HDR_MODE5

Get Optional Feature Capabilities Byte 1 (GETCAPS) Format 1 - HDR Mode 5.

I3C_CCC_GETCAPS1_HDR_MODE6

Get Optional Feature Capabilities Byte 1 (GETCAPS) Format 1 - HDR Mode 6.

I3C_CCC_GETCAPS1_HDR_MODE7

Get Optional Feature Capabilities Byte 1 (GETCAPS) Format 1 - HDR Mode 7.

I3C_CCC_GETCAPS2_HDRDDR_WRITE_ABORT

Get Optional Feature Capabilities Byte 2 (GETCAPS) Format 1 - HDR-DDR Write Abort bit.

I3C_CCC_GETCAPS2_HDRDDR_ABORT_CRC

Get Optional Feature Capabilities Byte 2 (GETCAPS) Format 1 - HDR-DDR Abort CRC bit.

I3C_CCC_GETCAPS2_GRPADDR_CAP_SHIFT

Get Optional Feature Capabilities Byte 2 (GETCAPS) Format 1 - Group Address Capabilities bit shift value.

I3C_CCC_GETCAPS2_GRPADDR_CAP_MASK

Get Optional Feature Capabilities Byte 2 (GETCAPS) Format 1 - Group Address Capabilities bitmask.

I3C_CCC_GETCAPS2_GRPADDR_CAP(getcaps2)

Get Optional Feature Capabilities Byte 2 (GETCAPS) Format 1 - Group Address Capabilities.

Obtain Group Address Capabilities value from GETCAPS Format 1 value obtained via GETCAPS.

Parameters

- `getcaps2` – GETCAPS2 value.

I3C_CCC_GETCAPS2_SPEC_VER_SHIFT

Get Optional Feature Capabilities Byte 2 (GETCAPS) Format 1 - I3C 1.x Specification Version bit shift value.

I3C_CCC_GETCAPS2_SPEC_VER_MASK

Get Optional Feature Capabilities Byte 2 (GETCAPS) Format 1 - I3C 1.x Specification Version bitmask.

I3C_CCC_GETCAPS2_SPEC_VER(getcaps2)

Get Optional Feature Capabilities Byte 2 (GETCAPS) Format 1 - I3C 1.x Specification Version.

Obtain I3C 1.x Specification Version value from GETCAPS Format 1 value obtained via GETCAPS.

Parameters

- `getcaps2` – GETCAPS2 value.

I3C_CCC_GETCAPS3_MLANE_SUPPORT

Get Optional Feature Capabilities Byte 3 (GETCAPS) Format 1 - Multi-Lane Data Transfer Support bit.

I3C_CCC_GETCAPS3_D2DXFER_SUPPORT

Get Optional Feature Capabilities Byte 3 (GETCAPS) Format 1 - Device to Device Transfer (D2DXFER) Support bit.

I3C_CCC_GETCAPS3_D2DXFER_IBI_CAPABLE

Get Optional Feature Capabilities Byte 3 (GETCAPS) Format 1 - Device to Device Transfer (D2DXFER) IBI Capable bit.

I3C_CCC_GETCAPS3_GETCAPS_DEFINING_BYTE_SUPPORT

Get Optional Feature Capabilities Byte 3 (GETCAPS) Format 1 - Defining Byte Support in GETCAPS bit.

I3C_CCC_GETCAPS3_GETSTATUS_DEFINING_BYTE_SUPPORT

Get Optional Feature Capabilities Byte 3 (GETCAPS) Format 1 - Defining Byte Support in GETSTATUS bit.

I3C_CCC_GETCAPS3_HDRBT_CRC32_SUPPORT

Get Optional Feature Capabilities Byte 3 (GETCAPS) Format 1 - HDR-BT CRC-32 Support bit.

I3C_CCC_GETCAPS3_IBI_MDR_PENDING_READ_NOTIFICATION

Get Optional Feature Capabilities Byte 3 (GETCAPS) Format 1 - IBI MDB Support for Pending Read Notification bit.

I3C_CCC_GETCAPS_TESTPAT1

Get Fixed Test Pattern (GETCAPS) Format 2 - Fixed Test Pattern Byte 1.

I3C_CCC_GETCAPS_TESTPAT2

Get Fixed Test Pattern (GETCAPS) Format 2 - Fixed Test Pattern Byte 2.

I3C_CCC_GETCAPS_TESTPAT3

Get Fixed Test Pattern (GETCAPS) Format 2 - Fixed Test Pattern Byte 3.

I3C_CCC_GETCAPS_TESTPAT4

Get Fixed Test Pattern (GETCAPS) Format 2 - Fixed Test Pattern Byte 4.

I3C_CCC_GETCAPS_TESTPAT

Get Fixed Test Pattern (GETCAPS) Format 2 - Fixed Test Pattern Word in Big Endian.

I3C_CCC_GETCAPS_CRCAPS1_HJ_SUPPORT

Get Controller Handoff Capabilities Byte 1 (GETCAPS) Format 2 - Hot-Join Support.

I3C_CCC_GETCAPS_CRCAPS1_GRP_MANAGEMENT_SUPPORT

Get Controller Handoff Capabilities Byte 1 (GETCAPS) Format 2 - Group Management Support.

I3C_CCC_GETCAPS_CRCAPS1_ML_SUPPORT

Get Controller Handoff Capabilities Byte 1 (GETCAPS) Format 2 - Multi-Lane Support.

I3C_CCC_GETCAPS_CRCAPS2_IBI_TIR_SUPPORT

Get Controller Handoff Capabilities Byte 2 (GETCAPS) Format 2 - In-Band Interrupt Support.

I3C_CCC_GETCAPS_CRCAPS2_CONTROLLER_PASSBACK

Get Controller Handoff Capabilities Byte 2 (GETCAPS) Format 2 - Controller Pass-Back.

I3C_CCC_GETCAPS_CRCAPS2_DEEP_SLEEP_CAPABLE

Get Controller Handoff Capabilities Byte 2 (GETCAPS) Format 2 - Deep Sleep Capable.

I3C_CCC_GETCAPS_CRCAPS2_DELAYED_CONTROLLER_HANDOFF

Get Controller Handoff Capabilities Byte 2 (GETCAPS) Format 2 - Deep Sleep Capable.

I3C_CCC_GETCAPS_VTCAP1_VITRUAL_TARGET_TYPE_SHIFT

Get Capabilities (GETCAPS) - VTCAP1 - Virtual Target Type bit shift value.

I3C_CCC_GETCAPS_VTCAP1_VITRUAL_TARGET_TYPE_MASK

Get Capabilities (GETCAPS) - VTCAP1 - Virtual Target Type bitmask.

I3C_CCC_GETCAPS_VTCAP1_VITRUAL_TARGET_TYPE(vtcap1)

Get Capabilities (GETCAPS) - VTCAP1 - Virtual Target Type.

Obtain Virtual Target Type value from VTCAP1 value obtained via GETCAPS format 2 VTCAP def byte.

Parameters

- vtcap1 – VTCAP1 value.

I3C_CCC_GETCAPS_VTCAP1_SIDE_EFFECTS

Get Virtual Target Capabilities Byte 1 (GETCAPS) Format 2 - Side Effects.

I3C_CCC_GETCAPS_VTCAP1_SHARED_PERIPH_DETECT

Get Virtual Target Capabilities Byte 1 (GETCAPS) Format 2 - Shared Peripheral Detect.

I3C_CCC_GETCAPS_VTCAP2_INTERRUPT_REQUESTS_SHIFT

Get Capabilities (GETCAPS) - VTCAP2 - Interrupt Requests bit shift value.

I3C_CCC_GETCAPS_VTCAP2_INTERRUPT_REQUESTS_MASK

Get Capabilities (GETCAPS) - VTCAP2 - Interrupt Requests bitmask.

I3C_CCC_GETCAPS_VTCAP2_INTERRUPT_REQUESTS(vtcap2)

Get Capabilities (GETCAPS) - VTCAP2 - Interrupt Requests.

Obtain Interrupt Requests value from VTCAP2 value obtained via GETCAPS format 2 VTCAP def byte.

Parameters

- vtcap2 – VTCAP2 value.

I3C_CCC_GETCAPS_VTCAP2_ADDRESS_REMAPPING
: Get Virtual Target Capabilities Byte 2 (GETCAPS) Format 2 - Address Remapping.

I3C_CCC_GETCAPS_VTCAP2_BUS_CONTEXT_AND_COND_SHIFT
: Get Capabilities (GETCAPS) - VTCAP2 - Bus Context and Condition bit shift value.

I3C_CCC_GETCAPS_VTCAP2_BUS_CONTEXT_AND_COND_MASK
: Get Capabilities (GETCAPS) - VTCAP2 - Bus Context and Condition bitmask.

I3C_CCC_GETCAPS_VTCAP2_BUS_CONTEXT_AND_COND(vtcap2)
: Get Capabilities (GETCAPS) - VTCAP2 - Bus Context and Condition.

  Obtain Bus Context and Condition value from VTCAP2 value obtained via GETCAPS format 2 VTCAP def byte.

  Parameters

  - vtcap2 – VTCAP2 value.

Enums

enum i3c_ccc_getstatus_fmt
: Indicate which format of GETSTATUS to use.

  Values:

  enumerator GETSTATUS_FORMAT_1
  : GETSTATUS Format 1.

  enumerator GETSTATUS_FORMAT_2
  : GETSTATUS Format 2.

enum i3c_ccc_getstatus_defbyte
: Defining byte values for GETSTATUS Format 2.

  Values:

  enumerator GETSTATUS_FORMAT_2_TGTSTAT = 0x00U
  : Target status.

  enumerator GETSTATUS_FORMAT_2_PRECR = 0x91U
  : PRECR - Alternate status format describing Controller-capable device.

  enumerator GETSTATUS_FORMAT_2_INVALID = 0x100U
  : Invalid defining byte.

enum i3c_ccc_getcaps_fmt
: Indicate which format of GETCAPS to use.

  Values:

  enumerator GETCAPS_FORMAT_1
  : GETCAPS Format 1.

enumerator GETCAPS_FORMAT_2
GETCAPS Format 2.

enum i3c_ccc_getcaps_defbyte
Enum for I3C Get Capabilities (GETCAPS) Format 2 Defining Byte Values.
Values:

enumerator GETCAPS_FORMAT_2_TGTCAPS = 0x00U
Standard Target capabilities and features.

enumerator GETCAPS_FORMAT_2_TESTPAT = 0x5AU
Fixed 32b test pattern.

enumerator GETCAPS_FORMAT_2_CRCAPS = 0x91U
Controller handoff capabilities and features.

enumerator GETCAPS_FORMAT_2_VTCAPS = 0x93U
Virtual Target capabilities and features.

enumerator GETCAPS_FORMAT_2_DBGCAPS = 0xD7U
Debug-capable Device capabilities and features.

enumerator GETCAPS_FORMAT_2_INVALID = 0x100
Invalid defining byte.

enum i3c_ccc_rstact_defining_byte
Enum for I3C Reset Action (RSTACT) Defining Byte Values.
Values:

enumerator I3C_CCC_RSTACT_NO_RESET = 0x00U
No Reset on Target Reset Pattern.

enumerator I3C_CCC_RSTACT_PERIPHERAL_ONLY = 0x01U
Reset the I3C Peripheral Only.

enumerator I3C_CCC_RSTACT_RESET_WHOLE_TARGET = 0x02U
Reset the Whole Target.

enumerator I3C_CCC_RSTACT_DEBUG_NETWORK_ADAPTER = 0x03U
Debug Network Adapter Reset.

enumerator I3C_CCC_RSTACT_VIRTUAL_TARGET_DETECT = 0x04U
Virtual Target Detect.

Functions

static inline bool **i3c_ccc_is_payload_broadcast**(const struct *i3c_ccc_payload* *payload)

Test if I3C CCC payload is for broadcast.

This tests if the CCC payload is for broadcast.

Parameters

- **payload** – **[in]** Pointer to the CCC payload.

Return values

- **true** – if payload target is broadcast
- **false** – if payload target is direct

int **i3c_ccc_do_getbcr**(struct *i3c_device_desc* *target, struct *i3c_ccc_getbcr* *bcr)

Get BCR from a target.

Helper function to get BCR (Bus Characteristic Register) from target device.

See also

i3c_do_ccc

Parameters

- **target** – **[in]** Pointer to the target device descriptor.
- **bcr** – **[out]** Pointer to the BCR payload structure.

Returns

int **i3c_ccc_do_getdcr**(struct *i3c_device_desc* *target, struct *i3c_ccc_getdcr* *dcr)

Get DCR from a target.

Helper function to get DCR (Device Characteristic Register) from target device.

See also

i3c_do_ccc

Parameters

- **target** – **[in]** Pointer to the target device descriptor.
- **dcr** – **[out]** Pointer to the DCR payload structure.

Returns

int **i3c_ccc_do_getpid**(struct *i3c_device_desc* *target, struct *i3c_ccc_getpid* *pid)

Get PID from a target.

Helper function to get PID (Provisioned ID) from target device.

See also

i3c_do_ccc

Parameters

- `target` – **[in]** Pointer to the target device descriptor.
- `pid` – **[out]** Pointer to the PID payload structure.

Returns

int `i3c_ccc_do_rstact_all`(const struct *device* *controller, enum *i3c_ccc_rstact_defining_byte* action)

Broadcast RSTACT to reset I3C Peripheral.

Helper function to broadcast Target Reset Action (RSTACT) to all connected targets to Reset the I3C Peripheral Only (0x01).

See also

i3c_do_ccc

Parameters

- `controller` – **[in]** Pointer to the controller device driver instance.
- `action` – **[in]** What reset action to perform.

Returns

int `i3c_ccc_do_rstdaa_all`(const struct *device* *controller)

Broadcast RSTDAA to reset dynamic addresses for all targets.

Helper function to reset dynamic addresses of all connected targets.

See also

i3c_do_ccc

Parameters

- `controller` – **[in]** Pointer to the controller device driver instance.

Returns

int `i3c_ccc_do_setdasa`(const struct *i3c_device_desc* *target)

Set Dynamic Address from Static Address for a target.

Helper function to do SETDASA (Set Dynamic Address from Static Address) for a particular target.

Note this does not update `target` with the new dynamic address.

See also

i3c_do_ccc

Parameters

- `target` – **[in]** Pointer to the target device descriptor where the device is configured with a static address.

Returns

int `i3c_ccc_do_setnewda`(const struct *i3c_device_desc* *target, struct *i3c_ccc_address* new_da)

Set New Dynamic Address for a target.

Helper function to do SETNEWDA(Set New Dynamic Address) for a particular target.

Note this does not update `target` with the new dynamic address.

See also

i3c_do_ccc

Parameters

- `target` – **[in]** Pointer to the target device descriptor where the device is configured with a static address.
- `new_da` – **[in]** Pointer to the new_da struct.

Returns

int `i3c_ccc_do_events_all_set`(const struct *device* *controller, bool enable, struct *i3c_ccc_events* *events)

Broadcast ENEC/DISEC to enable/disable target events.

Helper function to broadcast Target Events Command to enable or disable target events (ENEC/DISEC).

See also

i3c_do_ccc

Parameters

- `controller` – **[in]** Pointer to the controller device driver instance.
- `enable` – **[in]** ENEC if true, DISEC if false.
- `events` – **[in]** Pointer to the event struct.

Returns

int `i3c_ccc_do_events_set`(struct *i3c_device_desc* *target, bool enable, struct *i3c_ccc_events* *events)

Direct CCC ENEC/DISEC to enable/disable target events.

Helper function to send Target Events Command to enable or disable target events (ENEC/DISEC) on a single target.

See also

i3c_do_ccc

Parameters

- `target` – **[in]** Pointer to the target device descriptor.
- `enable` – **[in]** ENEC if true, DISEC if false.
- `events` – **[in]** Pointer to the event struct.

Returns

int `i3c_ccc_do_setmwl_all`(const struct *device* *controller, const struct *i3c_ccc_mwl* *mwl)

Broadcast SETMWL to Set Maximum Write Length.

Helper function to do SETMWL (Set Maximum Write Length) to all connected targets.

See also

i3c_do_ccc

Parameters

- `controller` – **[in]** Pointer to the controller device driver instance.
- `mwl` – **[in]** Pointer to SETMWL payload.

Returns

int `i3c_ccc_do_setmwl`(const struct *i3c_device_desc* *target, const struct *i3c_ccc_mwl* *mwl)

Single target SETMWL to Set Maximum Write Length.

Helper function to do SETMWL (Set Maximum Write Length) to one target.

See also

i3c_do_ccc

Parameters

- `target` – **[in]** Pointer to the target device descriptor.
- `mwl` – **[in]** Pointer to SETMWL payload.

Returns

int `i3c_ccc_do_getmwl`(const struct *i3c_device_desc* *target, struct *i3c_ccc_mwl* *mwl)

Single target GETMWL to Get Maximum Write Length.

Helper function to do GETMWL (Get Maximum Write Length) of one target.

See also

i3c_do_ccc

Parameters

- `target` – **[in]** Pointer to the target device descriptor.
- `mwl` – **[out]** Pointer to GETMWL payload.

Returns

int `i3c_ccc_do_setmrl_all`(const struct *device* *controller, const struct *i3c_ccc_mrl* *mrl, bool has_ibi_size)

Broadcast SETMRL to Set Maximum Read Length.

Helper function to do SETMRL (Set Maximum Read Length) to all connected targets.

See also

i3c_do_ccc

Parameters

- `controller` – **[in]** Pointer to the controller device driver instance.
- `mrl` – **[in]** Pointer to SETMRL payload.
- `has_ibi_size` – **[in]** True if also sending the optional IBI payload size. False if not sending.

Returns

int `i3c_ccc_do_setmrl`(const struct *i3c_device_desc* *target, const struct *i3c_ccc_mrl* *mrl)

Single target SETMRL to Set Maximum Read Length.

Helper function to do SETMRL (Set Maximum Read Length) to one target.

Note this uses the BCR of the target to determine whether to send the optional IBI payload size.

See also

i3c_do_ccc

Parameters

- `target` – **[in]** Pointer to the target device descriptor.
- `mrl` – **[in]** Pointer to SETMRL payload.

Returns

int **i3c_ccc_do_getmrl**(const struct *i3c_device_desc* *target, struct *i3c_ccc_mrl* *mrl)

Single target GETMRL to Get Maximum Read Length.

Helper function to do GETMRL (Get Maximum Read Length) of one target.

Note this uses the BCR of the target to determine whether to send the optional IBI payload size.

See also

i3c_do_ccc

Parameters

- **target** – **[in]** Pointer to the target device descriptor.
- **mrl** – **[out]** Pointer to GETMRL payload.

Returns

int **i3c_ccc_do_getstatus**(const struct *i3c_device_desc* *target, union *i3c_ccc_getstatus* *status, enum *i3c_ccc_getstatus_fmt* fmt, enum *i3c_ccc_getstatus_defbyte* defbyte)

Single target GETSTATUS to Get Target Status.

Helper function to do GETSTATUS (Get Target Status) of one target.

Note this uses the BCR of the target to determine whether to send the optional IBI payload size.

See also

i3c_do_ccc

Parameters

- **target** – **[in]** Pointer to the target device descriptor.
- **status** – **[out]** Pointer to GETSTATUS payload.
- **fmt** – **[in]** Which GETSTATUS to use.
- **defbyte** – **[in]** Defining Byte if using format 2.

Returns

static inline int **i3c_ccc_do_getstatus_fmt1**(const struct *i3c_device_desc* *target, union *i3c_ccc_getstatus* *status)

Single target GETSTATUS to Get Target Status (Format 1).

Helper function to do GETSTATUS (Get Target Status, format 1) of one target.

See also

i3c_do_ccc

Parameters

- `target` – **[in]** Pointer to the target device descriptor.
- `status` – **[out]** Pointer to GETSTATUS payload.

Returns

static inline int `i3c_ccc_do_getstatus_fmt2`(const struct *i3c_device_desc* *target, union *i3c_ccc_getstatus* *status, enum *i3c_ccc_getstatus_defbyte* defbyte)

Single target GETSTATUS to Get Target Status (Format 2).

Helper function to do GETSTATUS (Get Target Status, format 2) of one target.

See also

i3c_do_ccc

Parameters

- `target` – **[in]** Pointer to the target device descriptor.
- `status` – **[out]** Pointer to GETSTATUS payload.
- `defbyte` – **[in]** Defining Byte for GETSTATUS format 2.

Returns

int `i3c_ccc_do_getcaps`(const struct *i3c_device_desc* *target, union *i3c_ccc_getcaps* *caps, enum *i3c_ccc_getcaps_fmt* fmt, enum *i3c_ccc_getcaps_defbyte* defbyte)

Single target GETCAPS to Get Target Status.

Helper function to do GETCAPS (Get Capabilities) of one target.

This should only be supported if Advanced Capabilities Bit of the BCR is set

See also

i3c_do_ccc

Parameters

- `target` – **[in]** Pointer to the target device descriptor.
- `caps` – **[out]** Pointer to GETCAPS payload.
- `fmt` – **[in]** Which GETCAPS to use.
- `defbyte` – **[in]** Defining Byte if using format 2.

Returns

static inline int `i3c_ccc_do_getcaps_fmt1`(const struct *i3c_device_desc* *target, union *i3c_ccc_getcaps* *caps)

Single target GETCAPS to Get Capabilities (Format 1).

Helper function to do GETCAPS (Get Capabilities, format 1) of one target.

See also

i3c_do_ccc

Parameters

- `target` – **[in]** Pointer to the target device descriptor.
- `caps` – **[out]** Pointer to GETCAPS payload.

Returns

static inline int `i3c_ccc_do_getcaps_fmt2`(const struct *i3c_device_desc* *target, union *i3c_ccc_getcaps* *caps, enum *i3c_ccc_getcaps_defbyte* defbyte)

Single target GETCAPS to Get Capabilities (Format 2).

Helper function to do GETCAPS (Get Capabilities, format 2) of one target.

See also

i3c_do_ccc

Parameters

- `target` – **[in]** Pointer to the target device descriptor.
- `caps` – **[out]** Pointer to GETCAPS payload.
- `defbyte` – **[in]** Defining Byte for GETCAPS format 2.

Returns

struct `i3c_ccc_target_payload`

#include <*ccc.h*> Payload structure for Direct CCC to one target.

Public Members

uint8_t `addr`

Target address.

uint8_t `rnw`

`0` for Write, `1` for Read

uint8_t *`data`

- For Write CCC, pointer to the byte array of data to be sent, which may contain the Sub-Command Byte and additional data.
- For Read CCC, pointer to the byte buffer for data to be read into.

size_t `data_len`

Length in bytes for `data`.

size_t **num_xfer**

Total number of bytes transferred.

A Target can issue an EoD or the Controller can abort a transfer before the length of the buffer. It is expected for the driver to write to this after the transfer.

struct **i3c_ccc_payload**

#include <*ccc.h*> Payload structure for one CCC transaction.

Public Members

uint8_t **id**

The CCC ID (`I3C_CCC_*`).

uint8_t ***data**

Pointer to byte array of data for this CCC.

This is the bytes following the CCC command in CCC frame. Set to `NULL` if no associated data.

size_t **data_len**

Length in bytes for optional data array.

size_t **num_xfer**

Total number of bytes transferred.

A Controller can abort a transfer before the length of the buffer. It is expected for the driver to write to this after the transfer.

struct *i3c_ccc_target_payload* ***payloads**

Array of struct *i3c_ccc_target_payload*.

Each element describes the target and associated payloads for this CCC.

Use with Direct CCC.

size_t **num_targets**

Number of targets.

struct **i3c_ccc_events**

#include <*ccc.h*> Payload for ENEC/DISEC CCC (Target Events Command).

Public Members

uint8_t **events**

Event byte:

- Bit[0]: ENINT/DISINT:
  - Target Interrupt Requests
- Bit[1]: ENCR/DISCR:
  - Controller Role Requests

- Bit[3]: ENHJ/DISHJ:
  - Hot-Join Event

struct `i3c_ccc_mwl`

#include <*ccc.h*> Payload for SETMWL/GETMWL CCC (Set/Get Maximum Write Length).

> **Note**
>
> For drivers and help functions, the raw data coming back from target device is in big endian. This needs to be translated back to CPU endianness before passing back to function caller.

Public Members

uint16_t `len`

Maximum Write Length.

struct `i3c_ccc_mrl`

#include <*ccc.h*> Payload for SETMRL/GETMRL CCC (Set/Get Maximum Read Length).

> **Note**
>
> For drivers and help functions, the raw data coming back from target device is in big endian. This needs to be translated back to CPU endianness before passing back to function caller.

Public Members

uint16_t `len`

Maximum Read Length.

uint8_t `ibi_len`

Optional IBI Payload Size.

struct `i3c_ccc_deftgts_active_controller`

#include <*ccc.h*> The active controller part of payload for DEFTGTS CCC.

This is used by DEFTGTS (Define List of Targets) CCC to describe the active controller on the I3C bus.

Public Members

uint8_t `addr`

Dynamic Address of Active Controller.

uint8_t `dcr`

Device Characteristic Register of Active Controller.

uint8_t **bcr**

Bus Characteristic Register of Active Controller.

uint8_t **static_addr**

Static Address of Active Controller.

struct **i3c_ccc_deftgts_target**

#include <*ccc.h*> The target device part of payload for DEFTGTS CCC.

This is used by DEFTGTS (Define List of Targets) CCC to describe the existing target devices on the I3C bus.

Public Members

uint8_t **addr**

Dynamic Address of a target device, or a group address.

uint8_t **dcr**

Device Characteristic Register of a I3C target device or a group.

uint8_t **lvr**

Legacy Virtual Register for legacy I2C device.

uint8_t **bcr**

Bus Characteristic Register of a target device or a group.

uint8_t **static_addr**

Static Address of a target device or a group.

struct **i3c_ccc_deftgts**

#include <*ccc.h*> Payload for DEFTGTS CCC (Define List of Targets).

Note

i3c_ccc_deftgts_target is an array of targets, where the number of elements is dependent on the number of I3C targets on the bus. Please have enough space for both read and write of this CCC.

Public Members

struct *i3c_ccc_deftgts_active_controller* **active_controller**

Data describing the active controller.

struct *i3c_ccc_deftgts_target* **targets**[]

Data describing the target(s) on the bus.

struct `i3c_ccc_address`

#include <*ccc.h*> Payload for a single device address.

This is used for:

- SETDASA (Set Dynamic Address from Static Address)
- SETNEWDA (Set New Dynamic Address)
- SETGRPA (Set Group Address)
- GETACCCR (Get Accept Controller Role)

Note that the target address is encoded within struct *i3c_ccc_target_payload* instead of being encoded in this payload.

Public Members

uint8_t `addr`

- For SETDASA, Static Address to be assigned as Dynamic Address.
- For SETNEWDA, new Dynamic Address to be assigned.
- For SETGRPA, new Group Address to be set.
- For GETACCCR, the correct address of Secondary Controller.

> **Note**
>
> For SETDATA, SETNEWDA and SETGRAP, the address is left-shift by 1, and bit[0] is always 0.

> **Note**
>
> Fpr SET GETACCCR, the address is left-shift by 1, and bit[0] is the calculated odd parity bit.

struct `i3c_ccc_getpid`

#include <*ccc.h*> Payload for GETPID CCC (Get Provisioned ID).

Public Members

uint8_t `pid`[6]

48-bit Provisioned ID.

> **Note**
>
> Data is big-endian where first byte is MSB.

struct `i3c_ccc_getbcr`

#include <*ccc.h*> Payload for GETBCR CCC (Get Bus Characteristics Register).

Public Members

uint8_t **bcr**

Bus Characteristics Register.

struct **i3c_ccc_getdcr**

#include <*ccc.h*> Payload for GETDCR CCC (Get Device Characteristics Register).

Public Members

uint8_t **dcr**

Device Characteristics Register.

union **i3c_ccc_getstatus**

#include <*ccc.h*> Payload for GETSTATUS CCC (Get Device Status).

Public Members

uint16_t **status**

Device Status.

- Bit[15:8]: Reserved.
- Bit[7:6]: Activity Mode.
- Bit[5]: Protocol Error.
- Bit[4]: Reserved.
- Bit[3:0]: Number of Pending Interrupts.

Note

For drivers and help functions, the raw data coming back from target device is in big endian. This needs to be translated back to CPU endianness before passing back to function caller.

struct *i3c_ccc_getstatus* **fmt1**

uint16_t **tgtstat**

Defining Byte 0x00: TGTSTAT.

See also

i3c_ccc_getstatus::fmt1::status

uint16_t **precr**

Defining Byte 0x91: PRECR.

- Bit[15:8]: Vendor Reserved
- Bit[7:2]: Reserved
- Bit[1]: Handoff Delay NACK
- Bit[0]: Deep Sleep Detected

> **Note**
>
> For drivers and help functions, the raw data coming back from target device is in big endian. This needs to be translated back to CPU endianness before passing back to function caller.

uint16_t `raw_u16`

union *i3c_ccc_getstatus* `fmt2`

struct `i3c_ccc_setbrgtgt_tgt`

#include <*ccc.h*> One Bridged Target for SETBRGTGT payload.

Public Members

uint8_t `addr`

Dynamic address of the bridged target.

> **Note**
>
> The address is left-shift by 1, and bit[0] is always 0.

uint16_t `id`

16-bit ID for the bridged target.

> **Note**
>
> For drivers and help functions, the raw data coming back from target device is in big endian. This needs to be translated back to CPU endianness before passing back to function caller.

struct `i3c_ccc_setbrgtgt`

#include <*ccc.h*> Payload for SETBRGTGT CCC (Set Bridge Targets).

Note that the bridge target address is encoded within struct *i3c_ccc_target_payload* instead of being encoded in this payload.

Public Members

uint8_t `count`

Number of bridged targets.

struct *i3c_ccc_setbrgtgt_tgt* **targets**[]
Array of bridged targets.

union **i3c_ccc_getmxds**
#include <*ccc.h*> Payload for GETMXDS CCC (Get Max Data Speed).

> **Note**
> This is only for GETMXDS Format 1 and Format 2.

Public Members

uint8_t **maxwr**
maxWr

uint8_t **maxrd**
maxRd

struct *i3c_ccc_getmxds* **fmt1**

uint8_t **maxrdturn**[3]
Maximum Read Turnaround Time in microsecond.
This is in little-endian where first byte is LSB.

struct *i3c_ccc_getmxds* **fmt2**

uint8_t **wrrdturn**
Defining Byte 0x00: WRRDTURN.

> **See also**
> *i3c_ccc_getmxds::fmt2*

uint8_t **crhdly1**
Defining Byte 0x91: CRHDLY.

- Bit[2]: Set Bus Activity State
- Bit[1:0]: Controller Handoff Activity State

struct *i3c_ccc_getmxds* **fmt3**

union **i3c_ccc_getcaps**
#include <*ccc.h*> Payload for GETCAPS CCC (Get Optional Feature Capabilities).

Note

Only supports GETCAPS Format 1 and Format 2. In I3C v1.0 this was GETHDRCAP which only returned a single byte which is the same as the GETCAPS1 byte.

Public Members

uint8_t **gethdrcap**

I3C v1.0 HDR Capabilities.

- Bit[0]: HDR-DDR
- Bit[1]: HDR-TSP
- Bit[2]: HDR-TSL
- Bit[7:3]: Reserved

uint8_t **getcaps**[4]

I3C v1.1+ Device Capabilities Byte 1 GETCAPS1.

- Bit[0]: HDR-DDR
- Bit[1]: HDR-TSP
- Bit[2]: HDR-TSL
- Bit[3]: HDR-BT
- Bit[7:4]: Reserved Byte 2 GETCAPS2
- Bit[3:0]: I3C 1.x Specification Version
- Bit[5:4]: Group Address Capabilities
- Bit[6]: HDR-DDR Write Abort
- Bit[7]: HDR-DDR Abort CRC Byte 3 GETCAPS3
- Bit[0]: Multi-Lane (ML) Data Transfer Support
- Bit[1]: Device to Device Transfer (D2DXFER) Support
- Bit[2]: Device to Device Transfer (D2DXFER) IBI Capable
- Bit[3]: Defining Byte Support in GETCAPS
- Bit[4]: Defining Byte Support in GETSTATUS
- Bit[5]: HDR-BT CRC-32 Support
- Bit[6]: IBI MDB Support for Pending Read Notification
- Bit[7]: Reserved Byte 4 GETCAPS4
- Bit[7:0]: Reserved

union *i3c_ccc_getcaps* **fmt1**

uint8_t **tgtcaps**[4]

Defining Byte 0x00: TGTCAPS.

See also

i3c_ccc_getcaps::fmt1::getcaps

uint32_t **testpat**

Defining Byte 0x5A: TESTPAT.

> **Note**
>
> should always be 0xA55AA55A in big endian

uint8_t **crcaps**[2]

Defining Byte 0x91: CRCAPS Byte 1 CRCAPS1.

- Bit[0]: Hot-Join Support
- Bit[1]: Group Management Support
- Bit[2]: Multi-Lane Support Byte 2 CRCAPS2
- Bit[0]: In-Band Interrupt Support
- Bit[1]: Controller Pass-Back
- Bit[2]: Deep Sleep Capable
- Bit[3]: Delayed Controller Handoff

uint8_t **vtcaps**[2]

Defining Byte 0x93: VTCAPS Byte 1 VTCAPS1.

- Bit[2:0]: Virtual Target Type
- Bit[4]: Side Effects
- Bit[5]: Shared Peripheral Detect Byte 2 VTCAPS2
- Bit[1:0]: Interrupt Requests
- Bit[2]: Address Remapping
- Bit[4:3]: Bus Context and Conditions

union *i3c_ccc_getcaps* **fmt2**

group **i3c_addresses**

I3C Address-related Helper Code.

Defines

I3C_BROADCAST_ADDR

Broadcast Address on I3C bus.

I3C_MAX_ADDR

Maximum value of device addresses.

Enums

enum **i3c_addr_slot_status**

Enum to indicate whether an address is reserved, has I2C/I3C device attached, or no device attached.

Values:

enumerator **I3C_ADDR_SLOT_STATUS_FREE** = 0U

Address has not device attached.

enumerator I3C_ADDR_SLOT_STATUS_RSVD
Address is reserved.

enumerator I3C_ADDR_SLOT_STATUS_I3C_DEV
Address is associated with an I3C device.

enumerator I3C_ADDR_SLOT_STATUS_I2C_DEV
Address is associated with an I2C device.

enumerator I3C_ADDR_SLOT_STATUS_MASK = 0x03U
Bit masks used to filter status bits.

Functions

int i3c_addr_slots_init(const struct *device* *dev)
Initialize the I3C address slots struct.

This clears out the assigned address bits, and set the reserved address bits according to the I3C specification.

Parameters

- dev – Pointer to controller device driver instance.

Return values

- 0 – if successful.
- -EINVAL – if duplicate addresses.

void i3c_addr_slots_set(struct *i3c_addr_slots* *slots, uint8_t dev_addr, enum *i3c_addr_slot_status* status)
Set the address status of a device.

Parameters

- slots – Pointer to the address slots structure.
- dev_addr – Device address.
- status – New status for the address dev_addr.

enum *i3c_addr_slot_status* i3c_addr_slots_status(struct *i3c_addr_slots* *slots, uint8_t dev_addr)
Get the address status of a device.

Parameters

- slots – Pointer to the address slots structure.
- dev_addr – Device address.

Returns
Address status for the address dev_addr.

bool i3c_addr_slots_is_free(struct *i3c_addr_slots* *slots, uint8_t dev_addr)
Check if the address is free.

Parameters

- slots – Pointer to the address slots structure.
- dev_addr – Device address.

Return values

- `true` – if address is free.
- `false` – if address is not free.

uint8_t `i3c_addr_slots_next_free_find`(struct *i3c_addr_slots* *slots, uint8_t start_addr)

Find the next free address.

This can be used to find the next free address that can be assigned to a new device.

Parameters

- `slots` – Pointer to the address slots structure.
- `start_addr` – Where to start searching

Returns

The next free address, or 0 if none found.

static inline void `i3c_addr_slots_mark_free`(struct *i3c_addr_slots* *addr_slots, uint8_t addr)

Mark the address as free (not used) in device list.

Parameters

- `addr_slots` – Pointer to the address slots struct.
- `addr` – Device address.

static inline void `i3c_addr_slots_mark_rsvd`(struct *i3c_addr_slots* *addr_slots, uint8_t addr)

Mark the address as reserved in device list.

Parameters

- `addr_slots` – Pointer to the address slots struct.
- `addr` – Device address.

static inline void `i3c_addr_slots_mark_i3c`(struct *i3c_addr_slots* *addr_slots, uint8_t addr)

Mark the address as I3C device in device list.

Parameters

- `addr_slots` – Pointer to the address slots struct.
- `addr` – Device address.

static inline void `i3c_addr_slots_mark_i2c`(struct *i3c_addr_slots* *addr_slots, uint8_t addr)

Mark the address as I2C device in device list.

Parameters

- `addr_slots` – Pointer to the address slots struct.
- `addr` – Device address.

struct `i3c_addr_slots`

#include <*addresses.h*> Structure to keep track of addresses on I3C bus.

group `i3c_target_device`

I3C Target Device API.

Functions

static inline int **i3c_target_tx_write**(const struct *device* *dev, uint8_t *buf, uint16_t len)

Writes to the target's TX FIFO.

Write to the TX FIFO `dev` I3C bus driver using the provided buffer and length. Some I3C targets will NACK read requests until data is written to the TX FIFO. This function will write as much as it can to the FIFO return the total number of bytes written. It is then up to the application to utalize the target callbacks to write the remaining data. Negative returns indicate error.

Most of the existing hardware allows simultaneous support for master and target mode. This is however not guaranteed.

Parameters

- `dev` – Pointer to the device structure for an I3C controller driver configured in target mode.
- `buf` – Pointer to the buffer
- `len` – Length of the buffer

Return values

- `Total` – number of bytes written
- `-ENOTSUP` – Not in Target Mode
- `-ENOSPC` – No space in Tx FIFO
- `-ENOSYS` – If target mode is not implemented

static inline int **i3c_target_register**(const struct *device* *dev, struct *i3c_target_config* *cfg)

Registers the provided config as target device of a controller.

Enable I3C target mode for the `dev` I3C bus driver using the provided config struct (`cfg`) containing the functions and parameters to send bus events. The I3C target will be registered at the address provided as *i3c_target_config::address* struct member. Any I3C bus events related to the target mode will be passed onto I3C target device driver via a set of callback functions provided in the 'callbacks' struct member.

Most of the existing hardware allows simultaneous support for master and target mode. This is however not guaranteed.

Parameters

- `dev` – Pointer to the device structure for an I3C controller driver configured in target mode.
- `cfg` – Config struct with functions and parameters used by the I3C target driver to send bus events

Return values

- `0` – Is successful
- `-EINVAL` – If parameters are invalid
- `-EIO` – General input / output error.
- `-ENOSYS` – If target mode is not implemented

static inline int **i3c_target_unregister**(const struct *device* *dev, struct *i3c_target_config* *cfg)

Unregisters the provided config as target device.

This routine disables I3C target mode for the dev I3C bus driver using the provided config struct (cfg) containing the functions and parameters to send bus events.

Parameters

- dev – Pointer to the device structure for an I3C controller driver configured in target mode.
- cfg – Config struct with functions and parameters used by the I3C target driver to send bus events

Return values

- 0 – Is successful
- -EINVAL – If parameters are invalid
- -ENOSYS – If target mode is not implemented

struct i3c_config_target

#include <target_device.h> Configuration parameters for I3C hardware to act as target device.

This can also be used to configure the controller if it is to act as a secondary controller on the bus.

Public Members

bool enable

If the hardware is to act as a target device on the bus.

uint8_t static_addr

I3C target address.

Used used when operates as secondary controller or as a target device.

uint64_t pid

Provisioned ID.

bool pid_random

True if lower 32-bit of Provisioned ID is random.

This sets the bit 32 of Provisioned ID which means the lower 32-bit is random value.

uint8_t bcr

Bus Characteristics Register (BCR).

uint8_t dcr

Device Characteristics Register (DCR).

uint16_t max_read_len

Maximum Read Length (MRL).

uint16_t max_write_len

Maximum Write Length (MWL).

uint8_t **supported_hdr**

Bit mask of supported HDR modes (0 - 7).

This can be used to enable or disable HDR mode supported by the hardware at runtime.

struct **i3c_target_config**

#include <target_device.h> Structure describing a device that supports the I3C target API.

Instances of this are passed to the *i3c_target_register()* and *i3c_target_unregister()* functions to indicate addition and removal of a target device, respective.

Fields other than `node` must be initialized by the module that implements the device behavior prior to passing the object reference to *i3c_target_register()*.

Public Members

sys_snode_t **node**

Private, do not modify.

uint8_t **flags**

Flags for the target device defined by I3C_TARGET_FLAGS_* constants.

uint8_t **address**

Address for this target device.

const struct *i3c_target_callbacks* ***callbacks**

Callback functions.

struct **i3c_target_callbacks**

#include <target_device.h>

Public Members

int (***write_requested_cb**)(struct *i3c_target_config* *config)

Function called when a write to the device is initiated.

This function is invoked by the controller when the bus completes a start condition for a write operation to the address associated with a particular device.

A success return shall cause the controller to ACK the next byte received. An error return shall cause the controller to NACK the next byte received.

Param config
Configuration structure associated with the device to which the operation is addressed.

Return
0 if the write is accepted, or a negative error code.

int (***write_received_cb**)(struct *i3c_target_config* *config, uint8_t val)

Function called when a write to the device is continued.

This function is invoked by the controller when it completes reception of a byte of data in an ongoing write operation to the device.

A success return shall cause the controller to ACK the next byte received. An error return shall cause the controller to NACK the next byte received.

Param config
Configuration structure associated with the device to which the operation is addressed.

Param val
the byte received by the controller.

Return
0 if more data can be accepted, or a negative error code.

int (*`read_requested_cb`)(struct *i3c_target_config* *config, uint8_t *val)

Function called when a read from the device is initiated.

This function is invoked by the controller when the bus completes a start condition for a read operation from the address associated with a particular device.

The value returned in `val` will be transmitted. A success return shall cause the controller to react to additional read operations. An error return shall cause the controller to ignore bus operations until a new start condition is received.

Param config
Configuration structure associated with the device to which the operation is addressed.

Param val
Pointer to storage for the first byte of data to return for the read request.

Return
0 if more data can be requested, or a negative error code.

int (*`read_processed_cb`)(struct *i3c_target_config* *config, uint8_t *val)

Function called when a read from the device is continued.

This function is invoked by the controller when the bus is ready to provide additional data for a read operation from the address associated with the device device.

The value returned in `val` will be transmitted. A success return shall cause the controller to react to additional read operations. An error return shall cause the controller to ignore bus operations until a new start condition is received.

Param config
Configuration structure associated with the device to which the operation is addressed.

Param val
Pointer to storage for the next byte of data to return for the read request.

Return
0 if data has been provided, or a negative error code.

int (*`stop_cb`)(struct *i3c_target_config* *config)

Function called when a stop condition is observed after a start condition addressed to a particular device.

This function is invoked by the controller when the bus is ready to provide additional data for a read operation from the address associated with the device device. After the function returns the controller shall enter a state where it is ready to react to new start conditions.

Param config
Configuration structure associated with the device to which the operation is addressed.

Return
Ignored.

struct `i3c_target_driver_api`
#include <target_device.h>

7.6.26 Inter-Integrated Circuit (I2C) Bus

Overview

Note

The terminology used in Zephyr I2C APIs follows that of the NXP I2C Bus Specification Rev 7.0. These changed from previous revisions as of its release October 1, 2021.

I2C (Inter-Integrated Circuit, pronounced "eye squared see") is a commonly-used two-signal shared peripheral interface bus. Many system-on-chip solutions provide controllers that communicate on an I2C bus. Devices on the bus can operate in two roles: as a "controller" that initiates transactions and controls the clock, or as a "target" that responds to transaction commands. A I2C controller on a given SoC will generally support the controller role, and some will also support the target mode. Zephyr has API for both roles.

I2C Controller API Zephyr's I2C controller API is used when an I2C peripheral controls the bus, in particularly the start and stop conditions and the clock. This is the most common mode, used to interact with I2C devices like sensors and serial memory.

This API is supported in all in-tree I2C peripheral drivers and is considered stable.

I2C Target API Zephyr's I2C target API is used when an I2C peripheral responds to transactions initiated by a different controller on the bus. It might be used for a Zephyr application with transducer roles that are controlled by another device such as a host processor.

This API is supported in very few in-tree I2C peripheral drivers. The API is considered experimental, as it is not compatible with the capabilities of all I2C peripherals supported in controller mode.

Configuration Options

Related configuration options:

- `CONFIG_I2C`

API Reference

Related code samples

I2C Custom Target
Setup a custom I2C target on the I2C interface.

I2C Target
Setup an I2C target on the I2C interface.

STM32 I2C V2 timings

group `i2c_interface`

I2C Interface.

Since
1.0

Version
1.0.0

Defines

`I2C_SPEED_STANDARD`
I2C Standard Speed: 100 kHz.

`I2C_SPEED_FAST`
I2C Fast Speed: 400 kHz.

`I2C_SPEED_FAST_PLUS`
I2C Fast Plus Speed: 1 MHz.

`I2C_SPEED_HIGH`
I2C High Speed: 3.4 MHz.

`I2C_SPEED_ULTRA`
I2C Ultra Fast Speed: 5 MHz.

`I2C_SPEED_DT`
Device Tree specified speed.

`I2C_SPEED_SHIFT`

`I2C_SPEED_SET`(speed)

`I2C_SPEED_MASK`

`I2C_SPEED_GET`(cfg)

`I2C_ADDR_10_BITS`
Use 10-bit addressing.

DEPRECATED - Use I2C_MSG_ADDR_10_BITS instead.

`I2C_MODE_CONTROLLER`
Peripheral to act as Controller.

`I2C_DT_SPEC_GET_ON_I3C`(node_id)
Structure initializer for *i2c_dt_spec* from devicetree (on I3C bus)

This helper macro expands to a static initializer for a `struct` *i2c_dt_spec* by reading the relevant bus and address data from the devicetree.

Parameters

- **node_id** – Devicetree node identifier for the I2C device whose struct *i2c_dt_spec* to create an initializer for

I2C_DT_SPEC_GET_ON_I2C(node_id)

Structure initializer for *i2c_dt_spec* from devicetree (on I2C bus)

This helper macro expands to a static initializer for a `struct` *i2c_dt_spec* by reading the relevant bus and address data from the devicetree.

Parameters

- **node_id** – Devicetree node identifier for the I2C device whose struct *i2c_dt_spec* to create an initializer for

I2C_DT_SPEC_GET(node_id)

Structure initializer for *i2c_dt_spec* from devicetree.

This helper macro expands to a static initializer for a `struct` *i2c_dt_spec* by reading the relevant bus and address data from the devicetree.

Parameters

- **node_id** – Devicetree node identifier for the I2C device whose struct *i2c_dt_spec* to create an initializer for

I2C_DT_SPEC_INST_GET(inst)

Structure initializer for *i2c_dt_spec* from devicetree instance.

This is equivalent to *I2C_DT_SPEC_GET(DT_DRV_INST(inst))*.

Parameters

- **inst** – Devicetree instance number

I2C_MSG_WRITE

Write message to I2C bus.

I2C_MSG_READ

Read message from I2C bus.

I2C_MSG_STOP

Send STOP after this message.

I2C_MSG_RESTART

RESTART I2C transaction for this message.

Note

Not all I2C drivers have or require explicit support for this feature. Some drivers require this be present on a read message that follows a write, or vice-versa. Some drivers will merge adjacent fragments into a single transaction using this flag; some will not.

I2C_MSG_ADDR_10_BITS

Use 10-bit addressing for this message.

Note

Not all SoC I2C implementations support this feature.

I2C_TARGET_FLAGS_ADDR_10_BITS

Target device responds to 10-bit addressing.

I2C_DEVICE_DT_DEFINE(node_id, init_fn, pm, data, config, level, prio, api, ...)

Like *DEVICE_DT_DEFINE()* with I2C specifics.

Defines a device which implements the I2C API. May generate a custom *device_state* container struct and init_fn wrapper when needed depending on I2C CONFIG_I2C_STATS.

Parameters

- **node_id** – The devicetree node identifier.
- **init_fn** – Name of the init function of the driver. Can be NULL.
- **pm** – PM device resources reference (NULL if device does not use PM).
- **data** – Pointer to the device's private data.
- **config** – The address to the structure containing the configuration information for this instance of the driver.
- **level** – The initialization level. See SYS_INIT() for details.
- **prio** – Priority within the selected initialization level. See SYS_INIT() for details.
- **api** – Provides an initial pointer to the API function struct used by the driver. Can be NULL.

I2C_DEVICE_DT_INST_DEFINE(inst, ...)

Like *I2C_DEVICE_DT_DEFINE()* for an instance of a DT_DRV_COMPAT compatible.

Parameters

- **inst** – instance number. This is replaced by DT_DRV_COMPAT(inst) in the call to *I2C_DEVICE_DT_DEFINE()*.
- **...** – other parameters as expected by *I2C_DEVICE_DT_DEFINE()*.

I2C_DT_IODEV_DEFINE(name, node_id)

Define an iodev for a given dt node on the bus.

These do not need to be shared globally but doing so will save a small amount of memory.

Parameters

- **name** – Symbolic name of the iodev to define
- **node_id** – Devicetree node identifier

I2C_IODEV_DEFINE(name, _bus, _addr)

Define an iodev for a given i2c device on a bus.

These do not need to be shared globally but doing so will save a small amount of memory.

Parameters

- **name** – Symbolic name of the iodev to define
- **_bus** – Node ID for I2C bus
- **_addr** – I2C target address

Typedefs

typedef void (*`i2c_callback_t`)(const struct *device* *dev, int result, void *data)

I2C callback for asynchronous transfer requests.

Param dev
I2C device which is notifying of transfer completion or error

Param result
Result code of the transfer request. 0 is success, -errno for failure.

Param data
Transfer requester supplied data which is passed along to the callback.

typedef int (*`i2c_target_write_requested_cb_t`)(struct *i2c_target_config* *config)

Function called when a write to the device is initiated.

This function is invoked by the controller when the bus completes a start condition for a write operation to the address associated with a particular device.

A success return shall cause the controller to ACK the next byte received. An error return shall cause the controller to NACK the next byte received.

Param config
the configuration structure associated with the device to which the operation is addressed.

Return
0 if the write is accepted, or a negative error code.

typedef int (*`i2c_target_write_received_cb_t`)(struct *i2c_target_config* *config, uint8_t val)

Function called when a write to the device is continued.

This function is invoked by the controller when it completes reception of a byte of data in an ongoing write operation to the device.

A success return shall cause the controller to ACK the next byte received. An error return shall cause the controller to NACK the next byte received.

Param config
the configuration structure associated with the device to which the operation is addressed.

Param val
the byte received by the controller.

Return
0 if more data can be accepted, or a negative error code.

typedef int (*`i2c_target_read_requested_cb_t`)(struct *i2c_target_config* *config, uint8_t *val)

Function called when a read from the device is initiated.

This function is invoked by the controller when the bus completes a start condition for a read operation from the address associated with a particular device.

The value returned in `*val` will be transmitted. A success return shall cause the controller to react to additional read operations. An error return shall cause the controller to ignore bus operations until a new start condition is received.

Param config
the configuration structure associated with the device to which the operation is addressed.

Param val
pointer to storage for the first byte of data to return for the read request.

Return
0 if more data can be requested, or a negative error code.

typedef int (*`i2c_target_read_processed_cb_t`)(struct *i2c_target_config* *config, uint8_t *val)

Function called when a read from the device is continued.

This function is invoked by the controller when the bus is ready to provide additional data for a read operation from the address associated with the device device.

The value returned in `*val` will be transmitted. A success return shall cause the controller to react to additional read operations. An error return shall cause the controller to ignore bus operations until a new start condition is received.

Param config
the configuration structure associated with the device to which the operation is addressed.

Param val
pointer to storage for the next byte of data to return for the read request.

Return
0 if data has been provided, or a negative error code.

typedef int (*`i2c_target_stop_cb_t`)(struct *i2c_target_config* *config)

Function called when a stop condition is observed after a start condition addressed to a particular device.

This function is invoked by the controller when the bus is ready to provide additional data for a read operation from the address associated with the device device. After the function returns the controller shall enter a state where it is ready to react to new start conditions.

Param config
the configuration structure associated with the device to which the operation is addressed.

Return
Ignored.

Functions

static inline bool `i2c_is_ready_dt`(const struct *i2c_dt_spec* *spec)

Validate that I2C bus is ready.

Parameters

- `spec` – I2C specification from devicetree

Return values

- `true` – if the I2C bus is ready for use.
- `false` – if the I2C bus is not ready for use.

static inline bool **i2c_is_read_op**(struct *i2c_msg* *msg)

Check if the current message is a read operation.

Parameters

- **msg** – The message to check

Returns
true if the I2C message is sa read operation

Returns
false if the I2C message is a write operation

void **i2c_dump_msgs_rw**(const struct *device* *dev, const struct *i2c_msg* *msgs, uint8_t num_msgs, uint16_t addr, bool dump_read)

Dump out an I2C message.

Dumps out a list of I2C messages. For any that are writes (W), the data is displayed in hex. Setting dump_read will dump the data for read messages too, which only makes sense when called after the messages have been processed.

It looks something like this (with name "testing"):

```
D: I2C msg: testing, addr=56
D:    W len=01: 06
D:    W len=0e:
D: contents:
D: 00 01 02 03 04 05 06 07 |........
D: 08 09 0a 0b 0c 0d       |......
D:    W len=01: 0f
D:    R len=01: 6c
```

Parameters

- **dev** – Target for the messages being sent. Its name will be printed in the log.
- **msgs** – Array of messages to dump.
- **num_msgs** – Number of messages to dump.
- **addr** – Address of the I2C target device.
- **dump_read** – Dump data from I2C reads, otherwise only writes have data dumped.

static inline void **i2c_dump_msgs**(const struct *device* *dev, const struct *i2c_msg* *msgs, uint8_t num_msgs, uint16_t addr)

Dump out an I2C message, before it is executed.

This is equivalent to:

```
i2c_dump_msgs_rw(dev, msgs, num_msgs, addr, false);
```

The read messages' data isn't dumped.

Parameters

- **dev** – Target for the messages being sent. Its name will be printed in the log.
- **msgs** – Array of messages to dump.
- **num_msgs** – Number of messages to dump.
- **addr** – Address of the I2C target device.

static inline void **i2c_xfer_stats**(const struct *device* *dev, struct *i2c_msg* *msgs, uint8_t num_msgs)

Updates the i2c stats for i2c transfers.

Parameters

- **dev** – I2C device to update stats for
- **msgs** – Array of struct *i2c_msg*
- **num_msgs** – Number of i2c_msgs

int **i2c_configure**(const struct *device* *dev, uint32_t dev_config)

Configure operation of a host controller.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **dev_config** – Bit-packed 32-bit value to the device runtime configuration for the I2C controller.

Return values

- **0** – If successful.
- **-EIO** – General input / output error, failed to configure device.

int **i2c_get_config**(const struct *device* *dev, uint32_t *dev_config)

Get configuration of a host controller.

This routine provides a way to get current configuration. It is allowed to call the function before i2c_configure, because some I2C ports can be configured during init process. However, if the I2C port is not configured, i2c_get_config returns an error.

i2c_get_config can return cached config or probe hardware, but it has to be up to date with current configuration.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **dev_config** – Pointer to return bit-packed 32-bit value of the I2C controller configuration.

Return values

- **0** – If successful.
- **-EIO** – General input / output error.
- **-ERANGE** – Configured I2C frequency is invalid.
- **-ENOSYS** – If get config is not implemented

int **i2c_transfer**(const struct *device* *dev, struct *i2c_msg* *msgs, uint8_t num_msgs, uint16_t addr)

Perform data transfer to another I2C device in controller mode.

This routine provides a generic interface to perform data transfer to another I2C device synchronously. Use *i2c_read()*/i2c_write() for simple read or write.

The array of message *msgs* must not be NULL. The number of message *num_msgs* may be zero,in which case no transfer occurs.

Note

Not all scatter/gather transactions can be supported by all drivers. As an example, a gather write (multiple consecutive *i2c_msg* buffers all configured for

I2C_MSG_WRITE) may be packed into a single transaction by some drivers, but others may emit each fragment as a distinct write transaction, which will not produce the same behavior. See the documentation of struct *i2c_msg* for limitations on support for multi-message bus transactions.

Note

The last message in the scatter/gather transaction implies a STOP whether or not it is explicitly set. This ensures the bus is in a good state for the next transaction which may be from a different call context.

Parameters

- **dev** – Pointer to the device structure for an I2C controller driver configured in controller mode.
- **msgs** – Array of messages to transfer.
- **num_msgs** – Number of messages to transfer.
- **addr** – Address of the I2C target device.

Return values

- **0** – If successful.
- **-EIO** – General input / output error.

static inline int **i2c_transfer_cb**(const struct *device* *dev, struct *i2c_msg* *msgs, uint8_t num_msgs, uint16_t addr, *i2c_callback_t* cb, void *userdata)

Perform data transfer to another I2C device in controller mode.

This routine provides a generic interface to perform data transfer to another I2C device asynchronously with a callback completion.

See also

i2c_transfer()

Function properties (list may not be complete)

isr-ok

Parameters

- **dev** – Pointer to the device structure for an I2C controller driver configured in controller mode.
- **msgs** – Array of messages to transfer, must live until callback completes.
- **num_msgs** – Number of messages to transfer.
- **addr** – Address of the I2C target device.
- **cb** – Function pointer for completion callback.
- **userdata** – Userdata passed to callback.

Return values

- `0` – If successful.
- `-EIO` – General input / output error.
- `-ENOSYS` – If transfer async is not implemented
- `-EWOULDBLOCK` – If the device is temporarily busy doing another transfer

static inline int **i2c_transfer_cb_dt**(const struct *i2c_dt_spec* *spec, struct *i2c_msg* *msgs, uint8_t num_msgs, *i2c_callback_t* cb, void *userdata)

Perform data transfer to another I2C device in master mode asynchronously.

This is equivalent to:

```
i2c_transfer_cb(spec->bus, msgs, num_msgs, spec->addr, cb, userdata);
```

Parameters

- `spec` – I2C specification from devicetree.
- `msgs` – Array of messages to transfer.
- `num_msgs` – Number of messages to transfer.
- `cb` – Function pointer for completion callback.
- `userdata` – Userdata passed to callback.

Returns

a value from *i2c_transfer_cb()*

static inline int **i2c_write_read_cb**(const struct *device* *dev, struct *i2c_msg* *msgs, uint8_t num_msgs, uint16_t addr, const void *write_buf, size_t num_write, void *read_buf, size_t num_read, *i2c_callback_t* cb, void *userdata)

Write then read data from an I2C device asynchronously.

This supports the common operation “this is what I want”, “now give

it to me” transaction pair through a combined write-then-read bus transaction but using i2c_transfer_cb. This helper function expects caller to pass a message pointer with 2 and only 2 size.

Parameters

- `dev` – Pointer to the device structure for an I2C controller driver configured in master mode.
- `msgs` – Array of messages to transfer.
- `num_msgs` – Number of messages to transfer.
- `addr` – Address of the I2C device
- `write_buf` – Pointer to the data to be written
- `num_write` – Number of bytes to write
- `read_buf` – Pointer to storage for read data
- `num_read` – Number of bytes to read
- `cb` – Function pointer for completion callback.
- `userdata` – Userdata passed to callback.

Return values

- `0` – if successful
- `negative` – on error.

static inline int **i2c_write_read_cb_dt**(const struct *i2c_dt_spec* *spec, struct *i2c_msg* *msgs, uint8_t num_msgs, const void *write_buf, size_t num_write, void *read_buf, size_t num_read, *i2c_callback_t* cb, void *userdata)

Write then read data from an I2C device asynchronously.

This is equivalent to:

```
i2c_write_read_cb(spec->bus, msgs, num_msgs,
                  spec->addr, write_buf,
                  num_write, read_buf, num_read);
```

Parameters

- `spec` – I2C specification from devicetree.
- `msgs` – Array of messages to transfer.
- `num_msgs` – Number of messages to transfer.
- `write_buf` – Pointer to the data to be written
- `num_write` – Number of bytes to write
- `read_buf` – Pointer to storage for read data
- `num_read` – Number of bytes to read
- `cb` – Function pointer for completion callback.
- `userdata` – Userdata passed to callback.

Returns
a value from *i2c_write_read_cb()*

static inline int **i2c_transfer_signal**(const struct *device* *dev, struct *i2c_msg* *msgs, uint8_t num_msgs, uint16_t addr, struct *k_poll_signal* *sig)

Perform data transfer to another I2C device in controller mode.

This routine provides a generic interface to perform data transfer to another I2C device asynchronously with a *k_poll_signal* completion.

See also

i2c_transfer_cb()

Function properties (list may not be complete)
isr-ok

Parameters

- `dev` – Pointer to the device structure for an I2C controller driver configured in controller mode.
- `msgs` – Array of messages to transfer, must live until callback completes.
- `num_msgs` – Number of messages to transfer.

- **addr** – Address of the I2C target device.
- **sig** – Signal to notify of transfer completion.

Return values

- **0** – If successful.
- **-EIO** – General input / output error.
- **-ENOSYS** – If transfer async is not implemented
- **-EWOULDBLOCK** – If the device is temporarily busy doing another transfer

static inline void **i2c_iodev_submit**(struct *rtio_iodev_sqe* *iodev_sqe)

Submit request(s) to an I2C device with RTIO.

Parameters

- **iodev_sqe** – Prepared submissions queue entry connected to an iodev defined by I2C_DT_IODEV_DEFINE.

struct *rtio_sqe* ***i2c_rtio_copy**(struct *rtio* *r, struct *rtio_iodev* *iodev, const struct *i2c_msg* *msgs, uint8_t num_msgs)

Copy the i2c_msgs into a set of RTIO requests.

Parameters

- **r** – RTIO context
- **iodev** – RTIO IODev to target for the submissions
- **msgs** – Array of messages
- **num_msgs** – Number of i2c msgs in array

Return values

- **sqe** – Last submission in the queue added
- **NULL** – Not enough memory in the context to copy the requests

static inline int **i2c_transfer_dt**(const struct *i2c_dt_spec* *spec, struct *i2c_msg* *msgs, uint8_t num_msgs)

Perform data transfer to another I2C device in controller mode.

This is equivalent to:

```
i2c_transfer(spec->bus, msgs, num_msgs, spec->addr);
```

Parameters

- **spec** – I2C specification from devicetree.
- **msgs** – Array of messages to transfer.
- **num_msgs** – Number of messages to transfer.

Returns

a value from *i2c_transfer()*

int **i2c_recover_bus**(const struct *device* *dev)

Recover the I2C bus.

Attempt to recover the I2C bus.

Parameters

- **dev** – Pointer to the device structure for an I2C controller driver configured in controller mode.

Return values

- `0` – If successful
- `-EBUSY` – If bus is not clear after recovery attempt.
- `-EIO` – General input / output error.
- `-ENOSYS` – If bus recovery is not implemented

static inline int `i2c_target_register`(const struct *device* *dev, struct *i2c_target_config* *cfg)

Registers the provided config as Target device of a controller.

Enable I2C target mode for the 'dev' I2C bus driver using the provided 'config' struct containing the functions and parameters to send bus events. The I2C target will be registered at the address provided as 'address' struct member. Addressing mode - 7 or 10 bit - depends on the 'flags' struct member. Any I2C bus events related to the target mode will be passed onto I2C target device driver via a set of callback functions provided in the 'callbacks' struct member.

Most of the existing hardware allows simultaneous support for controller and target mode. This is however not guaranteed.

Parameters

- `dev` – Pointer to the device structure for an I2C controller driver configured in target mode.
- `cfg` – Config struct with functions and parameters used by the I2C driver to send bus events

Return values

- `0` – Is successful
- `-EINVAL` – If parameters are invalid
- `-EIO` – General input / output error.
- `-ENOSYS` – If target mode is not implemented

static inline int `i2c_target_unregister`(const struct *device* *dev, struct *i2c_target_config* *cfg)

Unregisters the provided config as Target device.

This routine disables I2C target mode for the 'dev' I2C bus driver using the provided 'config' struct containing the functions and parameters to send bus events.

Parameters

- `dev` – Pointer to the device structure for an I2C controller driver configured in target mode.
- `cfg` – Config struct with functions and parameters used by the I2C driver to send bus events

Return values

- `0` – Is successful
- `-EINVAL` – If parameters are invalid
- `-ENOSYS` – If target mode is not implemented

int `i2c_target_driver_register`(const struct *device* *dev)

Instructs the I2C Target device to register itself to the I2C Controller.

This routine instructs the I2C Target device to register itself to the I2C Controller via its parent controller's *i2c_target_register()* API.

Parameters

- `dev` – Pointer to the device structure for the I2C target device (not itself an I2C controller).

Return values

- `0` – Is successful
- `-EINVAL` – If parameters are invalid
- `-EIO` – General input / output error.

int `i2c_target_driver_unregister`(const struct *device* *dev)

Instructs the I2C Target device to unregister itself from the I2C Controller.

This routine instructs the I2C Target device to unregister itself from the I2C Controller via its parent controller's *i2c_target_register()* API.

Parameters

- `dev` – Pointer to the device structure for the I2C target device (not itself an I2C controller).

Return values

- `0` – Is successful
- `-EINVAL` – If parameters are invalid

static inline int `i2c_write`(const struct *device* *dev, const uint8_t *buf, uint32_t num_bytes, uint16_t addr)

Write a set amount of data to an I2C device.

This routine writes a set amount of data synchronously.

Parameters

- `dev` – Pointer to the device structure for an I2C controller driver configured in controller mode.
- `buf` – Memory pool from which the data is transferred.
- `num_bytes` – Number of bytes to write.
- `addr` – Address to the target I2C device for writing.

Return values

- `0` – If successful.
- `-EIO` – General input / output error.

static inline int `i2c_write_dt`(const struct *i2c_dt_spec* *spec, const uint8_t *buf, uint32_t num_bytes)

Write a set amount of data to an I2C device.

This is equivalent to:

```
i2c_write(spec->bus, buf, num_bytes, spec->addr);
```

Parameters

- `spec` – I2C specification from devicetree.
- `buf` – Memory pool from which the data is transferred.
- `num_bytes` – Number of bytes to write.

Returns

a value from *i2c_write()*

static inline int **i2c_read**(const struct *device* *dev, uint8_t *buf, uint32_t num_bytes, uint16_t addr)

Read a set amount of data from an I2C device.

This routine reads a set amount of data synchronously.

Parameters

- **dev** – Pointer to the device structure for an I2C controller driver configured in controller mode.
- **buf** – Memory pool that stores the retrieved data.
- **num_bytes** – Number of bytes to read.
- **addr** – Address of the I2C device being read.

Return values

- **0** – If successful.
- **-EIO** – General input / output error.

static inline int **i2c_read_dt**(const struct *i2c_dt_spec* *spec, uint8_t *buf, uint32_t num_bytes)

Read a set amount of data from an I2C device.

This is equivalent to:

```
i2c_read(spec->bus, buf, num_bytes, spec->addr);
```

Parameters

- **spec** – I2C specification from devicetree.
- **buf** – Memory pool that stores the retrieved data.
- **num_bytes** – Number of bytes to read.

Returns
a value from *i2c_read()*

static inline int **i2c_write_read**(const struct *device* *dev, uint16_t addr, const void *write_buf, size_t num_write, void *read_buf, size_t num_read)

Write then read data from an I2C device.

This supports the common operation “this is what I want”, “now give

it to me” transaction pair through a combined write-then-read bus transaction.

Parameters

- **dev** – Pointer to the device structure for an I2C controller driver configured in controller mode.
- **addr** – Address of the I2C device
- **write_buf** – Pointer to the data to be written
- **num_write** – Number of bytes to write
- **read_buf** – Pointer to storage for read data
- **num_read** – Number of bytes to read

Return values

- `0` – if successful
- `negative` – on error.

static inline int **i2c_write_read_dt**(const struct *i2c_dt_spec* *spec, const void *write_buf, size_t num_write, void *read_buf, size_t num_read)

Write then read data from an I2C device.

This is equivalent to:

```
i2c_write_read(spec->bus, spec->addr,
               write_buf, num_write,
               read_buf, num_read);
```

Parameters

- `spec` – I2C specification from devicetree.
- `write_buf` – Pointer to the data to be written
- `num_write` – Number of bytes to write
- `read_buf` – Pointer to storage for read data
- `num_read` – Number of bytes to read

Returns
a value from *i2c_write_read()*

static inline int **i2c_burst_read**(const struct *device* *dev, uint16_t dev_addr, uint8_t start_addr, uint8_t *buf, uint32_t num_bytes)

Read multiple bytes from an internal address of an I2C device.

This routine reads multiple bytes from an internal address of an I2C device synchronously.

Instances of this may be replaced by *i2c_write_read()*.

Parameters

- `dev` – Pointer to the device structure for an I2C controller driver configured in controller mode.
- `dev_addr` – Address of the I2C device for reading.
- `start_addr` – Internal address from which the data is being read.
- `buf` – Memory pool that stores the retrieved data.
- `num_bytes` – Number of bytes being read.

Return values

- `0` – If successful.
- `-EIO` – General input / output error.

static inline int **i2c_burst_read_dt**(const struct *i2c_dt_spec* *spec, uint8_t start_addr, uint8_t *buf, uint32_t num_bytes)

Read multiple bytes from an internal address of an I2C device.

This is equivalent to:

```
i2c_burst_read(spec->bus, spec->addr, start_addr, buf, num_bytes);
```

Parameters

- `spec` – I2C specification from devicetree.

- **start_addr** – Internal address from which the data is being read.
- **buf** – Memory pool that stores the retrieved data.
- **num_bytes** – Number of bytes to read.

Returns
a value from *i2c_burst_read()*

static inline int **i2c_burst_write**(const struct *device* *dev, uint16_t dev_addr, uint8_t start_addr, const uint8_t *buf, uint32_t num_bytes)

Write multiple bytes to an internal address of an I2C device.

This routine writes multiple bytes to an internal address of an I2C device synchronously.

Warning

The combined write synthesized by this API may not be supported on all I2C devices. Uses of this API may be made more portable by replacing them with calls to *i2c_write()* passing a buffer containing the combined address and data.

Parameters

- **dev** – Pointer to the device structure for an I2C controller driver configured in controller mode.
- **dev_addr** – Address of the I2C device for writing.
- **start_addr** – Internal address to which the data is being written.
- **buf** – Memory pool from which the data is transferred.
- **num_bytes** – Number of bytes being written.

Return values

- **0** – If successful.
- **-EIO** – General input / output error.

static inline int **i2c_burst_write_dt**(const struct *i2c_dt_spec* *spec, uint8_t start_addr, const uint8_t *buf, uint32_t num_bytes)

Write multiple bytes to an internal address of an I2C device.

This is equivalent to:

```
i2c_burst_write(spec->bus, spec->addr, start_addr, buf, num_bytes);
```

Parameters

- **spec** – I2C specification from devicetree.
- **start_addr** – Internal address to which the data is being written.
- **buf** – Memory pool from which the data is transferred.
- **num_bytes** – Number of bytes being written.

Returns
a value from *i2c_burst_write()*

static inline int **i2c_reg_read_byte**(const struct *device* *dev, uint16_t dev_addr, uint8_t reg_addr, uint8_t *value)

Read internal register of an I2C device.

This routine reads the value of an 8-bit internal register of an I2C device synchronously.

Parameters

- `dev` – Pointer to the device structure for an I2C controller driver configured in controller mode.
- `dev_addr` – Address of the I2C device for reading.
- `reg_addr` – Address of the internal register being read.
- `value` – Memory pool that stores the retrieved register value.

Return values

- `0` – If successful.
- `-EIO` – General input / output error.

static inline int **i2c_reg_read_byte_dt**(const struct *i2c_dt_spec* *spec, uint8_t reg_addr, uint8_t *value)

Read internal register of an I2C device.

This is equivalent to:

```
i2c_reg_read_byte(spec->bus, spec->addr, reg_addr, value);
```

Parameters

- `spec` – I2C specification from devicetree.
- `reg_addr` – Address of the internal register being read.
- `value` – Memory pool that stores the retrieved register value.

Returns

a value from *i2c_reg_read_byte()*

static inline int **i2c_reg_write_byte**(const struct *device* *dev, uint16_t dev_addr, uint8_t reg_addr, uint8_t value)

Write internal register of an I2C device.

This routine writes a value to an 8-bit internal register of an I2C device synchronously.

Note

This function internally combines the register and value into a single bus transaction.

Parameters

- `dev` – Pointer to the device structure for an I2C controller driver configured in controller mode.
- `dev_addr` – Address of the I2C device for writing.
- `reg_addr` – Address of the internal register being written.
- `value` – Value to be written to internal register.

Return values

- `0` – If successful.

- `-EIO` – General input / output error.

static inline int **i2c_reg_write_byte_dt**(const struct *i2c_dt_spec* *spec, uint8_t reg_addr, uint8_t value)

Write internal register of an I2C device.

This is equivalent to:

```
i2c_reg_write_byte(spec->bus, spec->addr, reg_addr, value);
```

Parameters

- `spec` – I2C specification from devicetree.
- `reg_addr` – Address of the internal register being written.
- `value` – Value to be written to internal register.

Returns
a value from *i2c_reg_write_byte()*

static inline int **i2c_reg_update_byte**(const struct *device* *dev, uint8_t dev_addr, uint8_t reg_addr, uint8_t mask, uint8_t value)

Update internal register of an I2C device.

This routine updates the value of a set of bits from an 8-bit internal register of an I2C device synchronously.

Note

If the calculated new register value matches the value that was read this function will not generate a write operation.

Parameters

- `dev` – Pointer to the device structure for an I2C controller driver configured in controller mode.
- `dev_addr` – Address of the I2C device for updating.
- `reg_addr` – Address of the internal register being updated.
- `mask` – Bitmask for updating internal register.
- `value` – Value for updating internal register.

Return values

- `0` – If successful.
- `-EIO` – General input / output error.

static inline int **i2c_reg_update_byte_dt**(const struct *i2c_dt_spec* *spec, uint8_t reg_addr, uint8_t mask, uint8_t value)

Update internal register of an I2C device.

This is equivalent to:

```
i2c_reg_update_byte(spec->bus, spec->addr, reg_addr, mask, value);
```

Parameters

- `spec` – I2C specification from devicetree.

- `reg_addr` – Address of the internal register being updated.
- `mask` – Bitmask for updating internal register.
- `value` – Value for updating internal register.

Returns
a value from *i2c_reg_update_byte()*

Variables

const struct *rtio_iodev_api* `i2c_iodev_api`

struct `i2c_dt_spec`

#include <*i2c.h*> Complete I2C DT information.

Param bus
is the I2C bus

Param addr
is the target address

struct `i2c_msg`

#include <*i2c.h*> One I2C Message.

This defines one I2C message to transact on the I2C bus.

> **Note**
>
> Some of the configurations supported by this API may not be supported by specific SoC I2C hardware implementations, in particular features related to bus transactions intended to read or write data from different buffers within a single transaction. Invocations of *i2c_transfer()* may not indicate an error when an unsupported configuration is encountered. In some cases drivers will generate separate transactions for each message fragment, with or without presence of *I2C_MSG_RESTART* in *flags*.

Public Members

uint8_t *`buf`

Data buffer in bytes.

uint32_t `len`

Length of buffer in bytes.

uint8_t `flags`

Flags for this message.

struct `i2c_target_callbacks`

#include <*i2c.h*> Structure providing callbacks to be implemented for devices that supports the I2C target API.

This structure may be shared by multiple devices that implement the same API at different addresses on the bus.

struct **i2c_target_config**

#include <*i2c.h*> Structure describing a device that supports the I2C target API.

Instances of this are passed to the *i2c_target_register()* and *i2c_target_unregister()* functions to indicate addition and removal of a target device, respective.

Fields other than node must be initialized by the module that implements the device behavior prior to passing the object reference to *i2c_target_register()*.

Public Members

sys_snode_t **node**

Private, do not modify.

uint8_t **flags**

Flags for the target device defined by I2C_TARGET_FLAGS_* constants.

uint16_t **address**

Address for this target device.

const struct *i2c_target_callbacks* ***callbacks**

Callback functions.

struct **i2c_device_state**

#include <*i2c.h*> I2C specific device state which allows for i2c device class specific additions.

7.6.27 Inter-Processor Mailbox (IPM)

Overview

API Reference

Related code samples

IPM on ESP32
Implement inter-processor mailbox (IPM) between ESP32 APP and PRO CPUs.

IPM on NXP LPC
Implement inter-processor mailbox (IPM) on NXP LPC family.

IPM on NXP i.MX
Implement inter-processor mailbox (IPM) on i.MX SoCs containing a Messaging Unit peripheral.

IPM over IVSHMEM
Implement inter-processor mailbox (IPM) over IVSHMEM (Inter-VM shared memory)

IPM with ARM MHU
Implement inter-processor mailbox (IPM) using an MHU (Message Handling Unit)

OpenAMP
Send messages between two cores using OpenAMP.

OpenAMP using resource table
Send messages between two cores using OpenAMP and a resource table.

group `ipm_interface`

IPM Interface.

Since
1.0

Version
1.0.0

Typedefs

typedef void (***ipm_callback_t**)(const struct *device* *ipmdev, void *user_data, uint32_t id, volatile void *data)

Callback API for incoming IPM messages.

These callbacks execute in interrupt context. Therefore, use only interrupt-safe APIS. Registration of callbacks is done via *ipm_register_callback*

Param ipmdev
Driver instance

Param user_data
Pointer to some private data provided at registration time.

Param id
Message type identifier.

Param data
Message data pointer. The correct amount of data to read out must be inferred using the message id/upper level protocol.

typedef int (***ipm_send_t**)(const struct *device* *ipmdev, int wait, uint32_t id, const void *data, int size)

Callback API to send IPM messages.

See ipm_send() for argument definitions.

typedef int (***ipm_max_data_size_get_t**)(const struct *device* *ipmdev)

Callback API to get maximum data size.

See ipm_max_data_size_get() for argument definitions.

typedef uint32_t (***ipm_max_id_val_get_t**)(const struct *device* *ipmdev)

Callback API to get the ID's maximum value.

See ipm_max_id_val_get() for argument definitions.

typedef void (***ipm_register_callback_t**)(const struct *device* *port, *ipm_callback_t* cb, void *user_data)

Callback API upon registration.

See ipm_register_callback() for argument definitions.

typedef int (*`ipm_set_enabled_t`)(const struct *device* *ipmdev, int enable)

Callback API upon enablement of interrupts.

See ipm_set_enabled() for argument definitions.

typedef void (*`ipm_complete_t`)(const struct *device* *ipmdev)

Callback API upon command completion.

See ipm_complete() for argument definitions.

Functions

int `ipm_send`(const struct *device* *ipmdev, int wait, uint32_t id, const void *data, int size)

Try to send a message over the IPM device.

A message is considered consumed once the remote interrupt handler finishes. If there is deferred processing on the remote side, or if outgoing messages must be queued and wait on an event/semaphore, a high-level driver can implement that.

There are constraints on how much data can be sent or the maximum value of id. Use the *ipm_max_data_size_get* and *ipm_max_id_val_get* routines to determine them.

The *size* parameter is used only on the sending side to determine the amount of data to put in the message registers. It is not passed along to the receiving side. The upper-level protocol dictates the amount of data read back.

Parameters

- `ipmdev` – Driver instance
- `wait` – If nonzero, busy-wait for remote to consume the message. The message is considered consumed once the remote interrupt handler finishes. If there is deferred processing on the remote side, or you would like to queue outgoing messages and wait on an event/semaphore, you can implement that in a high-level driver
- `id` – Message identifier. Values are constrained by *ipm_max_data_size_get* since many boards only allow for a subset of bits in a 32-bit register to store the ID.
- `data` – Pointer to the data sent in the message.
- `size` – Size of the data.

Return values

- `-EBUSY` – If the remote hasn't yet read the last data sent.
- `-EMSGSIZE` – If the supplied data size is unsupported by the driver.
- `-EINVAL` – If there was a bad parameter, such as: too-large id value. or the device isn't an outbound IPM channel.
- `0` – On success.

static inline void `ipm_register_callback`(const struct *device* *ipmdev, *ipm_callback_t* cb, void *user_data)

Register a callback function for incoming messages.

Parameters

- `ipmdev` – Driver instance pointer.
- `cb` – Callback function to execute on incoming message interrupts.

- `user_data` – Application-specific data pointer which will be passed to the callback function when executed.

int `ipm_max_data_size_get`(const struct *device* *ipmdev)

Return the maximum number of bytes possible in an outbound message.

IPM implementations vary on the amount of data that can be sent in a single message since the data payload is typically stored in registers.

Parameters

- `ipmdev` – Driver instance pointer.

Returns

Maximum possible size of a message in bytes.

uint32_t `ipm_max_id_val_get`(const struct *device* *ipmdev)

Return the maximum id value possible in an outbound message.

Many IPM implementations store the message's ID in a register with some bits reserved for other uses.

Parameters

- `ipmdev` – Driver instance pointer.

Returns

Maximum possible value of a message ID.

int `ipm_set_enabled`(const struct *device* *ipmdev, int enable)

Enable interrupts and callbacks for inbound channels.

Parameters

- `ipmdev` – Driver instance pointer.
- `enable` – Set to 0 to disable and to nonzero to enable.

Return values

- `0` – On success.
- `-EINVAL` – If it isn't an inbound channel.

void `ipm_complete`(const struct *device* *ipmdev)

Signal asynchronous command completion.

Some IPM backends have an ability to deliver a command asynchronously. The callback will be invoked in interrupt context, but the message (including the provided data pointer) will stay "active" and unacknowledged until later code (presumably in thread mode) calls *ipm_complete()*.

This function is, obviously, a noop on drivers without async support.

Parameters

- `ipmdev` – Driver instance pointer.

struct `ipm_driver_api`

#include <ipm.h>

7.6.28 Keyboard Scan

Overview

The kscan driver (keyboard scan matrix) is used for detecting a key press in a connected matrix keyboard or any device with buttons such as joysticks. Typically, matrix keyboards are implemented using a two-dimensional configuration in order to sense several keys. This allows interfacing to many keys through fewer physical pins. Keyboard matrix drivers read the rows while applying power through the columns one at a time with the purpose of detecting key events. There is no correlation between the physical and electrical layout of keys. For, example, the physical layout may be one array of 16 or fewer keys, which may be electrically connected to a 4 x 4 array. In addition, key values are defined by a keymap provided by the keyboard manufacturer.

Configuration Options

Related configuration options:

- `CONFIG_KSCAN`

API Reference

Related code samples

HT16K33 LED driver with keyscan
Control up to 128 LEDs connected to an HT16K33 LED driver and log keyscan events.

KSCAN
Use the KSCAN API to read key presses and releases on a keyboard matrix.

group `kscan_interface`

KSCAN APIs.

Since
2.1

Version
1.0.0

Typedefs

typedef void (***kscan_callback_t**)(const struct *device* *dev, uint32_t row, uint32_t column, bool pressed)

Keyboard scan callback called when user press/release a key on a matrix keyboard.

Param dev
Pointer to the device structure for the driver instance.

Param row
Describes row change.

Param column
Describes column change.

Param pressed
Describes the kind of key event.

Functions

int `kscan_config`(const struct *device* *dev, *kscan_callback_t* callback)
Configure a Keyboard scan instance.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `callback` – called when keyboard devices reply to a keyboard event such as key pressed/released.

Return values

- `0` – If successful.
- `Negative` – errno code if failure.

int `kscan_enable_callback`(const struct *device* *dev)
Enables callback.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Return values

- `0` – If successful.
- `Negative` – errno code if failure.

int `kscan_disable_callback`(const struct *device* *dev)
Disables callback.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Return values

- `0` – If successful.
- `Negative` – errno code if failure.

7.6.29 Light-Emitting Diode (LED)

Overview

The LED API provides access to Light Emitting Diodes, both in individual and strip form.

Configuration Options

Related configuration options:

- `CONFIG_LED`
- `CONFIG_LED_STRIP`

API Reference

Related code samples

Breathing-blinking LED (BBLED)
Control a BBLED (Breathing-Blinking LED) using Microchip XEC driver.

HT16K33 LED driver with keyscan
Control up to 128 LEDs connected to an HT16K33 LED driver and log keyscan events.

IS31FL3194 RGB LED
Cycle colors on an RGB LED connected to the IS31FL3194 using the LED API.

IS31FL3216A LED
Control up to 16 PWM LEDs connected to an IS31FL3216A driver chip.

IS31FL3733 LED Matrix
Control a matrix of up to 192 LEDs connected to an IS31FL3733 driver chip.

LED PWM
Control PWM LEDs using the LED API.

LP3943 RGBW LED
Control up to 16 RGBW LEDs connected to an LP3943 driver chip.

LP50XX RGB LED
Control up to 12 RGB LEDs connected to an LP50xx driver chip.

LP5562 RGB LED
Control 4 RGB LEDs connected to an LP5562 driver chip.

LP5569 9-channel LED controller
Control 9 LEDs connected to an LP5569 driver chip.

PCA9633 LED
Control 4 LEDs connected to a PCA9633 driver chip.

SX1509B RGB LED
Control an RGB LED connected to an SX1509B driver chip.

LED

group `led_interface`

LED Interface.

Since
1.12

Version
1.0.0

Typedefs

typedef int (*`led_api_blink`)(const struct *device* *dev, uint32_t led, uint32_t delay_on, uint32_t delay_off)

Callback API for blinking an LED.

> **See also**
>
> *led_blink()* for argument descriptions.

typedef int (*`led_api_get_info`)(const struct *device* *dev, uint32_t led, const struct *led_info* **info)

Optional API callback to get LED information.

See also

led_get_info() for argument descriptions.

typedef int (*`led_api_set_brightness`)(const struct *device* *dev, uint32_t led, uint8_t value)

Callback API for setting brightness of an LED.

See also

led_set_brightness() for argument descriptions.

typedef int (*`led_api_set_color`)(const struct *device* *dev, uint32_t led, uint8_t num_colors, const uint8_t *color)

Optional API callback to set the colors of a LED.

See also

led_set_color() for argument descriptions.

typedef int (*`led_api_on`)(const struct *device* *dev, uint32_t led)

Callback API for turning on an LED.

See also

led_on() for argument descriptions.

typedef int (*`led_api_off`)(const struct *device* *dev, uint32_t led)

Callback API for turning off an LED.

See also

led_off() for argument descriptions.

typedef int (*`led_api_write_channels`)(const struct *device* *dev, uint32_t start_channel, uint32_t num_channels, const uint8_t *buf)

Callback API for writing a strip of LED channels.

See also

led_api_write_channels() for arguments descriptions.

Functions

int `led_blink`(const struct *device* *dev, uint32_t led, uint32_t delay_on, uint32_t delay_off)

Blink an LED.

This optional routine starts blinking a LED forever with the given time period.

Parameters

- `dev` – LED device
- `led` – LED number
- `delay_on` – Time period (in milliseconds) an LED should be ON
- `delay_off` – Time period (in milliseconds) an LED should be OFF

Returns

0 on success, negative on error

int `led_get_info`(const struct *device* *dev, uint32_t led, const struct *led_info* **info)

Get LED information.

This optional routine provides information about a LED.

Parameters

- `dev` – LED device
- `led` – LED number
- `info` – Pointer to a pointer filled with LED information

Returns

0 on success, negative on error

int `led_set_brightness`(const struct *device* *dev, uint32_t led, uint8_t value)

Set LED brightness.

This optional routine sets the brightness of a LED to the given value. Calling this function after *led_blink()* won't affect blinking.

LEDs which can only be turned on or off may provide this function. These should simply turn the LED on if `value` is nonzero, and off if `value` is zero.

Parameters

- `dev` – LED device
- `led` – LED number
- `value` – Brightness value to set in percent

Returns

0 on success, negative on error

int `led_write_channels`(const struct *device* *dev, uint32_t start_channel, uint32_t num_channels, const uint8_t *buf)

Write/update a strip of LED channels.

This optional routine writes a strip of LED channels to the given array of levels. Therefore it can be used to configure several LEDs at the same time.

Calling this function after *led_blink()* won't affect blinking.

Parameters

- `dev` – LED device
- `start_channel` – Absolute number (i.e. not relative to a LED) of the first channel to update.
- `num_channels` – The number of channels to write/update.
- `buf` – array of values to configure the channels with. num_channels entries must be provided.

Returns

0 on success, negative on error

int `led_set_channel`(const struct *device* *dev, uint32_t channel, uint8_t value)

Set a single LED channel.

This optional routine sets a single LED channel to the given value.

Calling this function after *led_blink()* won't affect blinking.

Parameters

- `dev` – LED device
- `channel` – Absolute channel number (i.e. not relative to a LED)
- `value` – Value to configure the channel with

Returns

0 on success, negative on error

int `led_set_color`(const struct *device* *dev, uint32_t led, uint8_t num_colors, const uint8_t *color)

Set LED color.

This routine configures all the color channels of a LED with the given color array.

Calling this function after *led_blink()* won't affect blinking.

Parameters

- `dev` – LED device
- `led` – LED number
- `num_colors` – Number of colors in the array.
- `color` – Array of colors. It must be ordered following the color mapping of the LED controller. See the color_mapping member in struct *led_info*.

Returns

0 on success, negative on error

int `led_on`(const struct *device* *dev, uint32_t led)

Turn on an LED.

This routine turns on an LED

Parameters

- `dev` – LED device
- `led` – LED number

Returns
0 on success, negative on error

int `led_off`(const struct *device* *dev, uint32_t led)
Turn off an LED.

This routine turns off an LED

Parameters

- `dev` – LED device
- `led` – LED number

Returns
0 on success, negative on error

struct `led_info`
#include <*led.h*> LED information structure.

This structure gathers useful information about LED controller.

Public Members

const char *`label`
LED label.

uint32_t `index`
Index of the LED on the controller.

uint8_t `num_colors`
Number of colors per LED.

const uint8_t *`color_mapping`
Mapping of the LED colors.

struct `led_driver_api`
#include <*led.h*> LED driver API.

Related code samples

LED strip
Control an LED strip example.

LED Strip

group `led_strip_interface`
LED Strip Interface.

Typedefs

typedef int (*`led_api_update_rgb`)(const struct *device* *dev, struct *led_rgb* *pixels, size_t num_pixels)

Callback API for updating an RGB LED strip.

See also

led_strip_update_rgb() for argument descriptions.

typedef int (*`led_api_update_channels`)(const struct *device* *dev, uint8_t *channels, size_t num_channels)

Callback API for updating channels without an RGB interpretation.

See also

led_strip_update_channels() for argument descriptions.

typedef size_t (*`led_api_length`)(const struct *device* *dev)

Callback API for getting length of an LED strip.

See also

led_strip_length() for argument descriptions.

Functions

static inline int `led_strip_update_rgb`(const struct *device* *dev, struct *led_rgb* *pixels, size_t num_pixels)

Mandatory function to update an LED strip with the given RGB array.

Warning

This routine may overwrite *pixels*.

Parameters

- `dev` – LED strip device.
- `pixels` – Array of pixel data.
- `num_pixels` – Length of pixels array.

Return values

- `0` – on success.
- `-errno` – negative errno code on failure.

static inline int **led_strip_update_channels**(const struct *device* *dev, uint8_t *channels, size_t num_channels)

Optional function to update an LED strip with the given channel array (each channel byte corresponding to an individually addressable color channel or LED.

Channels are updated linearly in strip order.

Warning

This routine may overwrite *channels*.

Parameters

- `dev` – LED strip device.
- `channels` – Array of per-channel data.
- `num_channels` – Length of channels array.

Return values

- `0` – on success.
- `-ENOSYS` – if not implemented.
- `-errno` – negative errno code on other failure.

static inline size_t **led_strip_length**(const struct *device* *dev)

Mandatory function to get chain length (in pixels) of an LED strip device.

Parameters

- `dev` – LED strip device.

Return values

`Length` – of LED strip device.

struct **led_rgb**

#include <*led_strip.h*> Color value for a single RGB LED.

Individual strip drivers may ignore lower-order bits if their resolution in any channel is less than a full byte.

Public Members

uint8_t **r**

Red channel.

uint8_t **g**

Green channel.

uint8_t **b**

Blue channel.

struct **led_strip_driver_api**

#include <*led_strip.h*> LED strip driver API.

This is the mandatory API any LED strip driver needs to expose.

7.6.30 Management Data Input/Output (MDIO)

Overview

MDIO is a bus that is commonly used to communicate with ethernet PHY devices. Many ethernet MAC controllers also provide hardware to communicate over MDIO bus with a peripheral device.

This API is intended to be used primarily by PHY drivers but can also be used by user firmware.

API Reference

group `mdio_interface`

MDIO Interface.

Functions

void `mdio_bus_enable`(const struct *device* *dev)

Enable MDIO bus.

Parameters

- `dev` – **[in]** Pointer to the device structure for the controller

void `mdio_bus_disable`(const struct *device* *dev)

Disable MDIO bus and tri-state drivers.

Parameters

- `dev` – **[in]** Pointer to the device structure for the controller

int `mdio_read`(const struct *device* *dev, uint8_t prtad, uint8_t regad, uint16_t *data)

Read from MDIO Bus.

This routine provides a generic interface to perform a read on the MDIO bus.

Parameters

- `dev` – **[in]** Pointer to the device structure for the controller
- `prtad` – **[in]** Port address
- `regad` – **[in]** Register address
- `data` – Pointer to receive read data

Return values

- `0` – If successful.
- `-EIO` – General input / output error.
- `-ETIMEDOUT` – If transaction timedout on the bus
- `-ENOSYS` – if read is not supported

int `mdio_write`(const struct *device* *dev, uint8_t prtad, uint8_t regad, uint16_t data)

Write to MDIO bus.

This routine provides a generic interface to perform a write on the MDIO bus.

Parameters

- `dev` – **[in]** Pointer to the device structure for the controller
- `prtad` – **[in]** Port address

- `regad` – **[in]** Register address
- `data` – **[in]** Data to write

Return values

- `0` – If successful.
- `-EIO` – General input / output error.
- `-ETIMEDOUT` – If transaction timedout on the bus
- `-ENOSYS` – if write is not supported

int `mdio_read_c45`(const struct *device* *dev, uint8_t prtad, uint8_t devad, uint16_t regad, uint16_t *data)

Read from MDIO Bus using Clause 45 access.

This routine provides an interface to perform a read on the MDIO bus using IEEE 802.3 Clause 45 access.

Parameters

- `dev` – **[in]** Pointer to the device structure for the controller
- `prtad` – **[in]** Port address
- `devad` – **[in]** Device address
- `regad` – **[in]** Register address
- `data` – Pointer to receive read data

Return values

- `0` – If successful.
- `-EIO` – General input / output error.
- `-ETIMEDOUT` – If transaction timedout on the bus
- `-ENOSYS` – if write using Clause 45 access is not supported

int `mdio_write_c45`(const struct *device* *dev, uint8_t prtad, uint8_t devad, uint16_t regad, uint16_t data)

Write to MDIO bus using Clause 45 access.

This routine provides an interface to perform a write on the MDIO bus using IEEE 802.3 Clause 45 access.

Parameters

- `dev` – **[in]** Pointer to the device structure for the controller
- `prtad` – **[in]** Port address
- `devad` – **[in]** Device address
- `regad` – **[in]** Register address
- `data` – **[in]** Data to write

Return values

- `0` – If successful.
- `-EIO` – General input / output error.
- `-ETIMEDOUT` – If transaction timedout on the bus
- `-ENOSYS` – if write using Clause 45 access is not supported

7.6.31 MIPI Display Bus Interface (DBI)

The MIPI DBI driver class implements support for MIPI DBI compliant display controllers.

MIPI DBI defines 3 interface types:

- Type A: Motorola 6800 parallel bus
- Type B: Intel 8080 parallel bus
- Type C: SPI Type serial bit bus with 3 options:
  1. 9 write clocks per byte, final bit is command/data selection bit
  2. Same as above, but 16 write clocks per byte
  3. 8 write clocks per byte. Command/data selected via GPIO pin

Currently, the API only supports Type C controllers, options 1 and 3.

API Reference

group `mipi_dbi_interface`

MIPI-DBI driver APIs.

Since
3.6

Version
0.1.0

Defines

`MIPI_DBI_SPI_CONFIG_DT`(node_id, operation_, delay_)

initialize a MIPI DBI SPI configuration struct from devicetree

This helper allows drivers to initialize a MIPI DBI SPI configuration structure using devicetree.

Parameters

- `node_id` – Devicetree node identifier for the MIPI DBI device whose struct *spi_config* to create an initializer for
- `operation_` – the desired operation field in the struct *spi_config*
- `delay_` – the desired delay field in the struct *spi_config*'s *spi_cs_control*, if there is one

`MIPI_DBI_SPI_CONFIG_DT_INST`(inst, operation_, delay_)

Initialize a MIPI DBI SPI configuration from devicetree instance.

This helper initializes a MIPI DBI SPI configuration from a devicetree instance. It is equivalent to *MIPI_DBI_SPI_CONFIG_DT(DT_DRV_INST(inst))*

Parameters

- `inst` – Instance number to initialize configuration from
- `operation_` – the desired operation field in the struct *spi_config*
- `delay_` – the desired delay field in the struct *spi_config*'s *spi_cs_control*, if there is one

MIPI_DBI_CONFIG_DT(node_id, operation_, delay_)

Initialize a MIPI DBI configuration from devicetree.

This helper allows drivers to initialize a MIPI DBI configuration structure from devicetree. It sets the MIPI DBI mode, as well as configuration fields in the SPI configuration structure

Parameters

- **node_id** – Devicetree node identifier for the MIPI DBI device to initialize
- **operation_** – the desired operation field in the struct *spi_config*
- **delay_** – the desired delay field in the struct *spi_config*'s *spi_cs_control*, if there is one

MIPI_DBI_CONFIG_DT_INST(inst, operation_, delay_)

Initialize a MIPI DBI configuration from device instance.

Equivalent to *MIPI_DBI_CONFIG_DT(DT_DRV_INST(inst), operation_, delay_)*

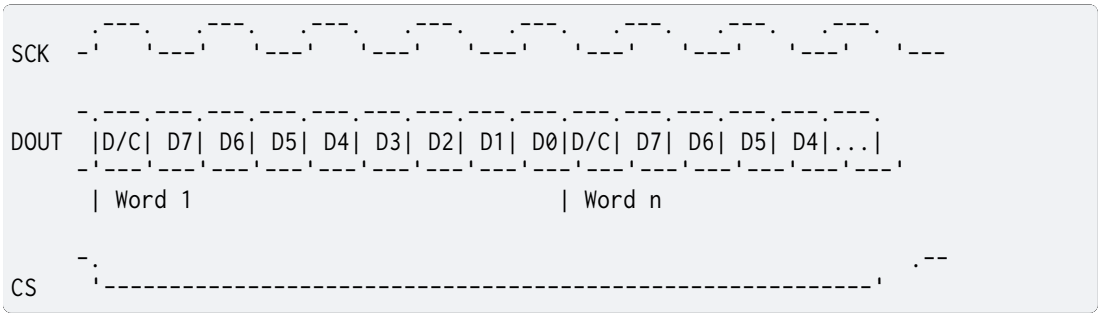
Parameters

- **inst** – Instance of the device to initialize a MIPI DBI configuration for
- **operation_** – the desired operation field in the struct *spi_config*
- **delay_** – the desired delay field in the struct *spi_config*'s *spi_cs_control*, if there is one

MIPI_DBI_MODE_SPI_3WIRE

SPI 3 wire (Type C1).

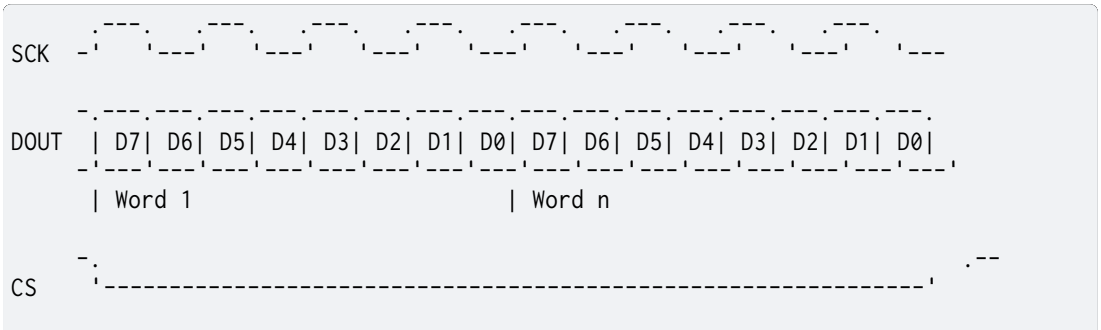
Uses 9 write clocks to send a byte of data. The bit sent on the 9th clock indicates whether the byte is a command or data byte



MIPI_DBI_MODE_SPI_4WIRE

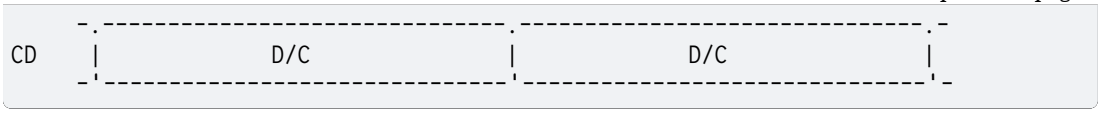
SPI 4 wire (Type C3).

Uses 8 write clocks to send a byte of data. an additional C/D pin will be use to indicate whether the byte is a command or data byte

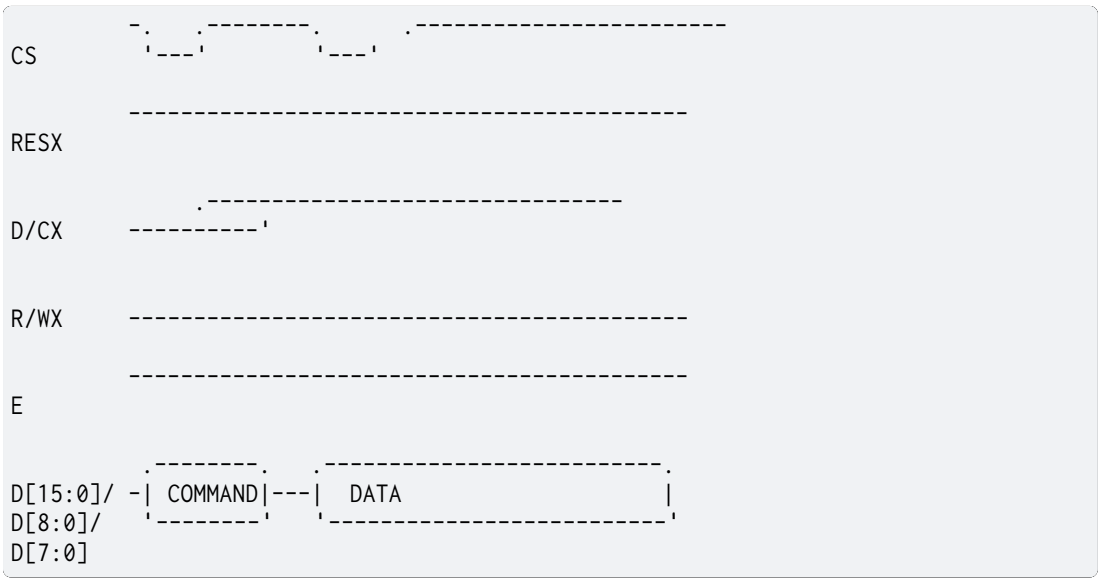


(continues on next page)

(continued from previous page)



MIPI_DBI_MODE_6800_BUS_16_BIT
: Parallel Bus protocol for MIPI DBI Type A based on Motorola 6800 bus.

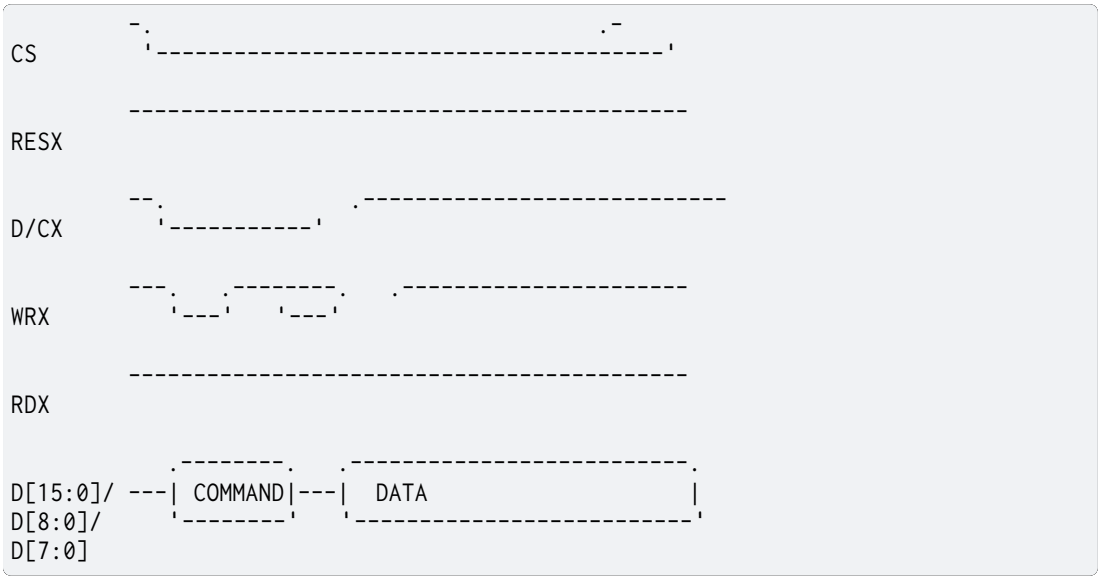


Please refer to the MIPI DBI specification for a detailed cycle diagram.

MIPI_DBI_MODE_6800_BUS_9_BIT

MIPI_DBI_MODE_6800_BUS_8_BIT

MIPI_DBI_MODE_8080_BUS_16_BIT
: Parallel Bus protocol for MIPI DBI Type B based on Intel 8080 bus.



Please refer to the MIPI DBI specification for a detailed cycle diagram.

`MIPI_DBI_MODE_8080_BUS_9_BIT`

`MIPI_DBI_MODE_8080_BUS_8_BIT`

Functions

static inline int **mipi_dbi_command_write**(const struct *device* *dev, const struct *mipi_dbi_config* *config, uint8_t cmd, const uint8_t *data, size_t len)

Write a command to the display controller.

Writes a command, along with an optional data buffer to the display. If data buffer and buffer length are NULL and 0 respectively, then only a command will be sent. Note that if the SPI configuration passed to this function locks the SPI bus, it is the caller's responsibility to release it with *mipi_dbi_release()*

Parameters

- **dev** – mipi dbi controller
- **config** – MIPI DBI configuration
- **cmd** – command to write to display controller
- **data** – optional data buffer to write after command
- **len** – size of data buffer in bytes. Set to 0 to skip sending data.

Return values

- **0** – command write succeeded
- **-EIO** – I/O error
- **-ETIMEDOUT** – transfer timed out
- **-EBUSY** – controller is busy
- **-ENOSYS** – not implemented

static inline int **mipi_dbi_command_read**(const struct *device* *dev, const struct *mipi_dbi_config* *config, uint8_t *cmds, size_t num_cmd, uint8_t *response, size_t len)

Read a command response from the display controller.

Reads a command response from the display controller.

Parameters

- **dev** – mipi dbi controller
- **config** – MIPI DBI configuration
- **cmds** – array of one byte commands to send to display controller
- **num_cmd** – number of commands to write to display controller
- **response** – response buffer, filled with display controller response
- **len** – size of response buffer in bytes.

Return values

- **0** – command read succeeded
- **-EIO** – I/O error

- `-ETIMEDOUT` – transfer timed out
- `-EBUSY` – controller is busy
- `-ENOSYS` – not implemented

static inline int **mipi_dbi_write_display**(const struct *device* *dev, const struct *mipi_dbi_config* *config, const uint8_t *framebuf, struct *display_buffer_descriptor* *desc, enum *display_pixel_format* pixfmt)

Write a display buffer to the display controller.

Writes a display buffer to the controller. If the controller requires a "Write memory" command before writing display data, this should be sent with *mipi_dbi_command_write*

Parameters

- `dev` – mipi dbi controller
- `config` – MIPI DBI configuration
- `framebuf` – framebuffer to write to display
- `desc` – descriptor of framebuffer to write. Note that the pitch must be equal to width. "buf_size" field determines how many bytes will be written.
- `pixfmt` – pixel format of framebuffer data

Return values

- `0` – buffer write succeeded.
- `-EIO` – I/O error
- `-ETIMEDOUT` – transfer timed out
- `-EBUSY` – controller is busy
- `-ENOSYS` – not implemented

static inline int **mipi_dbi_reset**(const struct *device* *dev, uint32_t delay)

Resets attached display controller.

Resets the attached display controller.

Parameters

- `dev` – mipi dbi controller
- `delay` – duration to set reset signal for, in milliseconds

Return values

- `0` – reset succeeded
- `-EIO` – I/O error
- `-ENOSYS` – not implemented
- `-ENOTSUP` – not supported

static inline int **mipi_dbi_release**(const struct *device* *dev, const struct *mipi_dbi_config* *config)

Releases a locked MIPI DBI device.

Releases a lock on a MIPI DBI device and/or the device's CS line if and only if the given config parameter was the last one to be used in any of the above functions, and if it has the SPI_LOCK_ON bit set and/or the SPI_HOLD_ON_CS bit set into its operation bits

field. This lock functions exactly like the SPI lock, and can be used if the caller needs to keep CS asserted for multiple transactions, or the MIPI DBI device locked.

Parameters

- `dev` – mipi dbi controller
- `config` – MIPI DBI configuration

Return values

- `0` – reset succeeded
- `-EIO` – I/O error
- `-ENOSYS` – not implemented
- `-ENOTSUP` – not supported

struct `mipi_dbi_config`

#include <*mipi_dbi.h*> MIPI DBI controller configuration.

Configuration for MIPI DBI controller write

Public Members

uint8_t `mode`

MIPI DBI mode (SPI 3 wire or 4 wire)

struct *spi_config* `config`

SPI configuration.

struct `mipi_dbi_driver_api`

#include <*mipi_dbi.h*> MIPI-DBI host driver API.

7.6.32 MIPI Display Serial Interface (DSI)

API Reference

group `mipi_dsi_interface`

MIPI-DSI driver APIs.

Since
3.1

Version
0.1.0

MIPI-DSI Device mode flags.

`MIPI_DSI_MODE_VIDEO`

Video mode.

`MIPI_DSI_MODE_VIDEO_BURST`
: Video burst mode.

`MIPI_DSI_MODE_VIDEO_SYNC_PULSE`
: Video pulse mode.

`MIPI_DSI_MODE_VIDEO_AUTO_VERT`
: Enable auto vertical count mode.

`MIPI_DSI_MODE_VIDEO_HSE`
: Enable hsync-end packets in vsync-pulse and v-porch area.

`MIPI_DSI_MODE_VIDEO_HFP`
: Disable hfront-porch area.

`MIPI_DSI_MODE_VIDEO_HBP`
: Disable hback-porch area.

`MIPI_DSI_MODE_VIDEO_HSA`
: Disable hsync-active area.

`MIPI_DSI_MODE_VSYNC_FLUSH`
: Flush display FIFO on vsync pulse.

`MIPI_DSI_MODE_EOT_PACKET`
: Disable EoT packets in HS mode.

`MIPI_DSI_CLOCK_NON_CONTINUOUS`
: Device supports non-continuous clock behavior (DSI spec 5.6.1)

`MIPI_DSI_MODE_LPM`
: Transmit data in low power.

MIPI-DSI Pixel formats.

`MIPI_DSI_PIXFMT_RGB888`
: RGB888 (24bpp).

`MIPI_DSI_PIXFMT_RGB666`
: RGB666 (24bpp).

`MIPI_DSI_PIXFMT_RGB666_PACKED`
: Packed RGB666 (18bpp).

`MIPI_DSI_PIXFMT_RGB565`
: RGB565 (16bpp).

Defines

MIPI_DSI_MSG_USE_LPM

Functions

static inline int **mipi_dsi_attach**(const struct *device* *dev, uint8_t channel, const struct *mipi_dsi_device* *mdev)

Attach a new device to the MIPI-DSI bus.

Parameters

- `dev` – MIPI-DSI host device.
- `channel` – Device channel (VID).
- `mdev` – MIPI-DSI device description.

Returns
0 on success, negative on error

static inline ssize_t **mipi_dsi_transfer**(const struct *device* *dev, uint8_t channel, struct *mipi_dsi_msg* *msg)

Transfer data to/from a device attached to the MIPI-DSI bus.

Parameters

- `dev` – MIPI-DSI device.
- `channel` – Device channel (VID).
- `msg` – Message.

Returns
Size of the transferred data on success, negative on error.

ssize_t **mipi_dsi_generic_read**(const struct *device* *dev, uint8_t channel, const void *params, size_t nparams, void *buf, size_t len)

MIPI-DSI generic read.

Parameters

- `dev` – MIPI-DSI host device.
- `channel` – Device channel (VID).
- `params` – Buffer containing request parameters.
- `nparams` – Number of parameters.
- `buf` – Buffer where read data will be stored.
- `len` – Length of the reception buffer.

Returns
Size of the read data on success, negative on error.

ssize_t **mipi_dsi_generic_write**(const struct *device* *dev, uint8_t channel, const void *buf, size_t len)

MIPI-DSI generic write.

Parameters

- `dev` – MIPI-DSI host device.
- `channel` – Device channel (VID).

- `buf` – Transmission buffer.
- `len` – Length of the transmission buffer

Returns
Size of the written data on success, negative on error.

ssize_t **mipi_dsi_dcs_read**(const struct *device* *dev, uint8_t channel, uint8_t cmd, void *buf, size_t len)

MIPI-DSI DCS read.

Parameters

- `dev` – MIPI-DSI host device.
- `channel` – Device channel (VID).
- `cmd` – DCS command.
- `buf` – Buffer where read data will be stored.
- `len` – Length of the reception buffer.

Returns
Size of the read data on success, negative on error.

ssize_t **mipi_dsi_dcs_write**(const struct *device* *dev, uint8_t channel, uint8_t cmd, const void *buf, size_t len)

MIPI-DSI DCS write.

Parameters

- `dev` – MIPI-DSI host device.
- `channel` – Device channel (VID).
- `cmd` – DCS command.
- `buf` – Transmission buffer.
- `len` – Length of the transmission buffer

Returns
Size of the written data on success, negative on error.

static inline int **mipi_dsi_detach**(const struct *device* *dev, uint8_t channel, const struct *mipi_dsi_device* *mdev)

Detach a device from the MIPI-DSI bus.

Parameters

- `dev` – MIPI-DSI host device.
- `channel` – Device channel (VID).
- `mdev` – MIPI-DSI device description.

Returns
0 on success, negative on error

struct **mipi_dsi_timings**

#include <mipi_dsi.h> MIPI-DSI display timings.

Public Members

uint32_t **hactive**
Horizontal active video.

uint32_t **hfp**
Horizontal front porch.

uint32_t **hbp**
Horizontal back porch.

uint32_t **hsync**
Horizontal sync length.

uint32_t **vactive**
Vertical active video.

uint32_t **vfp**
Vertical front porch.

uint32_t **vbp**
Vertical back porch.

uint32_t **vsync**
Vertical sync length.

struct **mipi_dsi_device**
#include <*mipi_dsi.h*> MIPI-DSI device.

Public Members

uint8_t **data_lanes**
Number of data lanes.

struct *mipi_dsi_timings* **timings**
Display timings.

uint32_t **pixfmt**
Pixel format.

uint32_t **mode_flags**
Mode flags.

struct **mipi_dsi_msg**
#include <*mipi_dsi.h*> MIPI-DSI read/write message.

Public Members

uint8_t **type**
: Payload data type.

uint16_t **flags**
: Flags controlling message transmission.

uint8_t **cmd**
: Command (only for DCS)

size_t **tx_len**
: Transmission buffer length.

const void ***tx_buf**
: Transmission buffer.

size_t **rx_len**
: Reception buffer length.

void ***rx_buf**
: Reception buffer.

struct **mipi_dsi_driver_api**
: *#include <mipi_dsi.h>* MIPI-DSI host driver API.

7.6.33 Multi-bit SPI Bus

The MSPI (multi-bit SPI) is provided as a generic API to accommodate advanced SPI peripherals and devices that typically require command, address and data phases, and multiple signal lines during these phases. While the API supports advanced features such as *XIP* and scrambling, it is also compatible with generic SPI.

- *MSPI Controller API*
  - *Transceive*
  - *Device Tree*
  - *Multi Peripheral*
- *Configuration Options*
- *API Reference*

MSPI Controller API

Zephyr's MSPI controller API may be used when a multi-bit SPI controller is present. E.g. Ambiq MSPI, QSPI, OSPI, Flexspi, etc. The API supports single to hex SDR/DDR IO with variable latency and advanced features such as *XIP* and scrambling. Applicable devices include but not limited to high-speed, high density flash/psram memory devices, displays and sensors.

The MSPI interface contains controller drivers that are SoC platform specific and implement the MSPI APIs, and device drivers that reference these APIs. The relationship between the controller and device drivers is many-to-many to allow for easy switching between platforms.

Here is a list of generic steps for initializing the MSPI controller and the MSPI bus inside the device driver initialization function:

1. Initialize the data structure of the MSPI controller driver instance. The usual device defining macros such as *DEVICE_DT_INST_DEFINE* can be used, and the initialization function, config and data provided as a parameter to the macro.
2. Initialize the hardware, including but not limited to:
   - Check `mspi_cfg` against hardware's own capabilities to prevent incorrect usages.
   - Setup default pinmux.
   - Setup the clock for the controller.
   - Power on the hardware.
   - Configure the hardware using `mspi_cfg` and possibly more platform specific settings.
   - Usually, the `mspi_cfg` is filled from device tree and contains static, boot time parameters. However, if needed, one can use `mspi_config()` to re-initialize the hardware with new parameters during runtime.
   - Release any lock if applicable.
3. Perform device driver initialization. As usually, *DEVICE_DT_INST_DEFINE* can be used. Inside device driver initialization function, perform the following required steps.
   1. Call `mspi_dev_config()` with device specific hardware settings obtained from device datasheets.
      - The `mspi_dev_cfg` should be filled by device tree and helper macro `MSPI_DEVICE_CONFIG_DT` can be used.
      - The controller driver should then validate the members of `mspi_dev_cfg` to prevent incorrect usage.
      - The controller driver should implement a mutex to protect from accidental access.
      - The controller driver may also switch between different devices based on `mspi_dev_id`.
   2. Call API for additional setups if supported by hardware
      - `mspi_xip_config()` for *XIP* feature
      - `mspi_scramble_config()` for scrambling feature
      - `mspi_timing_config()` for platform specific timing setup.
   3. Register any callback with `mspi_register_callback()` if needed.
   4. Release the controller mutex lock.

Transceive The transceive request is of type `mspi_xfer` which allows dynamic change to the transfer related settings once the mode of operation is determined and configured by `mspi_dev_config()`.

The API also supports bulk transfers with different starting addresses and sizes with `mspi_xfer_packet`. However, it is up to the controller implementation whether to support scatter IO and callback management. The controller can determine which user callback to trigger based on *mspi_bus_event_cb_mask* upon completion of each async/sync transfer if the callback had been registered using `mspi_register_callback()`. Or not to trigger any callback at all with

MSPI_BUS_NO_CB even if the callbacks are already registered. In which case that a controller supports hardware command queue, user could take full advantage of the hardware performance if scatter IO and callback management are supported by the driver implementation.

Device Tree Here is an example for defining an MSPI controller in device tree: The mspi controller's bindings should reference mspi-controller.yaml as one of the base.

```
mspi0: mspi@400 {
        status = "okay";
        compatible = "zephyr,mspi-emul-controller";

        reg = < 0x400 0x4 >;
        #address-cells = < 0x1 >;
        #size-cells = < 0x0 >;

        clock-frequency = < 0x17d7840 >;
        op-mode = "MSPI_CONTROLLER";
        duplex = "MSPI_HALF_DUPLEX";
        ce-gpios = < &gpio0 0x5 0x1 >, < &gpio0 0x12 0x1 >;
        dqs-support;

        pinctrl-0 = < &pinmux-mspi0 >;
        pinctrl-names = "default";
};
```

Here is an example for defining an MSPI device in device tree: The mspi device's bindings should reference mspi-device.yaml as one of the base.

```
&mspi0 {

        mspi_dev0: mspi_dev0@0 {
                status = "okay";
                compatible = "zephyr,mspi-emul-device";

                reg = < 0x0 >;
                size = < 0x10000 >;

                mspi-max-frequency = < 0x2dc6c00 >;
                mspi-io-mode = "MSPI_IO_MODE_QUAD";
                mspi-data-rate = "MSPI_DATA_RATE_SINGLE";
                mspi-hardware-ce-num = < 0x0 >;
                read-instruction = < 0xb >;
                write-instruction = < 0x2 >;
                instruction-length = "INSTR_1_BYTE";
                address-length = "ADDR_4_BYTE";
                rx-dummy = < 0x8 >;
                tx-dummy = < 0x0 >;
                xip-config = < 0x0 0x0 0x0 0x0 >;
                ce-break-config = < 0x0 0x0 >;
        };

};
```

User should specify target operating parameters in the DTS such as `mspi-max-frequency`, `mspi-io-mode` and `mspi-data-rate` even though they may subject to change during runtime. It should represent the typical configuration of the device during normal operations.

Multi Peripheral With `mspi_dev_id` defined as collection of the device index and CE GPIO from device tree, the API supports multiple devices on the same controller instance. The controller driver implementation may or may not support device switching, which can be performed either

by software or by hardware. If the switching is handled by software, it should be performed in `mspi_dev_config()` call.

The device driver should record the current operating conditions of the device to support software controlled device switching by saving and updating `mspi_dev_cfg` and other relevant mspi struct or private data structures. In particular, `mspi_dev_id` which contains the identity of the device needs to be used for every API call.

Configuration Options

Related configuration options:

- `CONFIG_MSPI`
- `CONFIG_MSPI_ASYNC`
- `CONFIG_MSPI_PERIPHERAL`
- `CONFIG_MSPI_XIP`
- `CONFIG_MSPI_SCRAMBLE`
- `CONFIG_MSPI_TIMING`
- `CONFIG_MSPI_INIT_PRIORITY`
- `CONFIG_MSPI_COMPLETION_TIMEOUT_TOLERANCE`

API Reference

Related code samples

MSPI asynchronous transfer
Use the MSPI API to interact with MSPI memory device asynchronously.

group `mspi_interface`

MSPI Driver APIs.

Typedefs

typedef int (***mspi_api_config**)(const struct mspi_dt_spec *spec)

MSPI driver API definition and system call entry points.

typedef int (***mspi_api_dev_config**)(const struct *device* *controller, const struct mspi_dev_id *dev_id, const enum *mspi_dev_cfg_mask* param_mask, const struct mspi_dev_cfg *cfg)

typedef int (***mspi_api_get_channel_status**)(const struct *device* *controller, uint8_t ch)

typedef int (***mspi_api_transceive**)(const struct *device* *controller, const struct mspi_dev_id *dev_id, const struct mspi_xfer *req)

typedef int (*mspi_api_register_callback)(const struct *device* *controller, const struct mspi_dev_id *dev_id, const enum *mspi_bus_event* evt_type, mspi_callback_handler_t cb, struct mspi_callback_context *ctx)

typedef int (*mspi_api_xip_config)(const struct *device* *controller, const struct mspi_dev_id *dev_id, const struct mspi_xip_cfg *xip_cfg)

typedef int (*mspi_api_scramble_config)(const struct *device* *controller, const struct mspi_dev_id *dev_id, const struct mspi_scramble_cfg *scramble_cfg)

typedef int (*mspi_api_timing_config)(const struct *device* *controller, const struct mspi_dev_id *dev_id, const uint32_t param_mask, void *timing_cfg)

Enums

enum mspi_op_mode

MSPI operational mode.

Values:

enumerator MSPI_OP_MODE_CONTROLLER = 0

enumerator MSPI_OP_MODE_PERIPHERAL = 1

enum mspi_duplex

MSPI duplex mode.

Values:

enumerator MSPI_HALF_DUPLEX = 0

enumerator MSPI_FULL_DUPLEX = 1

enum mspi_io_mode

MSPI I/O mode capabilities Postfix like 1_4_4 stands for the number of lines used for command, address and data phases.

Mode with no postfix has the same number of lines for all phases.

Values:

enumerator MSPI_IO_MODE_SINGLE = 0

enumerator MSPI_IO_MODE_DUAL = 1

enumerator MSPI_IO_MODE_DUAL_1_1_2 = 2

enumerator MSPI_IO_MODE_DUAL_1_2_2 = 3

enumerator MSPI_IO_MODE_QUAD = 4

enumerator MSPI_IO_MODE_QUAD_1_1_4 = 5

enumerator MSPI_IO_MODE_QUAD_1_4_4 = 6

enumerator MSPI_IO_MODE_OCTAL = 7

enumerator MSPI_IO_MODE_OCTAL_1_1_8 = 8

enumerator MSPI_IO_MODE_OCTAL_1_8_8 = 9

enumerator MSPI_IO_MODE_HEX = 10

enumerator MSPI_IO_MODE_HEX_8_8_16 = 11

enumerator MSPI_IO_MODE_HEX_8_16_16 = 12

enumerator MSPI_IO_MODE_MAX

enum mspi_data_rate

MSPI data rate capabilities SINGLE stands for single data rate for all phases.

DUAL stands for dual data rate for all phases. S_S_D stands for single data rate for command and address phases but dual data rate for data phase. S_D_D stands for single data rate for command phase but dual data rate for address and data phases.

Values:

enumerator MSPI_DATA_RATE_SINGLE = 0

enumerator MSPI_DATA_RATE_S_S_D = 1

enumerator MSPI_DATA_RATE_S_D_D = 2

enumerator MSPI_DATA_RATE_DUAL = 3

enumerator MSPI_DATA_RATE_MAX

enum mspi_cpp_mode

MSPI Polarity & Phase Modes.

Values:

enumerator MSPI_CPP_MODE_0 = 0

enumerator MSPI_CPP_MODE_1 = 1

enumerator MSPI_CPP_MODE_2 = 2

enumerator MSPI_CPP_MODE_3 = 3

enum mspi_endian

MSPI Endian.

Values:

enumerator MSPI_XFER_LITTLE_ENDIAN = 0

enumerator MSPI_XFER_BIG_ENDIAN = 1

enum mspi_ce_polarity

MSPI chip enable polarity.

Values:

enumerator MSPI_CE_ACTIVE_LOW = 0

enumerator MSPI_CE_ACTIVE_HIGH = 1

enum mspi_bus_event

MSPI bus event.

This is a preliminary list of events. I encourage the community to fill it up.

Values:

enumerator MSPI_BUS_RESET = 0

enumerator MSPI_BUS_ERROR = 1

enumerator MSPI_BUS_XFER_COMPLETE = 2

enumerator MSPI_BUS_EVENT_MAX

enum mspi_bus_event_cb_mask

MSPI bus event callback mask This is a preliminary list same as mspi_bus_event.

I encourage the community to fill it up.

Values:

enumerator MSPI_BUS_NO_CB = 0

enumerator MSPI_BUS_RESET_CB = *BIT*(0)

enumerator MSPI_BUS_ERROR_CB = *BIT*(1)

enumerator MSPI_BUS_XFER_COMPLETE_CB = *BIT*(2)

enum mspi_xfer_mode

MSPI transfer modes.

Values:

enumerator MSPI_PIO

enumerator MSPI_DMA

enum mspi_xfer_direction

MSPI transfer directions.

Values:

enumerator MSPI_RX

enumerator MSPI_TX

enum mspi_dev_cfg_mask

MSPI controller device specific configuration mask.

Values:

enumerator MSPI_DEVICE_CONFIG_NONE = 0

enumerator MSPI_DEVICE_CONFIG_CE_NUM = *BIT*(0)

enumerator MSPI_DEVICE_CONFIG_FREQUENCY = *BIT*(1)

enumerator MSPI_DEVICE_CONFIG_IO_MODE = *BIT*(2)

enumerator MSPI_DEVICE_CONFIG_DATA_RATE = *BIT*(3)

enumerator MSPI_DEVICE_CONFIG_CPP = *BIT*(4)

enumerator MSPI_DEVICE_CONFIG_ENDIAN = *BIT*(5)

enumerator MSPI_DEVICE_CONFIG_CE_POL = *BIT*(6)

enumerator MSPI_DEVICE_CONFIG_DQS = *BIT*(7)

enumerator MSPI_DEVICE_CONFIG_RX_DUMMY = *BIT*(8)

enumerator MSPI_DEVICE_CONFIG_TX_DUMMY = *BIT*(9)

enumerator MSPI_DEVICE_CONFIG_READ_CMD = *BIT*(10)

enumerator MSPI_DEVICE_CONFIG_WRITE_CMD = *BIT*(11)

enumerator MSPI_DEVICE_CONFIG_CMD_LEN = *BIT*(12)

enumerator MSPI_DEVICE_CONFIG_ADDR_LEN = *BIT*(13)

enumerator MSPI_DEVICE_CONFIG_MEM_BOUND = *BIT*(14)

enumerator MSPI_DEVICE_CONFIG_BREAK_TIME = *BIT*(15)

enumerator MSPI_DEVICE_CONFIG_ALL = *BIT_MASK*(16)

enum mspi_xip_permit

MSPI XIP access permissions.

Values:

enumerator MSPI_XIP_READ_WRITE = 0

enumerator MSPI_XIP_READ_ONLY = 1

struct mspi_driver_api

#include <mspi.h>

7.6.34 Multi-Channel Inter-Processor Mailbox (MBOX)

Overview

An MBOX device is a peripheral capable of passing signals (and data depending on the peripheral) between CPUs and clusters in the system. Each MBOX instance is providing one or more channels, each one targeting one other CPU cluster (multiple channels can target the same cluster).

API Reference

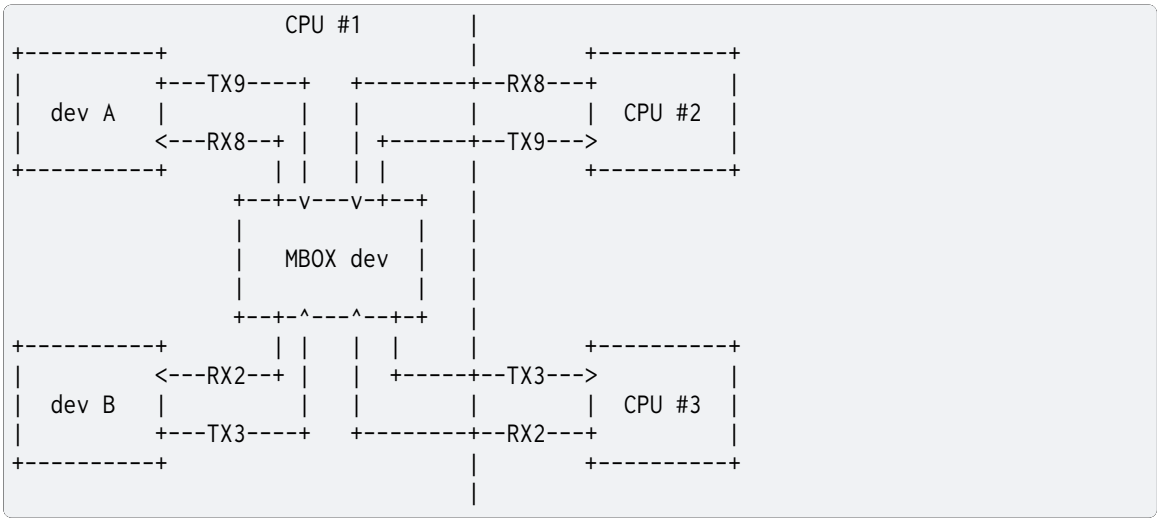
Related code samples

MBOX
Perform inter-processor mailbox communication using the MBOX API.

MBOX Data
Perform inter-processor mailbox communication using the MBOX API with data.

group mbox_interface

MBOX Interface.



Since
1.0

Version
0.1.0

An MBOX device is a peripheral capable of passing signals (and data depending on the peripheral) between CPUs and clusters in the system. Each MBOX instance is providing one or more channels, each one targeting one other CPU cluster (multiple channels can target the same cluster).

For example in the plot the device 'dev A' is using the TX channel 9 to signal (or send data to) the CPU #2 and it's expecting data or signals on the RX channel 8. Thus it can send the message through the channel 9, and it can register a callback on the channel 8 of the MBOX device.

This API supports two modes: signalling mode and data transfer mode.

In signalling mode:

- *mbox_mtu_get()* must return 0
- *mbox_send()* must have (msg == NULL)
- the callback must be called with (data == NULL)

In data transfer mode:

- *mbox_mtu_get()* must return a (value != 0)
- *mbox_send()* must have (msg != NULL)
- the callback must be called with (data != NULL)
- The msg content must be the same between sender and receiver

Defines

`MBOX_DT_SPEC_GET`(node_id, name)

Structure initializer for struct *mbox_dt_spec* from devicetree.

This helper macro expands to a static initializer for a struct *mbox_dt_spec* by reading the relevant device controller and channel number from the devicetree.

Example devicetree fragment:

```
n: node {
        mboxes = <&mbox1 8>,
                 <&mbox1 9>;
        mbox-names = "tx", "rx";
};
```

Example usage:

```
const struct mbox_dt_spec spec = MBOX_DT_SPEC_GET(DT_NODELABEL(n), tx);
```

Parameters

- **node_id** – Devicetree node identifier for the MBOX device
- **name** – lowercase-and-underscores name of the mboxes element

Returns
static initializer for a struct *mbox_dt_spec*

MBOX_DT_SPEC_INST_GET(inst, name)

Instance version of MBOX_DT_CHANNEL_GET()

Parameters

- **inst** – DT_DRV_COMPAT instance number
- **name** – lowercase-and-underscores name of the mboxes element

Returns
static initializer for a struct *mbox_dt_spec*

Typedefs

typedef uint32_t **mbox_channel_id_t**

Type for MBOX channel identifiers.

Functions

static inline bool **mbox_is_ready_dt**(const struct *mbox_dt_spec* *spec)

Validate if MBOX device instance from a struct *mbox_dt_spec* is ready.

Parameters

- **spec** – MBOX specification from devicetree

Returns
See return values for *mbox_send()*

int **mbox_send**(const struct *device* *dev, *mbox_channel_id_t* channel_id, const struct *mbox_msg* *msg)

Try to send a message over the MBOX device.

Send a message over an struct mbox_channel. The msg parameter must be NULL when the driver is used for signalling.

If the msg parameter is not NULL, this data is expected to be delivered on the receiving side using the data parameter of the receiving callback.

Parameters

- **dev** – MBOX device instance

- `channel_id` – MBOX channel identifier
- `msg` – Message

Return values

- `0` – On success.
- `-EBUSY` – If the remote hasn't yet read the last data sent.
- `-EMSGSIZE` – If the supplied data size is unsupported by the driver.
- `-EINVAL` – If there was a bad parameter, such as: too-large channel descriptor or the device isn't an outbound MBOX channel.

static inline int **mbox_send_dt**(const struct *mbox_dt_spec* *spec, const struct *mbox_msg* *msg)

Try to send a message over the MBOX device from a struct *mbox_dt_spec*.

Parameters

- `spec` – MBOX specification from devicetree
- `msg` – Message

Returns

See return values for *mbox_send()*

static inline int **mbox_register_callback**(const struct *device* *dev, *mbox_channel_id_t* channel_id, mbox_callback_t cb, void *user_data)

Register a callback function on a channel for incoming messages.

This function doesn't assume anything concerning the status of the interrupts. Use *mbox_set_enabled()* to enable or to disable the interrupts if needed.

Parameters

- `dev` – MBOX device instance
- `channel_id` – MBOX channel identifier
- `cb` – Callback function to execute on incoming message interrupts.
- `user_data` – Application-specific data pointer which will be passed to the callback function when executed.

Return values

- `0` – On success.
- `-errno` – Negative errno on error.

static inline int **mbox_register_callback_dt**(const struct *mbox_dt_spec* *spec, mbox_callback_t cb, void *user_data)

Register a callback function on a channel for incoming messages from a struct *mbox_dt_spec*.

Parameters

- `spec` – MBOX specification from devicetree
- `cb` – Callback function to execute on incoming message interrupts.
- `user_data` – Application-specific data pointer which will be passed to the callback function when executed.

Returns

See return values for *mbox_register_callback()*

int **mbox_mtu_get**(const struct *device* *dev)

Return the maximum number of bytes possible in an outbound message.

Returns the actual number of bytes that it is possible to send through an outgoing channel.

This number can be 0 when the driver only supports signalling or when on the receiving side the content and size of the message must be retrieved in an indirect way (i.e. probing some other peripheral, reading memory regions, etc...).

If this function returns 0, the msg parameter in *mbox_send()* is expected to be NULL.

Parameters

- **dev** – MBOX device instance.

Return values

- **>0** – Maximum possible size of a message in bytes
- **0** – Indicates signalling
- **-errno** – Negative errno on error.

static inline int **mbox_mtu_get_dt**(const struct *mbox_dt_spec* *spec)

Return the maximum number of bytes possible in an outbound message from struct *mbox_dt_spec*.

Parameters

- **spec** – MBOX specification from devicetree

Returns

See return values for *mbox_register_callback()*

int **mbox_set_enabled**(const struct *device* *dev, *mbox_channel_id_t* channel_id, bool enabled)

Enable (disable) interrupts and callbacks for inbound channels.

Enable interrupt for the channel when the parameter 'enable' is set to true. Disable it otherwise.

Immediately after calling this function with 'enable' set to true, the channel is considered enabled and ready to receive signal and messages (even already pending), so the user must take care of installing a proper callback (if needed) using *mbox_register_callback()* on the channel before enabling it. For this reason it is recommended that all the channels are disabled at probe time.

Enabling a channel for which there is no installed callback is considered undefined behavior (in general the driver must take care of gracefully handling spurious interrupts with no installed callback).

Parameters

- **dev** – MBOX device instance
- **channel_id** – MBOX channel identifier
- **enabled** – Enable (true) or disable (false) the channel.

Return values

- **0** – On success.
- **-EINVAL** – If it isn't an inbound channel.
- **-EALREADY** – If channel is already `enabled`.

static inline int **mbox_set_enabled_dt**(const struct *mbox_dt_spec* *spec, bool enabled)

Enable (disable) interrupts and callbacks for inbound channels from a struct *mbox_dt_spec*.

Parameters

- **spec** – MBOX specification from devicetree
- **enabled** – Enable (true) or disable (false) the channel.

Returns

See return values for *mbox_set_enabled()*

uint32_t **mbox_max_channels_get**(const struct *device* *dev)

Return the maximum number of channels.

Return the maximum number of channels supported by the hardware.

Parameters

- **dev** – MBOX device instance.

Returns

>0 Maximum possible number of supported channels on success

Returns

-errno Negative errno on error.

static inline int **mbox_max_channels_get_dt**(const struct *mbox_dt_spec* *spec)

Return the maximum number of channels from a struct *mbox_dt_spec*.

Parameters

- **spec** – MBOX specification from devicetree

Returns

See return values for *mbox_max_channels_get()*

struct **mbox_msg**

#include <*mbox.h*> Message struct (to hold data and its size).

Public Members

const void ***data**

Pointer to the data sent in the message.

size_t **size**

Size of the data.

struct **mbox_dt_spec**

#include <*mbox.h*> MBOX specification from DT.

Public Members

const struct *device* ***dev**

MBOX device pointer.

mbox_channel_id_t **channel_id**

Channel ID.

7.6.35 Peripheral Component Interconnect express Bus (PCIe)

Overview

API Reference

group `pcie_host_interface`

PCIe Host Interface.

Defines

`PCIE_ID_IS_VALID`(id)

`PCIE_DT_ID`(node_id)

Get the PCIe Vendor and Device ID for a node.

Parameters

- `node_id` – DTS node identifier

Returns

The VID/DID combination as pcie_id_t

`PCIE_DT_INST_ID`(inst)

Get the PCIe Vendor and Device ID for a node.

This is equivalent to *PCIE_DT_ID(DT_DRV_INST(inst))*

Parameters

- `inst` – Devicetree instance number

Returns

The VID/DID combination as pcie_id_t

`DEVICE_PCIE_DECLARE`(node_id)

Declare a PCIe context variable for a DTS node.

Declares a PCIe context for a DTS node. This must be done before using the *DEVICE_PCIE_INIT()* macro for the same node.

Parameters

- `node_id` – DTS node identifier

`DEVICE_PCIE_INST_DECLARE`(inst)

Declare a PCIe context variable for a DTS node.

This is equivalent to *DEVICE_PCIE_DECLARE(DT_DRV_INST(inst))*

Parameters

- `inst` – Devicetree instance number

`DEVICE_PCIE_INIT`(node_id, name)

Initialize a named struct member to point at a PCIe context.

Initialize PCIe-related information within a specific instance of a device config struct, using information from DTS. Using the macro requires having first created PCIe context struct using the *DEVICE_PCIE_DECLARE()* macro.

Example for an instance of a driver belonging to the “foo” subsystem

struct foo_config { struct *pcie_dev* *pcie; ... };

DEVICE_PCIE_ID_DECLARE(DT_DRV_INST(...)); struct foo_config my_config = { *DEVICE_PCIE_INIT(pcie, DT_DRV_INST(...))*, ... };

Parameters

- `node_id` – DTS node identifier
- `name` – Member name within config for the MMIO region

`DEVICE_PCIE_INST_INIT`(inst, name)

Initialize a named struct member to point at a PCIe context.

This is equivalent to *`DEVICE_PCIE_INIT(DT_DRV_INST(inst), name)`*

Parameters

- `inst` – Devicetree instance number
- `name` – Name of the struct member (of type struct *pcie_dev* *)

`PCIE_HOST_CONTROLLER`(n)

Get the BDF for a given PCI host controller.

This macro is useful when the PCI host controller behind PCIE_BDF(0, 0, 0) indicates a multifunction device. In such a case each function of this endpoint is a potential host controller itself.

Parameters

- `n` – Bus number

Returns

BDF value of the given host controller

`PCIE_CONF_CAPPTR`

`PCIE_CONF_CAPPTR_FIRST`(w)

`PCIE_CONF_CAP_ID`(w)

`PCIE_CONF_CAP_NEXT`(w)

`PCIE_CONF_EXT_CAPPTR`

`PCIE_CONF_EXT_CAP_ID`(w)

`PCIE_CONF_EXT_CAP_VER`(w)

`PCIE_CONF_EXT_CAP_NEXT`(w)

`PCIE_CONF_ID`

`PCIE_CONF_CMDSTAT`

`PCIE_CONF_CMDSTAT_IO`

`PCIE_CONF_CMDSTAT_MEM`

`PCIE_CONF_CMDSTAT_MASTER`

`PCIE_CONF_CMDSTAT_INTERRUPT`

PCIE_CONF_CMDSTAT_CAPS

PCIE_CONF_CLASSREV

PCIE_CONF_CLASSREV_CLASS(w)

PCIE_CONF_CLASSREV_SUBCLASS(w)

PCIE_CONF_CLASSREV_PROGIF(w)

PCIE_CONF_CLASSREV_REV(w)

PCIE_CONF_TYPE

PCIE_CONF_MULTIFUNCTION(w)

PCIE_CONF_TYPE_BRIDGE(w)

PCIE_CONF_TYPE_GET(w)

PCIE_CONF_TYPE_STANDARD

PCIE_CONF_TYPE_PCI_BRIDGE

PCIE_CONF_TYPE_CARDBUS_BRIDGE

PCIE_CONF_BAR0

PCIE_CONF_BAR1

PCIE_CONF_BAR2

PCIE_CONF_BAR3

PCIE_CONF_BAR4

PCIE_CONF_BAR5

PCIE_CONF_BAR_IO(w)

PCIE_CONF_BAR_MEM(w)

PCIE_CONF_BAR_64(w)

PCIE_CONF_BAR_ADDR(w)

PCIE_CONF_BAR_IO_ADDR(w)

PCIE_CONF_BAR_FLAGS(w)

PCIE_CONF_BAR_NONE

PCIE_CONF_BAR_INVAL

PCIE_CONF_BAR_INVAL64

PCIE_CONF_BAR_INVAL_FLAGS(w)

PCIE_BUS_NUMBER

PCIE_BUS_PRIMARY_NUMBER(w)

PCIE_BUS_SECONDARY_NUMBER(w)

PCIE_BUS_SUBORDINATE_NUMBER(w)

PCIE_SECONDARY_LATENCY_TIMER(w)

PCIE_BUS_NUMBER_VAL(prim, sec, sub, lat)

PCIE_IO_SEC_STATUS

PCIE_IO_BASE(w)

PCIE_IO_LIMIT(w)

PCIE_SEC_STATUS(w)

PCIE_IO_SEC_STATUS_VAL(iob, iol, sec_status)

PCIE_MEM_BASE_LIMIT

PCIE_MEM_BASE(w)

PCIE_MEM_LIMIT(w)

PCIE_MEM_BASE_LIMIT_VAL(memb, meml)

PCIE_PREFETCH_BASE_LIMIT

PCIE_PREFETCH_BASE(w)

PCIE_PREFETCH_LIMIT(w)

PCIE_PREFETCH_BASE_LIMIT_VAL(pmemb, pmeml)

PCIE_PREFETCH_BASE_UPPER

PCIE_PREFETCH_LIMIT_UPPER

PCIE_IO_BASE_LIMIT_UPPER

PCIE_IO_BASE_UPPER(w)

PCIE_IO_LIMIT_UPPER(w)

PCIE_IO_BASE_LIMIT_UPPER_VAL(iobu, iolu)

PCIE_CONF_INTR

PCIE_CONF_INTR_IRQ(w)

PCIE_CONF_INTR_IRQ_NONE

PCIE_MAX_BUS

PCIE_MAX_DEV

PCIE_MAX_FUNC

PCIE_IRQ_CONNECT(bdf_p, irq_p, priority_p, isr_p, isr_param_p, flags_p)

Initialize an interrupt handler for a PCIe endpoint IRQ.

This routine is only meant to be used by drivers using PCIe bus and having fixed or MSI based IRQ (so no runtime detection of the IRQ). In case of runtime detection see *pcie_connect_dynamic_irq()*

Parameters

- **bdf_p** – PCIe endpoint BDF
- **irq_p** – IRQ line number.
- **priority_p** – Interrupt priority.
- **isr_p** – Address of interrupt service routine.
- **isr_param_p** – Parameter passed to interrupt service routine.
- **flags_p** – Architecture-specific IRQ configuration flags..

Typedefs

typedef uint32_t **pcie_bdf_t**

A unique PCI(e) endpoint (bus, device, function).

A PCI(e) endpoint is uniquely identified topologically using a (bus, device, function) tuple. The internal structure is documented in include/dt-bindings/pcie/pcie.h: see PCIE_BDF() and friends, since these tuples are referenced from devicetree.

typedef uint32_t **pcie_id_t**

A unique PCI(e) identifier (vendor ID, device ID).

The PCIE_CONF_ID register for each endpoint is a (vendor ID, device ID) pair, which is meant to tell the system what the PCI(e) endpoint is. Again, look to PCIE_ID_* macros in include/dt-bindings/pcie/pcie.h for more.

typedef bool (***pcie_scan_cb_t**)(*pcie_bdf_t* bdf, *pcie_id_t* id, void *cb_data)

Callback type used for scanning for PCI endpoints.

Param bdf
BDF value for a found endpoint.

Param id
Vendor & Device ID for the found endpoint.

Param cb_data
Custom, use case specific data.

Return
true to continue scanning, false to stop scanning.

Enums

Values:

enumerator PCIE_SCAN_RECURSIVE = *BIT*(0)

Scan all available PCI host controllers and sub-busses.

enumerator PCIE_SCAN_CB_ALL = *BIT*(1)

Do the callback for all endpoint types, including bridges.

Functions

uint32_t pcie_conf_read(*pcie_bdf_t* bdf, unsigned int reg)

Read a 32-bit word from an endpoint's configuration space.

This function is exported by the arch/SoC/board code.

Parameters

- bdf – PCI(e) endpoint
- reg – the configuration word index (not address)

Returns

the word read (0xFFFFFFFFU if nonexistent endpoint or word)

void pcie_conf_write(*pcie_bdf_t* bdf, unsigned int reg, uint32_t data)

Write a 32-bit word to an endpoint's configuration space.

This function is exported by the arch/SoC/board code.

Parameters

- bdf – PCI(e) endpoint
- reg – the configuration word index (not address)
- data – the value to write

int pcie_scan(const struct *pcie_scan_opt* *opt)

Scan for PCIe devices.

Scan the PCI bus (or buses) for available endpoints.

Parameters

- opt – Options determining how to perform the scan.

Returns

0 on success, negative POSIX error number on failure.

bool pcie_get_mbar(*pcie_bdf_t* bdf, unsigned int bar_index, struct *pcie_bar* *mbar)

Get the MBAR at a specific BAR index.

Parameters

- bdf – the PCI(e) endpoint
- bar_index – 0-based BAR index
- mbar – Pointer to struct *pcie_bar*

Returns

true if the mbar was found and is valid, false otherwise

bool **pcie_probe_mbar**(*pcie_bdf_t* bdf, unsigned int index, struct *pcie_bar* *mbar)

Probe the nth MMIO address assigned to an endpoint.

A PCI(e) endpoint has 0 or more memory-mapped regions. This function allows the caller to enumerate them by calling with index=0..n. Value of n has to be below 6, as there is a maximum of 6 BARs. The indices are order-preserving with respect to the endpoint BARs: e.g., index 0 will return the lowest-numbered memory BAR on the endpoint.

Parameters

- **bdf** – the PCI(e) endpoint
- **index** – (0-based) index
- **mbar** – Pointer to struct *pcie_bar*

Returns

true if the mbar was found and is valid, false otherwise

bool **pcie_get_iobar**(*pcie_bdf_t* bdf, unsigned int bar_index, struct *pcie_bar* *iobar)

Get the I/O BAR at a specific BAR index.

Parameters

- **bdf** – the PCI(e) endpoint
- **bar_index** – 0-based BAR index
- **iobar** – Pointer to struct *pcie_bar*

Returns

true if the I/O BAR was found and is valid, false otherwise

bool **pcie_probe_iobar**(*pcie_bdf_t* bdf, unsigned int index, struct *pcie_bar* *iobar)

Probe the nth I/O BAR address assigned to an endpoint.

A PCI(e) endpoint has 0 or more I/O regions. This function allows the caller to enumerate them by calling with index=0..n. Value of n has to be below 6, as there is a maximum of 6 BARs. The indices are order-preserving with respect to the endpoint BARs: e.g., index 0 will return the lowest-numbered I/O BAR on the endpoint.

Parameters

- **bdf** – the PCI(e) endpoint
- **index** – (0-based) index
- **iobar** – Pointer to struct *pcie_bar*

Returns

true if the I/O BAR was found and is valid, false otherwise

void **pcie_set_cmd**(*pcie_bdf_t* bdf, uint32_t bits, bool on)

Set or reset bits in the endpoint command/status register.

Parameters

- **bdf** – the PCI(e) endpoint
- **bits** – the powerset of bits of interest
- **on** – use true to set bits, false to reset them

unsigned int **pcie_alloc_irq**(*pcie_bdf_t* bdf)

Allocate an IRQ for an endpoint.

This function first checks the IRQ register and if it contains a valid value this is returned. If the register does not contain a valid value allocation of a new one is attempted. Such function is only exposed if CONFIG_PCIE_CONTROLLER is unset. It is thus available where architecture tied dynamic IRQ allocation for PCIe device makes sense.

Parameters

- **bdf** – the PCI(e) endpoint

Returns
the IRQ number, or PCIE_CONF_INTR_IRQ_NONE if allocation failed.

unsigned int **pcie_get_irq**(*pcie_bdf_t* bdf)

Return the IRQ assigned by the firmware/board to an endpoint.

Parameters

- **bdf** – the PCI(e) endpoint

Returns
the IRQ number, or PCIE_CONF_INTR_IRQ_NONE if unknown.

void **pcie_irq_enable**(*pcie_bdf_t* bdf, unsigned int irq)

Enable the PCI(e) endpoint to generate the specified IRQ.

If MSI is enabled and the endpoint supports it, the endpoint will be configured to generate the specified IRQ via MSI. Otherwise, it is assumed that the IRQ has been routed by the boot firmware to the specified IRQ, and the IRQ is enabled (at the I/O APIC, or wherever appropriate).

Parameters

- **bdf** – the PCI(e) endpoint
- **irq** – the IRQ to generate

uint32_t **pcie_get_cap**(*pcie_bdf_t* bdf, uint32_t cap_id)

Find a PCI(e) capability in an endpoint's configuration space.

Parameters

- **bdf** – the PCI endpoint to examine
- **cap_id** – the capability ID of interest

Returns
the index of the configuration word, or 0 if no capability.

uint32_t **pcie_get_ext_cap**(*pcie_bdf_t* bdf, uint32_t cap_id)

Find an Extended PCI(e) capability in an endpoint's configuration space.

Parameters

- **bdf** – the PCI endpoint to examine
- **cap_id** – the capability ID of interest

Returns
the index of the configuration word, or 0 if no capability.

bool **pcie_connect_dynamic_irq**(*pcie_bdf_t* bdf, unsigned int irq, unsigned int priority, void (*routine)(const void *parameter), const void *parameter, uint32_t flags)

Dynamically connect a PCIe endpoint IRQ to an ISR handler.

Parameters

- **bdf** – the PCI endpoint to examine
- **irq** – the IRQ to connect (see *pcie_alloc_irq()*)
- **priority** – priority of the IRQ
- **routine** – the ISR handler to connect to the IRQ
- **parameter** – the parameter to provide to the handler
- **flags** – IRQ connection flags

Returns
true if connected, false otherwise

struct **pcie_dev**

#include <pcie.h>

struct **pcie_bar**

#include <pcie.h>

struct **pcie_scan_opt**

#include <pcie.h> Options for performing a scan for PCI devices.

Public Members

uint8_t **bus**

Initial bus number to scan.

pcie_scan_cb_t **cb**

Function to call for each found endpoint.

void ***cb_data**

Custom data to pass to the scan callback.

uint32_t **flags**

Scan flags.

7.6.36 Platform Environment Control Interface (PECI)

Overview

The Platform Environment Control Interface, abbreviated as PECI, is a thermal management standard introduced in 2006 with the Intel Core 2 Duo Microprocessors. The PECI interface allows external devices to read processor temperature, perform processor manageability functions, and manage processor interface tuning and diagnostics. The PECI bus driver APIs enable the interaction between Embedded Microcontrollers and CPUs.

Configuration Options

Related configuration options:

- CONFIG_PECI

API Reference

Related code samples

PECI interface
Monitor CPU temperature using PECI.

group peci_interface
PECI Interface 3.0.

Since
2.1

Version
1.0.0

PECI read/write supported responses.

PECI_CC_RSP_SUCCESS

PECI_CC_RSP_TIMEOUT

PECI_CC_OUT_OF_RESOURCES_TIMEOUT

PECI_CC_RESOURCES_LOWPWR_TIMEOUT

PECI_CC_ILLEGAL_REQUEST

Ping command format.

PECI_PING_WR_LEN

PECI_PING_RD_LEN

PECI_PING_LEN

GetDIB command format.

PECI_GET_DIB_WR_LEN

PECI_GET_DIB_RD_LEN

PECI_GET_DIB_CMD_LEN

PECI_GET_DIB_DEVINFO

PECI_GET_DIB_REVNUM

PECI_GET_DIB_DOMAIN_BIT_MASK

PECI_GET_DIB_MAJOR_REV_MASK

PECI_GET_DIB_MINOR_REV_MASK

GetTemp command format.

PECI_GET_TEMP_WR_LEN

PECI_GET_TEMP_RD_LEN

PECI_GET_TEMP_CMD_LEN

PECI_GET_TEMP_LSB

PECI_GET_TEMP_MSB

PECI_GET_TEMP_ERR_MSB

PECI_GET_TEMP_ERR_LSB_GENERAL

PECI_GET_TEMP_ERR_LSB_RES

PECI_GET_TEMP_ERR_LSB_TEMP_LO

PECI_GET_TEMP_ERR_LSB_TEMP_HI

RdPkgConfig command format.

PECI_RD_PKG_WR_LEN

PECI_RD_PKG_LEN_BYTE

PECI_RD_PKG_LEN_WORD

PECI_RD_PKG_LEN_DWORD

PECI_RD_PKG_CMD_LEN

WrPkgConfig command format.

PECI_WR_PKG_RD_LEN

PECI_WR_PKG_LEN_BYTE

PECI_WR_PKG_LEN_WORD

PECI_WR_PKG_LEN_DWORD

PECI_WR_PKG_CMD_LEN

RdIAMSR command format.

PECI_RD_IAMSR_WR_LEN

PECI_RD_IAMSR_LEN_BYTE

PECI_RD_IAMSR_LEN_WORD

PECI_RD_IAMSR_LEN_DWORD

PECI_RD_IAMSR_LEN_QWORD

PECI_RD_IAMSR_CMD_LEN

WrIAMSR command format.

PECI_WR_IAMSR_RD_LEN

PECI_WR_IAMSR_LEN_BYTE

PECI_WR_IAMSR_LEN_WORD

PECI_WR_IAMSR_LEN_DWORD

PECI_WR_IAMSR_LEN_QWORD

PECI_WR_IAMSR_CMD_LEN

RdPCIConfig command format.

PECI_RD_PCICFG_WR_LEN

PECI_RD_PCICFG_LEN_BYTE

PECI_RD_PCICFG_LEN_WORD

PECI_RD_PCICFG_LEN_DWORD

PECI_RD_PCICFG_CMD_LEN

WrPCIConfig command format.

PECI_WR_PCICFG_RD_LEN

PECI_WR_PCICFG_LEN_BYTE

PECI_WR_PCICFG_LEN_WORD

PECI_WR_PCICFG_LEN_DWORD

PECI_WR_PCICFG_CMD_LEN

RdPCIConfigLocal command format.

PECI_RD_PCICFGL_WR_LEN

PECI_RD_PCICFGL_RD_LEN_BYTE

PECI_RD_PCICFGL_RD_LEN_WORD

PECI_RD_PCICFGL_RD_LEN_DWORD

PECI_RD_PCICFGL_CMD_LEN

WrPCIConfigLocal command format.

PECI_WR_PCICFGL_RD_LEN

PECI_WR_PCICFGL_WR_LEN_BYTE

PECI_WR_PCICFGL_WR_LEN_WORD

PECI_WR_PCICFGL_WR_LEN_DWORD

PECI_WR_PCICFGL_CMD_LEN

Enums

enum peci_error_code

PECI error codes.

Values:

enumerator PECI_GENERAL_SENSOR_ERROR = 0x8000

enumerator PECI_UNDERFLOW_SENSOR_ERROR = 0x8002

enumerator PECI_OVERFLOW_SENSOR_ERROR = 0x8003

enum peci_command_code

PECI commands.

Values:

enumerator PECI_CMD_PING = 0x00

enumerator PECI_CMD_GET_TEMP0 = 0x01

enumerator PECI_CMD_GET_TEMP1 = 0x02

enumerator PECI_CMD_RD_PCI_CFG0 = 0x61

enumerator PECI_CMD_RD_PCI_CFG1 = 0x62

enumerator PECI_CMD_WR_PCI_CFG0 = 0x65

enumerator PECI_CMD_WR_PCI_CFG1 = 0x66

enumerator PECI_CMD_RD_PKG_CFG0 = 0xA1

enumerator PECI_CMD_RD_PKG_CFG1 = 0xA

enumerator PECI_CMD_WR_PKG_CFG0 = 0xA5

enumerator PECI_CMD_WR_PKG_CFG1 = 0xA6

enumerator PECI_CMD_RD_IAMSR0 = 0xB1

enumerator PECI_CMD_RD_IAMSR1 = 0xB2

enumerator PECI_CMD_WR_IAMSR0 = 0xB5

enumerator PECI_CMD_WR_IAMSR1 = 0xB6

enumerator PECI_CMD_RD_PCI_CFG_LOCAL0 = 0xE1

enumerator PECI_CMD_RD_PCI_CFG_LOCAL1 = 0xE2

enumerator PECI_CMD_WR_PCI_CFG_LOCAL0 = 0xE5

enumerator PECI_CMD_WR_PCI_CFG_LOCAL1 = 0xE6

enumerator PECI_CMD_GET_DIB = 0xF7

Functions

int peci_config(const struct *device* *dev, uint32_t bitrate)

Configures the PECI interface.

Parameters

- dev – Pointer to the device structure for the driver instance.
- bitrate – the selected bitrate expressed in Kbps.

Return values

- 0 – If successful.
- Negative – errno code if failure.

int peci_enable(const struct *device* *dev)

Enable PECI interface.

Parameters

- dev – Pointer to the device structure for the driver instance.

Return values

- 0 – If successful.
- Negative – errno code if failure.

int `peci_disable`(const struct *device* *dev)

Disable PECI interface.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Return values

- `0` – If successful.
- `Negative` – errno code if failure.

int `peci_transfer`(const struct *device* *dev, struct *peci_msg* *msg)

Performs a PECI transaction.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `msg` – Structure representing a PECI transaction.

Return values

- `0` – If successful.
- `Negative` – errno code if failure.

struct `peci_buf`

#include <*peci.h*> PECI buffer structure.

Public Members

uint8_t *`buf`

Valid pointer on a data buffer, or NULL otherwise.

size_t `len`

Length of the data buffer expected to be received without considering the frame check sequence byte.

> **Note**
>
> Frame check sequence byte is added into rx buffer: need to allocate an additional byte for this in rx buffer.

struct `peci_msg`

#include <*peci.h*> PECI transaction packet format.

Public Members

uint8_t `addr`

Client address.

enum *peci_command_code* `cmd_code`

Command code.

struct *peci_buf* **tx_buffer**
Pointer to buffer of write data.

struct *peci_buf* **rx_buffer**
Pointer to buffer of read data.

uint8_t **flags**
PECI msg flags.

7.6.37 PS/2

Overview

The PS/2 connector first hit the market in 1987 on IBM's desktop PC line of the same name before becoming an industry-wide standard for mouse and keyboard connections. Starting around 2007, USB superseded PS/2 and is the modern peripheral device connection standard. For legacy support on boards with a PS/2 connector, Zephyr provides these PS/2 driver APIs.

Configuration Options

Related configuration options:

- `CONFIG_PS2`

API Reference

Related code samples

PS/2 interface
Communicate with a PS/2 mouse.

group **ps2_interface**
PS/2 Driver APIs.

Typedefs

typedef void (***ps2_callback_t**)(const struct *device* *dev, uint8_t data)
PS/2 callback called when user types or click a mouse.

Param dev
Pointer to the device structure for the driver instance.

Param data
Data byte passed pack to the user.

Functions

int `ps2_config`(const struct *device* *dev, *ps2_callback_t* callback_isr)

Configure a ps2 instance.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `callback_isr` – called when PS/2 devices reply to a configuration command or when a mouse/keyboard send data to the client application.

Return values

- `0` – If successful.
- `Negative` – errno code if failure.

int `ps2_write`(const struct *device* *dev, uint8_t value)

Write to PS/2 device.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `value` – Data for the PS2 device.

Return values

- `0` – If successful.
- `Negative` – errno code if failure.

int `ps2_read`(const struct *device* *dev, uint8_t *value)

Read slave-to-host values from PS/2 device.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `value` – Pointer used for reading the PS/2 device.

Return values

- `0` – If successful.
- `Negative` – errno code if failure.

int `ps2_enable_callback`(const struct *device* *dev)

Enables callback.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Return values

- `0` – If successful.
- `Negative` – errno code if failure.

int `ps2_disable_callback`(const struct *device* *dev)

Disables callback.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Return values

- `0` – If successful.
- `Negative` – errno code if failure.

7.6.38 Pulse Width Modulation (PWM)

Overview

API Reference

Related code samples

Fade LED
Fade an LED using the PWM API.

PWM Blinky
Blink an LED using the PWM API.

PWM RGB LED
Drive an RGB LED using the PWM API.

Servomotor
Drive a servomotor using the PWM API.

group `pwm_interface`

PWM Interface.

Since
1.0

Version
1.0.0

PWM capture configuration flags

`PWM_CAPTURE_TYPE_PERIOD`
PWM pin capture captures period.

`PWM_CAPTURE_TYPE_PULSE`
PWM pin capture captures pulse width.

`PWM_CAPTURE_TYPE_BOTH`
PWM pin capture captures both period and pulse width.

`PWM_CAPTURE_MODE_SINGLE`
PWM pin capture captures a single period/pulse width.

`PWM_CAPTURE_MODE_CONTINUOUS`
PWM pin capture captures period/pulse width continuously.

PWM period set helpers

The period cell in the PWM specifier needs to be provided in nanoseconds.

However, in some applications it is more convenient to use another scale.

PWM_NSEC(x)
: Specify PWM period in nanoseconds.

PWM_USEC(x)
: Specify PWM period in microseconds.

PWM_MSEC(x)
: Specify PWM period in milliseconds.

PWM_SEC(x)
: Specify PWM period in seconds.

PWM_HZ(x)
: Specify PWM frequency in hertz.

PWM_KHZ(x)
: Specify PWM frequency in kilohertz.

PWM polarity flags

The PWM_POLARITY_* flags are used with *pwm_set_cycles()*, *pwm_set()* or *pwm_configure_capture()* to specify the polarity of a PWM channel.

The flags are on the lower 8bits of the pwm_flags_t

PWM_POLARITY_NORMAL
: PWM pin normal polarity (active-high pulse).

PWM_POLARITY_INVERTED
: PWM pin inverted polarity (active-low pulse).

Defines

PWM_DT_SPEC_GET_BY_NAME(node_id, name)
: Static initializer for a struct *pwm_dt_spec*.

  This returns a static initializer for a struct *pwm_dt_spec* given a devicetree node identifier and an index.

  Example devicetree fragment:

```
n: node {
    pwms = <&pwm1 1 1000 PWM_POLARITY_NORMAL>,
           <&pwm2 3 2000 PWM_POLARITY_INVERTED>;
    pwm-names = "alpha", "beta";
};
```

  Example usage:

```
const struct pwm_dt_spec spec =
    PWM_DT_SPEC_GET_BY_NAME(DT_NODELABEL(n), alpha);

// Initializes 'spec' to:
// {
//         .dev = DEVICE_DT_GET(DT_NODELABEL(pwm1)),
//         .channel = 1,
```

(continues on next page)

(continued from previous page)

```
//         .period = 1000,
//         .flags = PWM_POLARITY_NORMAL,
// }
```

The device (dev) must still be checked for readiness, e.g. using *device_is_ready()*. It is an error to use this macro unless the node exists, has the 'pwms' property, and that 'pwms' property specifies a PWM controller, a channel, a period in nanoseconds and optionally flags.

See also

PWM_DT_SPEC_INST_GET_BY_NAME

Parameters

- `node_id` – Devicetree node identifier.
- `name` – Lowercase-and-underscores name of a pwms element as defined by the node's pwm-names property.

Returns

Static initializer for a struct *pwm_dt_spec* for the property.

`PWM_DT_SPEC_INST_GET_BY_NAME`(inst, name)

Static initializer for a struct *pwm_dt_spec* from a DT_DRV_COMPAT instance.

See also

PWM_DT_SPEC_GET_BY_NAME

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `name` – Lowercase-and-underscores name of a pwms element as defined by the node's pwm-names property.

Returns

Static initializer for a struct *pwm_dt_spec* for the property.

`PWM_DT_SPEC_GET_BY_NAME_OR`(node_id, name, default_value)

Like *PWM_DT_SPEC_GET_BY_NAME()*, with a fallback to a default value.

If the devicetree node identifier 'node_id' refers to a node with a property 'pwms', this expands to *PWM_DT_SPEC_GET_BY_NAME(node_id, name)*. The `default_value` parameter is not expanded in this case. Otherwise, this expands to `default_value`.

See also

PWM_DT_SPEC_INST_GET_BY_NAME_OR

Parameters

- **node_id** – Devicetree node identifier.
- **name** – Lowercase-and-underscores name of a pwms element as defined by the node's pwm-names property
- **default_value** – Fallback value to expand to.

Returns
Static initializer for a struct *pwm_dt_spec* for the property, or default_value if the node or property do not exist.

PWM_DT_SPEC_INST_GET_BY_NAME_OR(inst, name, default_value)

Like *PWM_DT_SPEC_INST_GET_BY_NAME()*, with a fallback to a default value.

See also

PWM_DT_SPEC_GET_BY_NAME_OR

Parameters

- **inst** – DT_DRV_COMPAT instance number
- **name** – Lowercase-and-underscores name of a pwms element as defined by the node's pwm-names property.
- **default_value** – Fallback value to expand to.

Returns
Static initializer for a struct *pwm_dt_spec* for the property, or default_value if the node or property do not exist.

PWM_DT_SPEC_GET_BY_IDX(node_id, idx)

Static initializer for a struct *pwm_dt_spec*.

This returns a static initializer for a struct *pwm_dt_spec* given a devicetree node identifier and an index.

Example devicetree fragment:

```
n: node {
    pwms = <&pwm1 1 1000 PWM_POLARITY_NORMAL>,
           <&pwm2 3 2000 PWM_POLARITY_INVERTED>;
};
```

Example usage:

```
const struct pwm_dt_spec spec =
    PWM_DT_SPEC_GET_BY_IDX(DT_NODELABEL(n), 1);

// Initializes 'spec' to:
// {
//         .dev = DEVICE_DT_GET(DT_NODELABEL(pwm2)),
//         .channel = 3,
//         .period = 2000,
//         .flags = PWM_POLARITY_INVERTED,
// }
```

The device (dev) must still be checked for readiness, e.g. using *device_is_ready()*. It is an error to use this macro unless the node exists, has the 'pwms' property, and that 'pwms' property specifies a PWM controller, a channel, a period in nanoseconds and optionally flags.

See also

PWM_DT_SPEC_INST_GET_BY_IDX

Parameters

- `node_id` – Devicetree node identifier.
- `idx` – Logical index into 'pwms' property.

Returns
Static initializer for a struct *pwm_dt_spec* for the property.

`PWM_DT_SPEC_INST_GET_BY_IDX`(inst, idx)
Static initializer for a struct *pwm_dt_spec* from a DT_DRV_COMPAT instance.

See also

PWM_DT_SPEC_GET_BY_IDX

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `idx` – Logical index into 'pwms' property.

Returns
Static initializer for a struct *pwm_dt_spec* for the property.

`PWM_DT_SPEC_GET_BY_IDX_OR`(node_id, idx, default_value)
Like *PWM_DT_SPEC_GET_BY_IDX()*, with a fallback to a default value.

If the devicetree node identifier 'node_id' refers to a node with a property 'pwms', this expands to *PWM_DT_SPEC_GET_BY_IDX(node_id, idx)*. The `default_value` parameter is not expanded in this case. Otherwise, this expands to `default_value`.

See also

PWM_DT_SPEC_INST_GET_BY_IDX_OR

Parameters

- `node_id` – Devicetree node identifier.
- `idx` – Logical index into 'pwms' property.
- `default_value` – Fallback value to expand to.

Returns
Static initializer for a struct *pwm_dt_spec* for the property, or `default_value` if the node or property do not exist.

PWM_DT_SPEC_INST_GET_BY_IDX_OR(inst, idx, default_value)

Like *PWM_DT_SPEC_INST_GET_BY_IDX()*, with a fallback to a default value.

See also

PWM_DT_SPEC_GET_BY_IDX_OR

Parameters

- **inst** – DT_DRV_COMPAT instance number
- **idx** – Logical index into 'pwms' property.
- **default_value** – Fallback value to expand to.

Returns

Static initializer for a struct *pwm_dt_spec* for the property, or default_value if the node or property do not exist.

PWM_DT_SPEC_GET(node_id)

Equivalent to *PWM_DT_SPEC_GET_BY_IDX(node_id, 0)*.

See also

PWM_DT_SPEC_GET_BY_IDX

See also

PWM_DT_SPEC_INST_GET

Parameters

- **node_id** – Devicetree node identifier.

Returns

Static initializer for a struct *pwm_dt_spec* for the property.

PWM_DT_SPEC_INST_GET(inst)

Equivalent to *PWM_DT_SPEC_INST_GET_BY_IDX(inst, 0)*.

See also

PWM_DT_SPEC_INST_GET_BY_IDX

See also

PWM_DT_SPEC_GET

Parameters

- **inst** – DT_DRV_COMPAT instance number

Returns
Static initializer for a struct *pwm_dt_spec* for the property.

PWM_DT_SPEC_GET_OR(node_id, default_value)
Equivalent to *PWM_DT_SPEC_GET_BY_IDX_OR(node_id, 0, default_value)*.

See also
PWM_DT_SPEC_GET_BY_IDX_OR

See also
PWM_DT_SPEC_INST_GET_OR

Parameters

- **node_id** – Devicetree node identifier.
- **default_value** – Fallback value to expand to.

Returns
Static initializer for a struct *pwm_dt_spec* for the property.

PWM_DT_SPEC_INST_GET_OR(inst, default_value)
Equivalent to *PWM_DT_SPEC_INST_GET_BY_IDX_OR(inst, 0, default_value)*.

See also
PWM_DT_SPEC_INST_GET_BY_IDX_OR

See also
PWM_DT_SPEC_GET_OR

Parameters

- **inst** – DT_DRV_COMPAT instance number
- **default_value** – Fallback value to expand to.

Returns
Static initializer for a struct *pwm_dt_spec* for the property.

Typedefs

typedef uint16_t **pwm_flags_t**
Provides a type to hold PWM configuration flags.

The lower 8 bits are used for standard flags. The upper 8 bits are reserved for SoC specific flags.

See also

PWM_CAPTURE_FLAGS.

typedef void (***pwm_capture_callback_handler_t**)(const struct *device* *dev, uint32_t channel, uint32_t period_cycles, uint32_t pulse_cycles, int status, void *user_data)

PWM capture callback handler function signature.

Note

The callback handler will be called in interrupt context.

Note

`CONFIG_PWM_CAPTURE` must be selected to enable PWM capture support.

Param dev
[in] PWM device instance.

Param channel
PWM channel.

Param period_cycles
Captured PWM period width (in clock cycles). HW specific.

Param pulse_cycles
Captured PWM pulse width (in clock cycles). HW specific.

Param status
Status for the PWM capture (0 if no error, negative errno otherwise. See *pwm_capture_cycles()* return value descriptions for details).

Param user_data
User data passed to *pwm_configure_capture()*

Functions

int **pwm_set_cycles**(const struct *device* *dev, uint32_t channel, uint32_t period, uint32_t pulse, *pwm_flags_t* flags)

Set the period and pulse width for a single PWM output.

The PWM period and pulse width will synchronously be set to the new values without glitches in the PWM signal, but the call will not block for the change to take effect.

Passing 0 as `pulse` will cause the pin to be driven to a constant inactive level. Passing a non-zero `pulse` equal to `period` will cause the pin to be driven to a constant active level.

Note

Not all PWM controllers support synchronous, glitch-free updates of the PWM period and pulse width. Depending on the hardware, changing the PWM period and/or pulse width may cause a glitch in the generated PWM signal.

Note

Some multi-channel PWM controllers share the PWM period across all channels. Depending on the hardware, changing the PWM period for one channel may affect the PWM period for the other channels of the same PWM controller.

Parameters

- `dev` – **[in]** PWM device instance.
- `channel` – PWM channel.
- `period` – Period (in clock cycles) set to the PWM. HW specific.
- `pulse` – Pulse width (in clock cycles) set to the PWM. HW specific.
- `flags` – Flags for pin configuration.

Return values

- `0` – If successful.
- `-EINVAL` – If pulse > period.
- `-errno` – Negative errno code on failure.

int `pwm_get_cycles_per_sec`(const struct *device* *dev, uint32_t channel, uint64_t *cycles)

Get the clock rate (cycles per second) for a single PWM output.

Parameters

- `dev` – **[in]** PWM device instance.
- `channel` – PWM channel.
- `cycles` – **[out]** Pointer to the memory to store clock rate (cycles per sec). HW specific.

Return values

- `0` – If successful.
- `-errno` – Negative errno code on failure.

static inline int `pwm_set`(const struct *device* *dev, uint32_t channel, uint32_t period, uint32_t pulse, *pwm_flags_t* flags)

Set the period and pulse width in nanoseconds for a single PWM output.

Note

Utility macros such as *PWM_MSEC()* can be used to convert from other scales or units to nanoseconds, the units used by this function.

Parameters

- `dev` – **[in]** PWM device instance.

- **channel** – PWM channel.
- **period** – Period (in nanoseconds) set to the PWM.
- **pulse** – Pulse width (in nanoseconds) set to the PWM.
- **flags** – Flags for pin configuration (polarity).

Return values

- **0** – If successful.
- **-ENOTSUP** – If requested period or pulse cycles are not supported.
- **-errno** – Other negative errno code on failure.

static inline int **pwm_set_dt**(const struct *pwm_dt_spec* *spec, uint32_t period, uint32_t pulse)

Set the period and pulse width in nanoseconds from a struct *pwm_dt_spec* (with custom period).

This is equivalent to:

```
pwm_set(spec->dev, spec->channel, period, pulse, spec->flags)
```

The period specified in spec is ignored. This API call can be used when the period specified in Devicetree needs to be changed at runtime.

See also

pwm_set_pulse_dt()

Parameters

- **spec** – **[in]** PWM specification from devicetree.
- **period** – Period (in nanoseconds) set to the PWM.
- **pulse** – Pulse width (in nanoseconds) set to the PWM.

Returns

A value from *pwm_set()*.

static inline int **pwm_set_pulse_dt**(const struct *pwm_dt_spec* *spec, uint32_t pulse)

Set the period and pulse width in nanoseconds from a struct *pwm_dt_spec*.

This is equivalent to:

```
pwm_set(spec->dev, spec->channel, spec->period, pulse, spec->flags)
```

See also

pwm_set_pulse_dt()

Parameters

- **spec** – **[in]** PWM specification from devicetree.
- **pulse** – Pulse width (in nanoseconds) set to the PWM.

Returns

A value from *pwm_set()*.

static inline int **pwm_cycles_to_usec**(const struct *device* *dev, uint32_t channel, uint32_t cycles, uint64_t *usec)

Convert from PWM cycles to microseconds.

Parameters

- `dev` – **[in]** PWM device instance.
- `channel` – PWM channel.
- `cycles` – Cycles to be converted.
- `usec` – **[out]** Pointer to the memory to store calculated usec.

Return values

- `0` – If successful.
- `-ERANGE` – If result is too large.
- `-errno` – Other negative errno code on failure.

static inline int **pwm_cycles_to_nsec**(const struct *device* *dev, uint32_t channel, uint32_t cycles, uint64_t *nsec)

Convert from PWM cycles to nanoseconds.

Parameters

- `dev` – **[in]** PWM device instance.
- `channel` – PWM channel.
- `cycles` – Cycles to be converted.
- `nsec` – **[out]** Pointer to the memory to store the calculated nsec.

Return values

- `0` – If successful.
- `-ERANGE` – If result is too large.
- `-errno` – Other negative errno code on failure.

static inline int **pwm_configure_capture**(const struct *device* *dev, uint32_t channel, *pwm_flags_t* flags, *pwm_capture_callback_handler_t* cb, void *user_data)

Configure PWM period/pulse width capture for a single PWM input.

After configuring PWM capture using this function, the capture can be enabled/disabled using *pwm_enable_capture()* and *pwm_disable_capture()*.

Note

This API function cannot be invoked from user space due to the use of a function callback. In user space, one of the simpler API functions (*pwm_capture_cycles()*, *pwm_capture_usec()*, or *pwm_capture_nsec()*) can be used instead.

Note

`CONFIG_PWM_CAPTURE` must be selected for this function to be available.

Parameters

- `dev` – **[in]** PWM device instance.
- `channel` – PWM channel.
- `flags` – PWM capture flags
- `cb` – **[in]** Application callback handler function to be called upon capture
- `user_data` – **[in]** User data to pass to the application callback handler function

Return values

- `-EINVAL` – if invalid function parameters were given
- `-ENOSYS` – if PWM capture is not supported or the given flags are not supported
- `-EIO` – if IO error occurred while configuring
- `-EBUSY` – if PWM capture is already in progress

int `pwm_enable_capture`(const struct *device* *dev, uint32_t channel)

Enable PWM period/pulse width capture for a single PWM input.

The PWM pin must be configured using *pwm_configure_capture()* prior to calling this function.

Note

`CONFIG_PWM_CAPTURE` must be selected for this function to be available.

Parameters

- `dev` – **[in]** PWM device instance.
- `channel` – PWM channel.

Return values

- `0` – If successful.
- `-EINVAL` – if invalid function parameters were given
- `-ENOSYS` – if PWM capture is not supported
- `-EIO` – if IO error occurred while enabling PWM capture
- `-EBUSY` – if PWM capture is already in progress

int `pwm_disable_capture`(const struct *device* *dev, uint32_t channel)

Disable PWM period/pulse width capture for a single PWM input.

Note

`CONFIG_PWM_CAPTURE` must be selected for this function to be available.

Parameters

- `dev` – **[in]** PWM device instance.
- `channel` – PWM channel.

Return values

- `0` – If successful.

- `-EINVAL` – if invalid function parameters were given
- `-ENOSYS` – if PWM capture is not supported
- `-EIO` – if IO error occurred while disabling PWM capture

int **pwm_capture_cycles**(const struct *device* *dev, uint32_t channel, *pwm_flags_t* flags, uint32_t *period, uint32_t *pulse, *k_timeout_t* timeout)

Capture a single PWM period/pulse width in clock cycles for a single PWM input.

This API function wraps calls to *pwm_configure_capture()*, *pwm_enable_capture()*, and *pwm_disable_capture()* and passes the capture result to the caller. The function is blocking until either the PWM capture is completed or a timeout occurs.

Note

`CONFIG_PWM_CAPTURE` must be selected for this function to be available.

Parameters

- `dev` – **[in]** PWM device instance.
- `channel` – PWM channel.
- `flags` – PWM capture flags.
- `period` – **[out]** Pointer to the memory to store the captured PWM period width (in clock cycles). HW specific.
- `pulse` – **[out]** Pointer to the memory to store the captured PWM pulse width (in clock cycles). HW specific.
- `timeout` – Waiting period for the capture to complete.

Return values

- `0` – If successful.
- `-EBUSY` – PWM capture already in progress.
- `-EAGAIN` – Waiting period timed out.
- `-EIO` – IO error while capturing.
- `-ERANGE` – If result is too large.

static inline int **pwm_capture_usec**(const struct *device* *dev, uint32_t channel, *pwm_flags_t* flags, uint64_t *period, uint64_t *pulse, *k_timeout_t* timeout)

Capture a single PWM period/pulse width in microseconds for a single PWM input.

This API function wraps calls to *pwm_capture_cycles()* and *pwm_cycles_to_usec()* and passes the capture result to the caller. The function is blocking until either the PWM capture is completed or a timeout occurs.

Note

`CONFIG_PWM_CAPTURE` must be selected for this function to be available.

Parameters

- `dev` – **[in]** PWM device instance.
- `channel` – PWM channel.

- `flags` – PWM capture flags.
- `period` – **[out]** Pointer to the memory to store the captured PWM period width (in usec).
- `pulse` – **[out]** Pointer to the memory to store the captured PWM pulse width (in usec).
- `timeout` – Waiting period for the capture to complete.

Return values

- `0` – If successful.
- `-EBUSY` – PWM capture already in progress.
- `-EAGAIN` – Waiting period timed out.
- `-EIO` – IO error while capturing.
- `-ERANGE` – If result is too large.
- `-errno` – Other negative errno code on failure.

static inline int **pwm_capture_nsec**(const struct *device* *dev, uint32_t channel, *pwm_flags_t* flags, uint64_t *period, uint64_t *pulse, *k_timeout_t* timeout)

Capture a single PWM period/pulse width in nanoseconds for a single PWM input.

This API function wraps calls to *pwm_capture_cycles()* and *pwm_cycles_to_nsec()* and passes the capture result to the caller. The function is blocking until either the PWM capture is completed or a timeout occurs.

Note

`CONFIG_PWM_CAPTURE` must be selected for this function to be available.

Parameters

- `dev` – **[in]** PWM device instance.
- `channel` – PWM channel.
- `flags` – PWM capture flags.
- `period` – **[out]** Pointer to the memory to store the captured PWM period width (in nsec).
- `pulse` – **[out]** Pointer to the memory to store the captured PWM pulse width (in nsec).
- `timeout` – Waiting period for the capture to complete.

Return values

- `0` – If successful.
- `-EBUSY` – PWM capture already in progress.
- `-EAGAIN` – Waiting period timed out.
- `-EIO` – IO error while capturing.
- `-ERANGE` – If result is too large.
- `-errno` – Other negative errno code on failure.

static inline bool **pwm_is_ready_dt**(const struct *pwm_dt_spec* *spec)

Validate that the PWM device is ready.

Parameters

- **spec** – PWM specification from devicetree

Return values

- **true** – If the PWM device is ready for use
- **false** – If the PWM device is not ready for use

struct **pwm_dt_spec**

#include <pwm.h> Container for PWM information specified in devicetree.

This type contains a pointer to a PWM device, channel number (controlled by the PWM device), the PWM signal period in nanoseconds and the flags applicable to the channel. Note that not all PWM drivers support flags. In such case, flags will be set to 0.

See also

PWM_DT_SPEC_GET_BY_NAME

See also

PWM_DT_SPEC_GET_BY_NAME_OR

See also

PWM_DT_SPEC_GET_BY_IDX

See also

PWM_DT_SPEC_GET_BY_IDX_OR

See also

PWM_DT_SPEC_GET

See also

PWM_DT_SPEC_GET_OR

Public Members

const struct *device* *dev

PWM device instance.

uint32_t **channel**
: Channel number.

uint32_t **period**
: Period in nanoseconds.

pwm_flags_t **flags**
: Flags.

7.6.39 Real-Time Clock (RTC)

Overview

Table 1: **Glossary**

Word	Definition
Real-time clock	Low power device tracking time using broken-down time
Real-time counter	Low power counter which can be used to track time
RTC	Acronym for real-time clock

An RTC is a low power device which tracks time using broken-down time. It should not be confused with low-power counters which sometimes share the same name, acronym, or both.

RTCs are usually optimized for low energy consumption and are usually kept running even when the system is in a low power state.

RTCs usually contain one or more alarms which can be configured to trigger at a given time. These alarms are commonly used to wake up the system from a low power state.

History of RTCs in Zephyr

RTCs have been supported before this API was created, using the *Counter* API. The unix timestamp was used to convert between broken-down time and the unix timestamp within the RTC drivers, which internally used the broken-down time representation.

The disadvantages of this approach were that hardware counters could not be set to a specific count, requiring all RTCs to use device specific APIs to set the time, converting from unix time to broken-down time, unnecessarily in some cases, and some common features missing, like input clock calibration and the update callback.

Configuration Options

Related configuration options:

- `CONFIG_RTC`
- `CONFIG_RTC_ALARM`
- `CONFIG_RTC_UPDATE`
- `CONFIG_RTC_CALIBRATION`

API Reference

group `rtc_interface`

RTC Interface.

Since
3.4

Version
0.1.0

RTC Interface Alarm

int `rtc_alarm_get_supported_fields`(const struct *device* *dev, uint16_t id, uint16_t *mask)

API for getting the supported fields of the RTC alarm time.

> **Note**
> Bits in the mask param are defined here *RTC_ALARM_TIME_MASK*.

Parameters

- `dev` – Device instance
- `id` – Id of the alarm
- `mask` – Mask of fields in the alarm time which are supported

Returns
0 if successful

Returns
-EINVAL if id is out of range or time is invalid

Returns
-ENOTSUP if API is not supported by hardware

Returns
-errno code if failure

int `rtc_alarm_set_time`(const struct *device* *dev, uint16_t id, uint16_t mask, const struct *rtc_time* *timeptr)

API for setting RTC alarm time.

To enable an RTC alarm, one or more fields of the RTC alarm time must be enabled. The mask designates which fields of the RTC alarm time to enable. If the mask parameter is 0, the alarm will be disabled. The RTC alarm will trigger when all enabled fields of the alarm time match the RTC time.

> **Note**
> The timeptr param may be NULL if the mask param is 0

Note

Only the enabled fields in the timeptr param need to be configured

Note

Bits in the mask param are defined here *RTC_ALARM_TIME_MASK*

Parameters

- `dev` – Device instance
- `id` – Id of the alarm
- `mask` – Mask of fields in the alarm time to enable
- `timeptr` – The alarm time to set

Returns
0 if successful

Returns
-EINVAL if id is out of range or time is invalid

Returns
-ENOTSUP if API is not supported by hardware

Returns
-errno code if failure

int `rtc_alarm_get_time`(const struct *device* *dev, uint16_t id, uint16_t *mask, struct *rtc_time* *timeptr)

API for getting RTC alarm time.

Note

Bits in the mask param are defined here *RTC_ALARM_TIME_MASK*

Parameters

- `dev` – Device instance
- `id` – Id of the alarm
- `mask` – Destination for mask of fields which are enabled in the alarm time
- `timeptr` – Destination for the alarm time

Returns
0 if successful

Returns
-EINVAL if id is out of range

Returns
-ENOTSUP if API is not supported by hardware

Returns
-errno code if failure

int `rtc_alarm_is_pending`(const struct *device* *dev, uint16_t id)

API for testing if RTC alarm is pending.

Test whether or not the alarm with id is pending. If the alarm is pending, the pending status is cleared.

Parameters

- `dev` – Device instance
- `id` – Id of the alarm to test

Returns
1 if alarm was pending

Returns
0 if alarm was not pending

Returns
-EINVAL if id is out of range

Returns
-ENOTSUP if API is not supported by hardware

Returns
-errno code if failure

int `rtc_alarm_set_callback`(const struct *device* *dev, uint16_t id, *rtc_alarm_callback* callback, void *user_data)

API for setting alarm callback.

Setting the alarm callback for an alarm, will enable the alarm callback. When the callback for an alarm is enabled, the alarm triggered event will invoke the callback, after which the alarm pending status will be cleared automatically. The alarm will remain enabled until manually disabled using *rtc_alarm_set_time()*.

To disable the alarm callback for an alarm, the `callback` and `user_data` parameters must be set to NULL. When the alarm callback for an alarm is disabled, the alarm triggered event will set the alarm status to "pending". To check if the alarm status is "pending", use *rtc_alarm_is_pending()*.

Parameters

- `dev` – Device instance
- `id` – Id of the alarm for which the callback shall be set
- `callback` – Callback called when alarm occurs
- `user_data` – Optional user data passed to callback

Returns
0 if successful

Returns
-EINVAL if id is out of range

Returns
-ENOTSUP if API is not supported by hardware

Returns
-errno code if failure

RTC Interface Update

int `rtc_update_set_callback`(const struct *device* *dev, *rtc_update_callback* callback, void *user_data)

API for setting update callback.

Setting the update callback will enable the update callback. The update callback will be invoked every time the RTC clock is updated by 1 second. It can be used to synchronize the RTC clock with other clock sources.

To disable the update callback for the RTC clock, the `callback` and `user_data` parameters must be set to NULL.

Parameters

- `dev` – Device instance
- `callback` – Callback called when update occurs
- `user_data` – Optional user data passed to callback

Returns
0 if successful

Returns
-ENOTSUP if API is not supported by hardware

Returns
-errno code if failure

RTC Interface Calibration

int `rtc_set_calibration`(const struct *device* *dev, int32_t calibration)

API for setting RTC calibration.

Calibration is applied to the RTC clock input. A positive calibration value will increase the frequency of the RTC clock, a negative value will decrease the frequency of the RTC clock.

See also

rtc_calibration_from_frequency()

Parameters

- `dev` – Device instance
- `calibration` – Calibration to set in parts per billion

Returns
0 if successful

Returns
-EINVAL if calibration is out of range

Returns
-ENOTSUP if API is not supported by hardware

Returns
-errno code if failure

int **rtc_get_calibration**(const struct *device* *dev, int32_t *calibration)

API for getting RTC calibration.

Parameters

- **dev** – Device instance
- **calibration** – Destination for calibration in parts per billion

Returns

0 if successful

Returns

-ENOTSUP if API is not supported by hardware

Returns

-errno code if failure

RTC Interface Helpers

static inline struct tm ***rtc_time_to_tm**(struct *rtc_time* *timeptr)

Convenience function for safely casting a *rtc_time* pointer to a tm pointer.

static inline int32_t **rtc_calibration_from_frequency**(uint32_t frequency)

Determine required calibration to 1 Hertz from frequency.

Parameters

- **frequency** – Frequency of the RTC in nano Hertz

Returns

The required calibration in parts per billion

RTC Alarm Time Mask

Mask for alarm time fields to enable when setting alarm time

`RTC_ALARM_TIME_MASK_SECOND`

`RTC_ALARM_TIME_MASK_MINUTE`

`RTC_ALARM_TIME_MASK_HOUR`

`RTC_ALARM_TIME_MASK_MONTHDAY`

`RTC_ALARM_TIME_MASK_MONTH`

`RTC_ALARM_TIME_MASK_YEAR`

`RTC_ALARM_TIME_MASK_WEEKDAY`

`RTC_ALARM_TIME_MASK_YEARDAY`

`RTC_ALARM_TIME_MASK_NSEC`

Typedefs

typedef void (*`rtc_update_callback`)(const struct *device* *dev, void *user_data)

RTC update event callback.

Param dev
Device instance invoking the handler

Param user_data
Optional user data provided when update irq callback is set

typedef void (*`rtc_alarm_callback`)(const struct *device* *dev, uint16_t id, void *user_data)

RTC alarm triggered callback.

Param dev
Device instance invoking the handler

Param id
Alarm id

Param user_data
Optional user data passed with the alarm configuration

Functions

int `rtc_set_time`(const struct *device* *dev, const struct *rtc_time* *timeptr)

API for setting RTC time.

Parameters

- `dev` – Device instance
- `timeptr` – The time to set

Returns
0 if successful

Returns
-EINVAL if RTC time is invalid or exceeds hardware capabilities

Returns
-errno code if failure

int `rtc_get_time`(const struct *device* *dev, struct *rtc_time* *timeptr)

API for getting RTC time.

Parameters

- `dev` – Device instance
- `timeptr` – Destination for the time

Returns
0 if successful

Returns
-ENODATA if RTC time has not been set

Returns
-errno code if failure

struct **rtc_time**

#include <rtc.h> Structure for storing date and time values with sub-second precision.

The structure is 1-1 mapped to the struct tm for the members `tm_sec` to `tm_isdst` making it compatible with the standard time library.

> **Note**
>
> Use *rtc_time_to_tm()* to safely cast from a *rtc_time* pointer to a tm pointer.

Public Members

int **tm_sec**

Seconds [0, 59].

int **tm_min**

Minutes [0, 59].

int **tm_hour**

Hours [0, 23].

int **tm_mday**

Day of the month [1, 31].

int **tm_mon**

Month [0, 11].

int **tm_year**

Year - 1900.

int **tm_wday**

Day of the week [0, 6] (Sunday = 0) (Unknown = -1)

int **tm_yday**

Day of the year [0, 365] (Unknown = -1)

int **tm_isdst**

Daylight saving time flag [-1] (Unknown = -1)

int **tm_nsec**

Nanoseconds [0, 999999999] (Unknown = 0)

RTC device driver test suite

The test suite validates the behavior of the RTC device driver. It is designed to be portable between boards. It uses the device tree alias `rtc` to designate the RTC device to test.

This test suite tests the following:

- Setting and getting the time.

- RTC Time incrementing correctly.
- Alarms if supported by hardware, with and without callback enabled
- Calibration if supported by hardware.

The calibration test tests a range of values which are printed to the console to be manually compared. The user must review the set and gotten values to ensure they are valid.

By default, only the mandatory setting and getting of time is enabled for testing. To test the optional alarms, update event callback and clock calibration, these must be enabled by selecting `CONFIG_RTC_ALARM`, `CONFIG_RTC_UPDATE` and `CONFIG_RTC_CALIBRATION`.

The following examples build the test suite for the `native_sim` board. To build the test suite for a different board, replace the `native_sim` board with your board.

To build the test application with the default configuration, testing only the mandatory features, the following command can be used for reference:

```
# From the root of the zephyr repository
west build -b native_sim tests/drivers/rtc/rtc_api
```

To build the test with additional RTC features enabled, use menuconfig to enable the additional features by updating the configuration. The following command can be used for reference:

```
# From the root of the zephyr repository
west build -b native_sim tests/drivers/rtc/rtc_api
west build -t menuconfig
```

Then build the test application using the following command:

```
# From the root of the zephyr repository
west build -b native_sim tests/drivers/rtc/rtc_api
```

To run the test suite, flash and run the application on your board, the output will be printed to the console.

Note

The tests take up to 30 seconds each if they are testing real hardware.

RTC emulated device

The emulated RTC device fully implements the RTC API, and will behave like a real RTC device, with the following limitations:

- RTC time is not persistent across application initialization.
- RTC alarms are not persistent across application initialization.
- RTC time will drift over time.

Every time an application is initialized, the RTC's time and alarms are reset. Reading the time using *rtc_get_time()* will return `-ENODATA`, until the time is set using *rtc_set_time()*. The RTC will then behave as a real RTC, until the application is reset.

The emulated RTC device driver is built for the compatible `zephyr,rtc-emul` and will be included if `CONFIG_RTC` is selected.

7.6.40 Regulators

This subsystem provides control of voltage and current regulators. A common example is a GPIO that controls a transistor that supplies current to a device that is not always needed. Another example is a PMIC, typically a much more complex device.

The `*-supply` devicetree properties are used to identify the regulator(s) that a devicetree node directly depends on. Within the driver for the node the regulator API is used to issue requests for power when the device is to be active, and release the power request when the device shuts down.

The simplest case where a regulator is needed is one where there is only one client. For those situations the cost of using the regulator device infrastructure is not justified, and `*-gpios` devicetree properties should be used. There is no device interface to these regulators as they are entirely controlled within the driver for the corresponding node, e.g. a sensor.

API Reference

group `regulator_interface`

Regulator Interface.

Since
2.4

Version
0.1.0

Regulator error flags.

`REGULATOR_ERROR_OVER_VOLTAGE`

Voltage is too high.

`REGULATOR_ERROR_OVER_CURRENT`

Current is too high.

`REGULATOR_ERROR_OVER_TEMP`

Temperature is too high.

Typedefs

typedef uint8_t `regulator_dvs_state_t`

Opaque type to store regulator DVS states.

typedef uint8_t `regulator_mode_t`

Opaque type to store regulator modes.

typedef uint8_t `regulator_error_flags_t`

Opaque bit map for regulator error flags (see *REGULATOR_ERRORS*)

Functions

int **regulator_enable**(const struct *device* *dev)

Enable a regulator.

Reference-counted request that a regulator be turned on. A regulator is considered "on" when it has reached a stable/usable state. Regulators that are always on, or configured in devicetree with `regulator-always-on` will always stay enabled, and so this function will always succeed.

Parameters

- **dev** – Regulator device instance

Return values

- **0** – If regulator has been successfully enabled.
- **-errno** – Negative errno in case of failure.
- **-ENOTSUP** – If regulator enablement can not be controlled.

bool **regulator_is_enabled**(const struct *device* *dev)

Check if a regulator is enabled.

Parameters

- **dev** – Regulator device instance.

Return values

- **true** – If regulator is enabled.
- **false** – If regulator is disabled.

int **regulator_disable**(const struct *device* *dev)

Disable a regulator.

Release a regulator after a previous *regulator_enable()* completed successfully. Regulators that are always on, or configured in devicetree with `regulator-always-on` will always stay enabled, and so this function will always succeed.

This must be invoked at most once for each successful *regulator_enable()*.

Parameters

- **dev** – Regulator device instance.

Return values

- **0** – If regulator has been successfully disabled.
- **-errno** – Negative errno in case of failure.
- **-ENOTSUP** – If regulator disablement can not be controlled.

static inline unsigned int **regulator_count_voltages**(const struct *device* *dev)

Obtain the number of supported voltage levels.

Each voltage level supported by a regulator gets an index, starting from zero. The total number of supported voltage levels can be used together with *regulator_list_voltage()* to list all supported voltage levels.

Parameters

- **dev** – Regulator device instance.

Returns

Number of supported voltages.

static inline int **regulator_list_voltage**(const struct *device* *dev, unsigned int idx, int32_t *volt_uv)

Obtain the value of a voltage given an index.

Each voltage level supported by a regulator gets an index, starting from zero. Together with *regulator_count_voltages()*, this function can be used to iterate over all supported voltages.

Parameters

- **dev** – Regulator device instance.
- **idx** – Voltage index.
- **volt_uv** – **[out]** Where voltage for the given `index` will be stored, in microvolts.

Return values

- **0** – If `index` corresponds to a supported voltage.
- **-EINVAL** – If `index` does not correspond to a supported voltage.

bool **regulator_is_supported_voltage**(const struct *device* *dev, int32_t min_uv, int32_t max_uv)

Check if a voltage within a window is supported.

Parameters

- **dev** – Regulator device instance.
- **min_uv** – Minimum voltage in microvolts.
- **max_uv** – maximum voltage in microvolts.

Return values

- **true** – If voltage is supported.
- **false** – If voltage is not supported.

int **regulator_set_voltage**(const struct *device* *dev, int32_t min_uv, int32_t max_uv)

Set the output voltage.

The output voltage will be configured to the closest supported output voltage. *regulator_get_voltage()* can be used to obtain the actual configured voltage. The voltage will be applied to the active or selected mode. Output voltage may be limited using `regulator-min-microvolt` and/or `regulator-max-microvolt` in devicetree.

Parameters

- **dev** – Regulator device instance.
- **min_uv** – Minimum acceptable voltage in microvolts.
- **max_uv** – Maximum acceptable voltage in microvolts.

Return values

- **0** – If successful.
- **-EINVAL** – If the given voltage window is not valid.
- **-ENOSYS** – If function is not implemented.
- **-errno** – In case of any other error.

static inline int **regulator_get_voltage**(const struct *device* *dev, int32_t *volt_uv)

Obtain output voltage.

Parameters

- `dev` – Regulator device instance.
- `volt_uv` – **[out]** Where configured output voltage will be stored.

Return values

- `0` – If successful
- `-ENOSYS` – If function is not implemented.
- `-errno` – In case of any other error.

static inline unsigned int **regulator_count_current_limits**(const struct *device* *dev)

Obtain the number of supported current limit levels.

Each current limit level supported by a regulator gets an index, starting from zero. The total number of supported current limit levels can be used together with *regulator_list_current_limit()* to list all supported current limit levels.

Parameters

- `dev` – Regulator device instance.

Returns

Number of supported current limits.

static inline int **regulator_list_current_limit**(const struct *device* *dev, unsigned int idx, int32_t *current_ua)

Obtain the value of a current limit given an index.

Each current limit level supported by a regulator gets an index, starting from zero. Together with *regulator_count_current_limits()*, this function can be used to iterate over all supported current limits.

Parameters

- `dev` – Regulator device instance.
- `idx` – Current index.
- `current_ua` – **[out]** Where current for the given `index` will be stored, in microamps.

Return values

- `0` – If `index` corresponds to a supported current limit.
- `-EINVAL` – If `index` does not correspond to a supported current limit.

int **regulator_set_current_limit**(const struct *device* *dev, int32_t min_ua, int32_t max_ua)

Set output current limit.

The output current limit will be configured to the closest supported output current limit. *regulator_get_current_limit()* can be used to obtain the actual configured current limit. Current may be limited using `current-min-microamp` and/or `current-max-microamp` in Devicetree.

Parameters

- `dev` – Regulator device instance.
- `min_ua` – Minimum acceptable current limit in microamps.
- `max_ua` – Maximum acceptable current limit in microamps.

Return values

- `0` – If successful.
- `-EINVAL` – If the given current limit window is not valid.

- `-ENOSYS` – If function is not implemented.
- `-errno` – In case of any other error.

static inline int **regulator_get_current_limit**(const struct *device* *dev, int32_t *curr_ua)

Get output current limit.

Parameters

- `dev` – Regulator device instance.
- `curr_ua` – **[out]** Where output current limit will be stored.

Return values

- `0` – If successful.
- `-ENOSYS` – If function is not implemented.
- `-errno` – In case of any other error.

int **regulator_set_mode**(const struct *device* *dev, *regulator_mode_t* mode)

Set mode.

Regulators can support multiple modes in order to permit different voltage configuration or better power savings. This API will apply a mode for the regulator. Allowed modes may be limited using `regulator-allowed-modes` devicetree property.

Parameters

- `dev` – Regulator device instance.
- `mode` – Mode to select for this regulator.

Return values

- `0` – If successful.
- `-ENOTSUP` – If mode is not supported.
- `-ENOSYS` – If function is not implemented.
- `-errno` – In case of any other error.

static inline int **regulator_get_mode**(const struct *device* *dev, *regulator_mode_t* *mode)

Get mode.

Parameters

- `dev` – Regulator device instance.
- `mode` – **[out]** Where mode will be stored.

Return values

- `0` – If successful.
- `-ENOSYS` – If function is not implemented.
- `-errno` – In case of any other error.

static inline int **regulator_set_active_discharge**(const struct *device* *dev, bool active_discharge)

Set active discharge setting.

Parameters

- `dev` – Regulator device instance.
- `active_discharge` – Active discharge enable or disable.

Return values

- `0` – If successful.

- `-ENOSYS` – If function is not implemented.
- `-errno` – In case of any other error.

static inline int **regulator_get_active_discharge**(const struct *device* *dev, bool *active_discharge)

Get active discharge setting.

Parameters

- `dev` – Regulator device instance.
- `active_discharge` – **[out]** Where active discharge will be stored.

Return values

- `0` – If successful.
- `-ENOSYS` – If function is not implemented.
- `-errno` – In case of any other error.

static inline int **regulator_get_error_flags**(const struct *device* *dev, *regulator_error_flags_t* *flags)

Get active error flags.

Parameters

- `dev` – Regulator device instance.
- `flags` – **[out]** Where error flags will be stored.

Return values

- `0` – If successful.
- `-ENOSYS` – If function is not implemented.
- `-errno` – In case of any other error.

7.6.41 Reset Controller

Overview

Reset controllers are units that control the reset signals to multiple peripherals. The reset controller API allows peripheral drivers to request control over their reset input signals, including the ability to assert, deassert and toggle those signals. Also, the reset status of the reset input signal can be checked.

Mainly, the line_assert and line_deassert API functions are optional because in most cases we want to toggle the reset signals.

Configuration Options

Related configuration options:

- `CONFIG_RESET`

API Reference

group `reset_controller_interface`

Reset Controller Interface.

Since
3.1

Version
0.2.0

Defines

`RESET_DT_SPEC_GET_BY_IDX`(node_id, idx)

Static initializer for a *reset_dt_spec*.

This returns a static initializer for a *reset_dt_spec* structure given a devicetree node identifier, a property specifying a Reset Controller and an index.

Example devicetree fragment:

```
n: node {
   resets = <&reset 10>;
}
```

Example usage:

```
const struct reset_dt_spec spec = RESET_DT_SPEC_GET_BY_IDX(DT_NODELABEL(n), 0);
// Initializes 'spec' to:
// {
//         .dev = DEVICE_DT_GET(DT_NODELABEL(reset)),
//         .id = 10
// }
```

The 'reset' field must still be checked for readiness, e.g. using *device_is_ready()*. It is an error to use this macro unless the node exists, has the given property, and that property specifies a reset controller reset line id as shown above.

Parameters

- `node_id` – devicetree node identifier
- `idx` – logical index into "resets"

Returns
static initializer for a struct *reset_dt_spec* for the property

`RESET_DT_SPEC_GET_BY_IDX_OR`(node_id, idx, default_value)

Like *RESET_DT_SPEC_GET_BY_IDX()*, with a fallback to a default value.

If the devicetree node identifier 'node_id' refers to a node with a 'resets' property, this expands to *RESET_DT_SPEC_GET_BY_IDX(node_id, idx)*. The `default_value` parameter is not expanded in this case.

Otherwise, this expands to `default_value`.

Parameters

- `node_id` – devicetree node identifier
- `idx` – logical index into the 'resets' property
- `default_value` – fallback value to expand to

Returns
static initializer for a struct *reset_dt_spec* for the property, or default_value if the node or property do not exist

`RESET_DT_SPEC_GET`(node_id)

Equivalent to *RESET_DT_SPEC_GET_BY_IDX(node_id, 0)*.

See also

RESET_DT_SPEC_GET_BY_IDX()

Parameters

- `node_id` – devicetree node identifier

Returns
static initializer for a struct *reset_dt_spec* for the property

`RESET_DT_SPEC_GET_OR`(node_id, default_value)

Equivalent to *RESET_DT_SPEC_GET_BY_IDX_OR(node_id, 0, default_value)*.

Parameters

- `node_id` – devicetree node identifier
- `default_value` – fallback value to expand to

Returns
static initializer for a struct *reset_dt_spec* for the property, or default_value if the node or property do not exist

`RESET_DT_SPEC_INST_GET_BY_IDX`(inst, idx)

Static initializer for a *reset_dt_spec* from a DT_DRV_COMPAT instance's Reset Controller property at an index.

See also

RESET_DT_SPEC_GET_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `idx` – logical index into "resets"

Returns
static initializer for a struct *reset_dt_spec* for the property

`RESET_DT_SPEC_INST_GET_BY_IDX_OR`(inst, idx, default_value)

Static initializer for a *reset_dt_spec* from a DT_DRV_COMPAT instance's 'resets' property at an index, with fallback.

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `idx` – logical index into the 'resets' property
- `default_value` – fallback value to expand to

Returns
static initializer for a struct *reset_dt_spec* for the property, or default_value if the node or property do not exist

`RESET_DT_SPEC_INST_GET`(inst)
Equivalent to *RESET_DT_SPEC_INST_GET_BY_IDX(inst, 0)*.

See also

RESET_DT_SPEC_INST_GET_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number

Returns
static initializer for a struct *reset_dt_spec* for the property

`RESET_DT_SPEC_INST_GET_OR`(inst, default_value)
Equivalent to *RESET_DT_SPEC_INST_GET_BY_IDX_OR(node_id, 0, default_value)*.

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `default_value` – fallback value to expand to

Returns
static initializer for a struct *reset_dt_spec* for the property, or default_value if the node or property do not exist

Functions

int `reset_status`(const struct *device* *dev, uint32_t id, uint8_t *status)
Get the reset status.

This function returns the reset status of the device.

Parameters

- `dev` – Reset controller device.
- `id` – Reset line.
- `status` – Where to write the reset status.

Return values

- `0` – On success.
- `-ENOSYS` – If the functionality is not implemented by the driver.
- `-errno` – Other negative errno in case of failure.

static inline int `reset_status_dt`(const struct *reset_dt_spec* *spec, uint8_t *status)
Get the reset status from a *reset_dt_spec*.

This is equivalent to:

reset_status(spec->dev, spec->id, status);

Parameters

- `spec` – Reset controller specification from devicetree

- **status** – Where to write the reset status.

Returns
a value from *reset_status()*

int **reset_line_assert**(const struct *device* *dev, uint32_t id)

Put the device in reset state.

This function sets/clears the reset bits of the device, depending on the logic level (active-high/active-low).

Parameters

- **dev** – Reset controller device.
- **id** – Reset line.

Return values

- **0** – On success.
- **-ENOSYS** – If the functionality is not implemented by the driver.
- **-errno** – Other negative errno in case of failure.

static inline int **reset_line_assert_dt**(const struct *reset_dt_spec* *spec)

Assert the reset state from a *reset_dt_spec*.

This is equivalent to:

```
reset_line_assert(spec->dev, spec->id);
```

Parameters

- **spec** – Reset controller specification from devicetree

Returns
a value from *reset_line_assert()*

int **reset_line_deassert**(const struct *device* *dev, uint32_t id)

Take out the device from reset state.

This function sets/clears the reset bits of the device, depending on the logic level (active-low/active-high).

Parameters

- **dev** – Reset controller device.
- **id** – Reset line.

Return values

- **0** – On success.
- **-ENOSYS** – If the functionality is not implemented by the driver.
- **-errno** – Other negative errno in case of failure.

static inline int **reset_line_deassert_dt**(const struct *reset_dt_spec* *spec)

Deassert the reset state from a *reset_dt_spec*.

This is equivalent to:

```
reset_line_deassert(spec->dev, spec->id)
```

Parameters

- **spec** – Reset controller specification from devicetree

Returns
a value from *reset_line_deassert()*

int **reset_line_toggle**(const struct *device* *dev, uint32_t id)

Reset the device.

This function performs reset for a device (assert + deassert).

Parameters

- **dev** – Reset controller device.
- **id** – Reset line.

Return values

- **0** – On success.
- **-ENOSYS** – If the functionality is not implemented by the driver.
- **-errno** – Other negative errno in case of failure.

static inline int **reset_line_toggle_dt**(const struct *reset_dt_spec* *spec)

Reset the device from a *reset_dt_spec*.

This is equivalent to:

```
reset_line_toggle(spec->dev, spec->id)
```

Parameters

- **spec** – Reset controller specification from devicetree

Returns
a value from *reset_line_toggle()*

struct **reset_dt_spec**

#include <*reset.h*> Reset controller device configuration.

Public Members

const struct *device* ***dev**

Reset controller device.

uint32_t **id**

Reset line.

7.6.42 Retained Memory

Overview

The retained memory driver API provides a way of reading from/writing to memory areas whereby the contents of the memory is retained whilst the device is powered (data may be lost in low power modes).

Configuration Options

Related configuration options:

- CONFIG_RETAINED_MEM
- CONFIG_RETAINED_MEM_INIT_PRIORITY
- CONFIG_RETAINED_MEM_MUTEX_FORCE_DISABLE

Mutex protection

Mutex protection of retained memory drivers is enabled by default when applications are compiled with multithreading support. This means that different threads can safely call the retained memory functions without clashing with other concurrent thread function usage, but means that retained memory functions cannot be used from ISRs. It is possible to disable mutex protection globally on all retained memory drivers by enabling CONFIG_RETAINED_MEM_MUTEX_FORCE_DISABLE - users are then responsible for ensuring that the function calls do not conflict with each other.

API Reference

group `retained_mem_interface`

Retained memory driver interface.

Since
3.4

Version
0.8.0

Typedefs

typedef ssize_t (***retained_mem_size_api**)(const struct *device* *dev)

Callback API to get size of retained memory area.

See *retained_mem_size()* for argument description.

typedef int (***retained_mem_read_api**)(const struct *device* *dev, off_t offset, uint8_t *buffer, size_t size)

Callback API to read from retained memory area.

See *retained_mem_read()* for argument description.

typedef int (***retained_mem_write_api**)(const struct *device* *dev, off_t offset, const uint8_t *buffer, size_t size)

Callback API to write to retained memory area.

See *retained_mem_write()* for argument description.

typedef int (***retained_mem_clear_api**)(const struct *device* *dev)

Callback API to clear retained memory area (reset all data to 0x00).

See *retained_mem_clear()* for argument description.

Functions

ssize_t **retained_mem_size**(const struct *device* *dev)

Returns the size of the retained memory area.

Parameters

- **dev** – Retained memory device to use.

Return values

Positive – value indicating size in bytes on success, else negative errno code.

int **retained_mem_read**(const struct *device* *dev, off_t offset, uint8_t *buffer, size_t size)

Reads data from the Retained memory area.

Parameters

- **dev** – Retained memory device to use.
- **offset** – Offset to read data from.
- **buffer** – Buffer to store read data in.
- **size** – Size of data to read.

Return values

0 – on success else negative errno code.

int **retained_mem_write**(const struct *device* *dev, off_t offset, const uint8_t *buffer, size_t size)

Writes data to the Retained memory area - underlying data does not need to be cleared prior to writing.

Parameters

- **dev** – Retained memory device to use.
- **offset** – Offset to write data to.
- **buffer** – Data to write.
- **size** – Size of data to be written.

Return values

0 – on success else negative errno code.

int **retained_mem_clear**(const struct *device* *dev)

Clears data in the retained memory area by setting it to 0x00.

Parameters

- **dev** – Retained memory device to use.

Return values

0 – on success else negative errno code.

struct **retained_mem_driver_api**

#include <retained_mem.h> Retained memory driver API API which can be used by a device to store data in a retained memory area.

Retained memory is memory that is retained while the device is powered but is lost when power to the device is lost (note that low power modes in some devices may clear the data also). This may be in a non-initialised RAM region, or in specific registers, but is not reset when a different application begins execution or the device is rebooted (without power loss). It must support byte-level reading and writing without a need to erase data before writing.

Note that drivers must implement all functions, none of the functions are optional.

7.6.43 Secure Digital High Capacity (SDHC)

The SDHC api offers a generic interface for interacting with an SD host controller device. It is used by the SD subsystem, and is not intended to be directly used by the application

Basic Operation

SD Host Controller An SD host controller is a device capable of sending SD commands to an attached SD card. These commands can be sent using the native SD protocol, or over SPI. Some SD host controllers are also capable of communicating with MMC devices. The SDHC api is designed to provide a generic way to send commands to and interact with attached SD devices.

Requests The core of the SDHC api is the *sdhc_request()* api. Requests contain a *sdhc_command* command structure, and an optional *sdhc_data* data structure. The caller may check the return code, or the `response` field of the SD command structure to determine if the SDHC request succeeded. The data structure allows the caller to specify a number of blocks to transfer, and a buffer location to read or write them from. Whether the provided buffer is used for sending or reading data depends on the command opcode provided.

Host Controller I/O The *sdhc_set_io()* api allows the user to change I/O settings of the SD host controller, such as clock frequency, I/O voltage, and card power. Not all controllers will support applying all I/O settings. For example, SPI mode controllers typically cannot toggle power to the SD card.

Related configuration options:

- `CONFIG_SDHC`

API Reference

group `sdhc_interface`

SDHC interface.

Since
3.1

Version
0.1.0

SD command timeouts

`SDHC_TIMEOUT_FOREVER`

Defines

SDHC_NATIVE_RESPONSE_MASK

SDHC_SPI_RESPONSE_TYPE_MASK

Typedefs

typedef void (*sdhc_interrupt_cb_t)(const struct *device* *dev, int reason, const void *user_data)

SDHC card interrupt callback prototype.

Function prototype for SDHC card interrupt callback.

Param dev
SDHC device that produced interrupt

Param reason
one of *sdhc_interrupt_source* values.

Param user_data
User data, set via *sdhc_enable_interrupt*

Enums

enum sdhc_bus_mode

SD bus mode.

Most controllers will use push/pull, including spi, but SDHC controllers that implement SD host specification can support open drain mode

Values:

enumerator SDHC_BUSMODE_OPENDRAIN = 1

enumerator SDHC_BUSMODE_PUSHPULL = 2

enum sdhc_power

SD host controller power.

Many host controllers can control power to attached SD cards. This enum allows applications to request the host controller power off the SD card.

Values:

enumerator SDHC_POWER_OFF = 1

enumerator SDHC_POWER_ON = 2

enum sdhc_bus_width

SD host controller bus width.

Only relevant in SD mode, SPI does not support bus width. UHS cards will use 4 bit data bus, all cards start in 1 bit mode

Values:

enumerator SDHC_BUS_WIDTH1BIT = 1U

enumerator SDHC_BUS_WIDTH4BIT = 4U

enumerator SDHC_BUS_WIDTH8BIT = 8U

enum sdhc_timing_mode

SD host controller timing mode.

Used by SD host controller to determine the timing of the cards attached to the bus. Cards start with legacy timing, but UHS-II cards can go up to SDR104.

Values:

enumerator SDHC_TIMING_LEGACY = 1U

Legacy 3.3V Mode.

enumerator SDHC_TIMING_HS = 2U

Legacy High speed mode (3.3V)

enumerator SDHC_TIMING_SDR12 = 3U

Identification mode & SDR12.

enumerator SDHC_TIMING_SDR25 = 4U

High speed mode & SDR25.

enumerator SDHC_TIMING_SDR50 = 5U

SDR49 mode.

enumerator SDHC_TIMING_SDR104 = 6U

SDR104 mode.

enumerator SDHC_TIMING_DDR50 = 7U

DDR50 mode.

enumerator SDHC_TIMING_DDR52 = 8U

DDR52 mode.

enumerator SDHC_TIMING_HS200 = 9U

HS200 mode.

enumerator SDHC_TIMING_HS400 = 10U

HS400 mode.

enum `sd_voltage`

SD voltage.

UHS cards can run with 1.8V signalling for improved power consumption. Legacy cards may support 3.0V signalling, and all cards start at 3.3V. Only relevant for SD controllers, not SPI ones.

Values:

enumerator `SD_VOL_3_3_V` = 1U

card operation voltage around 3.3v

enumerator `SD_VOL_3_0_V` = 2U

card operation voltage around 3.0v

enumerator `SD_VOL_1_8_V` = 3U

card operation voltage around 1.8v

enumerator `SD_VOL_1_2_V` = 4U

card operation voltage around 1.2v

enum `sdhc_interrupt_source`

SD host controller interrupt sources.

Interrupt sources for SD host controller.

Values:

enumerator `SDHC_INT_SDIO` = *BIT*(0)

Card interrupt, used by SDIO cards.

enumerator `SDHC_INT_INSERTED` = *BIT*(1)

Card was inserted into slot.

enumerator `SDHC_INT_REMOVED` = *BIT*(2)

Card was removed from slot.

Functions

int `sdhc_hw_reset`(const struct *device* *dev)

reset SDHC controller state

Used when the SDHC has encountered an error. Resetting the SDHC controller should clear all errors on the SDHC, but does not necessarily reset I/O settings to boot (this can be done with *sdhc_set_io*)

Parameters

- `dev` – SD host controller device

Return values

- `0` – reset succeeded
- `-ETIMEDOUT` – controller reset timed out
- `-EIO` – reset failed

int **sdhc_request**(const struct *device* *dev, struct *sdhc_command* *cmd, struct *sdhc_data* *data)

Send command to SDHC.

Sends a command to the SD host controller, which will send this command to attached SD cards.

Parameters

- `dev` – SDHC device
- `cmd` – SDHC command
- `data` – SDHC data. Leave NULL to send SD command without data.

Return values

- `0` – command was sent successfully
- `-ETIMEDOUT` – command timed out while sending
- `-ENOTSUP` – host controller does not support command
- `-EIO` – I/O error

int **sdhc_set_io**(const struct *device* *dev, struct *sdhc_io* *io)

set I/O properties of SDHC

I/O properties should be reconfigured when the card has been sent a command to change its own SD settings. This function can also be used to toggle power to the SD card.

Parameters

- `dev` – SDHC device
- `io` – I/O properties

Returns
0 I/O was configured correctly

Returns
-ENOTSUP controller does not support these I/O settings

Returns
-EIO controller could not configure I/O settings

int **sdhc_card_present**(const struct *device* *dev)

check for SDHC card presence

Checks if card is present on the SD bus. Note that if a controller requires cards be powered up to detect presence, it should do so in this function.

Parameters

- `dev` – SDHC device

Return values

- `1` – card is present
- `0` – card is not present
- `-EIO` – I/O error

int **sdhc_execute_tuning**(const struct *device* *dev)

run SDHC tuning

SD cards require signal tuning for UHS modes SDR104 and SDR50. This function allows an application to request the SD host controller to tune the card.

Parameters

- `dev` – SDHC device

Return values

- `0` – tuning succeeded, card is ready for commands
- `-ETIMEDOUT` – tuning failed after timeout
- `-ENOTSUP` – controller does not support tuning
- `-EIO` – I/O error while tuning

int `sdhc_card_busy`(const struct *device* *dev)

check if SD card is busy

This check should generally be implemented as checking the line level of the DAT[0:3] lines of the SD bus. No SD commands need to be sent, the controller simply needs to report the status of the SD bus.

Parameters

- `dev` – SDHC device

Return values

- `0` – card is not busy
- `1` – card is busy
- `-EIO` – I/O error

int `sdhc_get_host_props`(const struct *device* *dev, struct *sdhc_host_props* *props)

Get SD host controller properties.

Gets host properties from the host controller. Host controller should initialize all values in the *sdhc_host_props* structure provided.

Parameters

- `dev` – SDHC device
- `props` – property structure to be filled by sdhc driver

Return values

- `0` – function succeeded.
- `-ENOTSUP` – host controller does not support this call

int `sdhc_enable_interrupt`(const struct *device* *dev, *sdhc_interrupt_cb_t* callback, int sources, void *user_data)

Enable SDHC interrupt sources.

Enables SDHC interrupt sources. Each subsequent call of this function should replace the previous callback set, and leave only the interrupts specified in the “sources” argument enabled.

Parameters

- `dev` – SDHC device
- `callback` – Callback called when interrupt occurs
- `sources` – bitmask of *sdhc_interrupt_source* values indicating which interrupts should produce a callback
- `user_data` – parameter that will be passed to callback function

Return values

- `0` – interrupts were enabled, and callback was installed
- `-ENOTSUP` – controller does not support this function

- `-EIO` – I/O error

int `sdhc_disable_interrupt`(const struct *device* *dev, int sources)

Disable SDHC interrupt sources.

Disables SDHC interrupt sources. If multiple sources are enabled, only the ones specified in “sources” will be masked.

Parameters

- `dev` – SDHC device
- `sources` – bitmask of *sdhc_interrupt_source* values indicating which interrupts should be disabled.

Return values

- `0` – interrupts were disabled
- `-ENOTSUP` – controller does not support this function
- `-EIO` – I/O error

struct `sdhc_command`

#include <*sdhc.h*> SD host controller command structure.

This command structure is used to send command requests to an SD host controller, which will be sent to SD devices.

Public Members

uint32_t `opcode`

SD Host specification CMD index.

uint32_t `arg`

SD host specification argument.

uint32_t `response`[4]

SD card response field.

uint32_t `response_type`

Expected SD response type.

unsigned int `retries`

Max number of retries.

int `timeout_ms`

Command timeout in milliseconds.

struct `sdhc_data`

#include <*sdhc.h*> SD host controller data structure.

This command structure is used to send data transfer requests to an SD host controller, which will be sent to SD devices.

Public Members

unsigned int **block_addr**
Block to start read from.

unsigned int **block_size**
Block size.

unsigned int **blocks**
Number of blocks.

unsigned int **bytes_xfered**
populated with number of bytes sent by SDHC

void ***data**
Data to transfer or receive.

int **timeout_ms**
data timeout in milliseconds

struct **sdhc_host_caps**
#include <*sdhc.h*> SD host controller capabilities.

SD host controller capability flags. These flags should be set by the SDHC driver, using the *sdhc_get_host_props* api.

Public Members

unsigned int **timeout_clk_freq**
Timeout clock frequency.

unsigned int **timeout_clk_unit**
Timeout clock unit.

unsigned int **sd_base_clk**
SD base clock frequency.

unsigned int **max_blk_len**
Max block length.

unsigned int **bus_8_bit_support**
8-bit Support for embedded device

unsigned int **bus_4_bit_support**
4 bit bus support

unsigned int **adma_2_support**
ADMA2 support.

unsigned int **high_spd_support**
High speed support.

unsigned int **sdma_support**
SDMA support.

unsigned int **suspend_res_support**
Suspend/Resume support.

unsigned int **vol_330_support**
Voltage support 3.3V.

unsigned int **vol_300_support**
Voltage support 3.0V.

unsigned int **vol_180_support**
Voltage support 1.8V.

unsigned int **address_64_bit_support_v4**
64-bit system address support for V4

unsigned int **address_64_bit_support_v3**
64-bit system address support for V3

unsigned int **sdio_async_interrupt_support**
Asynchronous interrupt support.

unsigned int **slot_type**
Slot type.

unsigned int **sdr50_support**
SDR50 support.

unsigned int **sdr104_support**
SDR104 support.

unsigned int **ddr50_support**
DDR50 support.

unsigned int **uhs_2_support**
UHS-II support.

unsigned int **drv_type_a_support**
Driver type A support.

unsigned int **drv_type_c_support**
Driver type C support.

unsigned int **drv_type_d_support**
Driver type D support.

unsigned int **retune_timer_count**
Timer count for re-tuning.

unsigned int **sdr50_needs_tuning**
Use tuning for SDR50.

unsigned int **retuning_mode**
Re-tuning mode.

unsigned int **clk_multiplier**
Clock multiplier.

unsigned int **adma3_support**
ADMA3 support.

unsigned int **vdd2_180_support**
1.8V VDD2 support

unsigned int **hs200_support**
HS200 support.

unsigned int **hs400_support**
HS400 support.

struct **sdhc_io**
#include <*sdhc.h*> SD host controller I/O control structure.

Controls I/O settings for the SDHC. Note that only a subset of these settings apply to host controllers in SPI mode. Populate this struct, then call *sdhc_set_io* to apply I/O settings

Public Members

enum sdhc_clock_speed **clock**
Clock rate.

enum *sdhc_bus_mode* **bus_mode**
command output mode

enum *sdhc_power* **power_mode**
SD power supply mode.

enum *sdhc_bus_width* **bus_width**
SD bus width.

enum *sdhc_timing_mode* **timing**
SD bus timing.

enum sd_driver_type **driver_type**
SD driver type.

enum *sd_voltage* **signal_voltage**
IO signalling voltage (usually 1.8 or 3.3V)

struct **sdhc_host_props**
#include <*sdhc.h*> SD host controller properties.

Populated by the host controller using *sdhc_get_host_props* api.

Public Members

unsigned int **f_max**
Max bus frequency.

unsigned int **f_min**
Min bus frequency.

unsigned int **power_delay**
Delay to allow SD to power up or down (in ms)

struct *sdhc_host_caps* **host_caps**
Host capability bitfield.

uint32_t **max_current_330**
Max current (in mA) at 3.3V.

uint32_t **max_current_300**
Max current (in mA) at 3.0V.

uint32_t **max_current_180**
Max current (in mA) at 1.8V.

bool **is_spi**
Is the host using SPI mode.

struct **sdhc_driver_api**
#include <*sdhc.h*>

7.6.44 Sensors

The sensor driver API provides functionality to uniformly read, configure, and setup event handling for devices that take real world measurements in meaningful units.

Sensors range from very simple temperature-reading devices that must be polled with a fixed scale to complex devices taking in readings from multitudes of sensors and themselves producing new inferred sensor data such as step counts, presence detection, orientation, and more.

Supporting this wide breadth of devices is a demanding task and the sensor API attempts to provide a uniform interface to them.

Using Sensors

Using sensors from an application there are some APIs and terms that are helpful to understand. Sensors in Zephyr are composed of *Sensor Channels*, *Sensor Attributes*, and *Sensor Triggers*. Attributes and triggers may be device or channel specific.

> **Note**
>
> Getting samples from sensors using the sensor API today can be done in one of two ways. A stable and long-lived API *Fetch and Get*, or a newer but rapidly stabilizing API *Read and Decode*. It's expected that in the near future *Fetch and Get* will be deprecated in favor of *Read and Decode*. Triggers are handled entirely differently for *Fetch and Get* or *Read and Decode* and the differences are noted in each of those sections.

Sensor Attributes *Attributes*, enumerated in *sensor_attribute*, are immutable and mutable properties of a sensor and its channels.

Attributes allow for obtaining metadata and changing configuration of a sensor. Common configuration parameters like channel scale, sampling frequency, adjusting channel offsets, signal filtering, power modes, on chip buffers, and event handling options are very common. Attributes provide a flexible API for inspecting and manipulating such device properties.

Attributes are specified using *sensor_attribute* which can be used with *sensor_attr_get()* and *sensor_attr_set()* to get and set a sensors attributes.

A quick example...

```
const struct device *accel_dev = DEVICE_DT_GET(DT_ALIAS(accel0));
struct sensor_value accel_sample_rate;
int rc;

rc = sensor_attr_get(accel_dev, SENSOR_CHAN_ACCEL_XYZ, SENSOR_ATTR_SAMPLING_FREQUENCY, &
↪accel_sample_rate);
if (rc != 0) {
            printk("Failed to get sampling frequency\n");
}

printk("Sample rate for accel %p is %d.06%d\n", accel_dev, accel_sample_rate.val1, accel_
↪sample_rate.val2*1000000);

accel_sample_rate.val1 = 2000;

rc = sensor_attr_set(accel_dev, SENSOR_CHAN_ACCEL_XYZ, SENSOR_ATTR_SAMPLING_FREQUENCY,␣
↪accel_sample_rate);
if (rc != 0) {
            printk("Failed to set sampling frequency\n");
}
```

Sensor Channels *Channels*, enumerated in *sensor_channel*, are quantities that a sensor device can measure.

Sensors may have multiple channels, either to represent different axes of the same physical property (e.g. acceleration); or because they can measure different properties altogether (ambient temperature, pressure and humidity). Sensors may have multiple channels of the same measurement type to enable measuring many readings of perhaps temperature, light intensity, amperage, voltage, or capacitance for example.

A channel is specified in Zephyr using a *sensor_chan_spec* which is a pair containing both the channel type (*sensor_channel*) and channel index. At times only *sensor_channel* is used but this

should be considered historical since the introduction of *sensor_chan_spec* for Zephyr 3.7.

Sensor Triggers *Triggers*, enumerated in *sensor_trigger_type*, are sensor generated events. Typically sensors allow setting up these events to cause digital line signaling for easy capture by a micro controller. The events can then commonly be inspected by reading registers to determine which event caused the digital line signaling to occur.

There are many kinds of triggers sensors provide, from informative ones such as data ready to physical events such as taps or steps.

Power Management Power management of sensors is often a non-trivial task as sensors may have multiple power states for various channels. Some sensors may allow for low noise, low power, or suspending channels potentially saving quite a bit of power at the cost of noise or sampling speed performance. In very low power states sensors may lose their state, turning off even the digital logic portion of the device.

All this is to say that power management of sensors is typically application specific! Often the channel states are mutable using *Sensor Attributes*. While total device suspending and resume can be done using the power management ref counting APIs if the device implements the necessary functionality.

Most likely the API sensors should use for their fully suspended/resume power states is *Device Runtime Power Management* using explicit calls at an application level to *pm_device_runtime_get()* and *pm_device_runtime_put()*.

In the future, with *Read and Decode* its possible that automatic management of device power management would be possible in the streaming case as the application informs the driver of usage at all times through requests to read on given events.

Device Tree In the context of sensors device tree provides the initial hardware configuration for sensors on a per device level. Each device must specify a device tree binding in Zephyr, and ideally, a set of hardware configuration options for things such as channel power modes, data rates, filters, decimation, and scales. These can then be used in a boards devicetree to configure a sensor to its initial state.

```
#include <zephyr/dt-bindings/icm42688.h>

&spi0 {
    /* SPI bus options here, not shown */

    accel_gyro0: icm42688p@0 {
        compatible = "invensense,icm42688";
        reg = <0>;
        int-gpios = <&pioc 6 GPIO_ACTIVE_HIGH>; /* SoC specific pin to select for interrupt␣
↪line */
        spi-max-frequency = <DT_FREQ_M(24)>; /* Maximum SPI bus frequency */
        accel-pwr-mode = <ICM42688_ACCEL_LN>; /* Low noise mode */
        accel-odr = <ICM42688_ACCEL_ODR_2000>; /* 2000 Hz sampling */
        accel-fs = <ICM42688_ACCEL_FS_16>; /* 16G scale */
        gyro-pwr-mode = <ICM42688_GYRO_LN>; /* Low noise mode */
        gyro-odr = <ICM42688_GYRO_ODR_2000>; /* 2000 Hz sampling */
        gyro-fs = <ICM42688_GYRO_FS_16>; /* 16G scale */
    };
 };
```

Fetch and Get The stable and long existing APIs for reading sensor data and handling triggers are:

- *sensor_sample_fetch()*
- *sensor_sample_fetch_chan()*
- *sensor_channel_get()*
- *sensor_trigger_set()*

These functions work together. The fetch APIs block the calling context which must be a thread until the requested *sensor_channel* (or all channels) has been obtained and stored into the driver instance's private data.

The channel data most recently fetched can then be obtained as a *sensor_value* by calling *sensor_channel_get()* for each channel type.

Warning

It should be noted that calling fetch and get from multiple contexts without a locking mechanism is undefined and most sensor drivers do not attempt to internally provide exclusive access to the device during or between these calls.

Polling Using fetch and get sensor can be read in a polling manner from software threads.

```
/*
 * Copyright (c) 2016 Intel Corporation
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <zephyr/kernel.h>
#include <zephyr/device.h>
#include <zephyr/drivers/sensor.h>
#include <zephyr/sys/printk.h>
#include <stdio.h>

int main(void)
{
        const struct device *const dev = DEVICE_DT_GET(DT_ALIAS(magn0));
        struct sensor_value value_x, value_y, value_z;
        int ret;

        if (!device_is_ready(dev)) {
                printk("sensor: device not ready.\n");
                return 0;
        }

        printk("Polling magnetometer data from %s.\n", dev->name);

        while (1) {
                ret = sensor_sample_fetch(dev);
                if (ret) {
                        printk("sensor_sample_fetch failed ret %d\n", ret);
                        return 0;
                }

                ret = sensor_channel_get(dev, SENSOR_CHAN_MAGN_X, &value_x);
                ret = sensor_channel_get(dev, SENSOR_CHAN_MAGN_Y, &value_y);
                ret = sensor_channel_get(dev, SENSOR_CHAN_MAGN_Z, &value_z);
                printf("( x y z ) = ( %f  %f  %f )\n",
                       sensor_value_to_double(&value_x),
                       sensor_value_to_double(&value_y),
```

(continues on next page)

(continued from previous page)

```
                        sensor_value_to_double(&value_z));

                k_sleep(K_MSEC(500));
        }
        return 0;
}
```

Triggers Triggers in the stable API require enabling triggers with a device specific Kconfig. The device specific Kconfig typically allows selecting the context the trigger runs. The application then needs to register a callback with a function signature matching *sensor_trigger_handler_t* using *sensor_trigger_set()* for the specific triggers (events) to listen for.

> **Note**
>
> Triggers may not be set from user mode threads, and the callback is not run in a user mode context.

There are typically two options provided for each driver where to run trigger handlers. Either the trigger handler is run using the system work queue thread (*Workqueue Threads*) or a dedicated thread. A great example can be found in the BMI160 driver which has Kconfig options for selecting a trigger mode. See CONFIG_BMI160_TRIGGER_NONE, CONFIG_BMI160_TRIGGER_GLOBAL_THREAD (work queue), CONFIG_BMI160_TRIGGER_OWN_THREAD (dedicated thread).

Some notable attributes of using a driver dedicated thread vs the system work queue.

- Driver dedicated threads have dedicated stack (RAM) which only gets used for that single trigger handler function.
- Driver dedicated threads *do* get their own priority typically which lets you prioritize trigger handling among other threads.
- Driver dedicated threads will not have head of line blocking if the driver needs time to handle the trigger.

> **Note**
>
> In all cases it's very likely there will be variable delays from the actual interrupt to your callback function being run. In the work queue (GLOBAL_THREAD) case the work queue itself can be the source of variable latency!

```
/*
 * Copyright (c) 2024 Intel Corporation
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <zephyr/drivers/sensor.h>


const struct device *const accel0 = DEVICE_DT_GET(DT_ALIAS(accel0));

static struct tap_count_state {
        struct sensor_trigger trig;
        uint32_t count;
} tap_count_state = {
```

(continues on next page)

(continued from previous page)

```
        .trig = {
                .chan = SENSOR_CHAN_ACCEL_XYZ,
                .type = SENSOR_TRIG_TAP,
        },
        .count = 0,
};

void tap_handler(const struct device *dev, const struct sensor_trigger *trig)
{
        struct tap_count_state *state = CONTAINER_OF(trig, struct tap_count_state, trig);

        state->count++;

        printk("Tap! Total Taps: %u\n", state->count);
}

int main(void)
{
        int rc;

        printk("Tap Counter Example (%s)\n", CONFIG_ARCH);

        rc = sensor_trigger_set(accel0, &tap_count_state.trig, tap_handler);

        if (rc != 0) {
                printk("Failed to set trigger handler for taps, error %d\n", rc);
                return rc;
        }

        return rc;
}
```

Read and Decode The quickly stabilizing experimental APIs for reading sensor data are:

- *sensor_read()*
- *sensor_read_async_mempool()*
- *sensor_get_decoder()*
- *sensor_decode()*

Benefits over Fetch and Get These APIs allow for a wider usage of sensors, sensor types, and data flows with sensors. These are the future looking APIs in Zephyr and solve many issues that have been run into with *Fetch and Get*.

sensor_read() and similar functions acquire sensor encoded data into a buffer provided by the caller. Decode (*sensor_decode()*) then decodes the sensor specific encoded data into fixed point *q31_t* values as vectors per channel. This allows further processing using fixed point DSP functions that work on vectors of data to be done (e.g. low-pass filters, FFT, fusion, etc).

Reading is by default asynchronous in its implementation and takes advantage of *Real Time I/O (RTIO)* to enable chaining asynchronous requests, or starting requests against many sensors simultaneously from a single call context.

This enables incredibly useful code flows when working with sensors such as:

- Obtaining the raw sensor data, decoding never, later, or on a separate processor (e.g. a phone).
- Starting a read for sensors directly from an interrupt handler. No dedicated thread needed saving precious stack space. No work queue needed introducing variable latency. Starting a

read for multiple sensors simultaneously from a single call context (interrupt/thread/work queue).

- Requesting multiple reads to the same device for Ping-Pong (double buffering) setups.
- Creating entire pipelines of data flow from sensors allowing for software defined virtual sensors (*Sensing Subsystem*) all from a single thread with DAG process ordering.
- Potentially pre-programming DMAs to trigger on GPIO events, leaving the CPU entirely out of the loop in handling sensor events like FIFO watermarks.

Additionally, other shortcomings of *Fetch and Get* related to memory and trigger handling are solved.

- Triggers result in enqueued events, not callbacks.
- Triggers can be setup to automatically fetch data. Potentially enabling pre-programmed DMA transfers on GPIO interrupts.
- Far less likely triggers are missed due to long held interrupt masks from callbacks and context swapping.
- Sensor FIFOs supported by wiring up FIFO triggers to read data into mempool allocated buffers.
- All sensor processing can be done in user mode (memory protected) threads.
- Multiple sensor channels of the same type are better supported.

Note

For *Read and Decode* benefits to be fully realized requires *Real Time I/O (RTIO)* compliant communication access to the sensor. Typically this means an *Real Time I/O (RTIO)* enabled bus driver for SPI or I2C.

Polling Read Polling reads with *Read and Decode* can be accomplished by instantiating a polling I/O device (akin to a file descriptor) for the sensor with the desired channels to poll. Requesting either blocking or non-blocking reads, then optionally decoding the data into fixed point values.

Polling a temperature sensor and printing its readout is likely the simplest sample to show how this all works.

```
/*
 * Copyright (c) 2024 Intel Corporation
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <zephyr/drivers/sensor.h>

#define TEMP_CHANNEL {SENSOR_CHAN_AMBIENT_TEMP, 0}

const struct device *const temp0 = DEVICE_DT_GET(DT_ALIAS(temp0));

SENSOR_DT_READ_IODEV(temp_iodev, DT_ALIAS(temp0), {TEMP_CHANNEL});
RTIO_DEFINE(temp_ctx, 1, 1);

int main(void)
{
        int rc;
        uint8_t buf[8];
        uint32_t temp_frame_iter = 0;
```

(continues on next page)

(continued from previous page)

```
        struct sensor_q31_data temp_data = {0};
        struct sensor_decode_context temp_decoder = SENSOR_DECODE_CONTEXT_INIT(
                SENSOR_DECODER_DT_GET(DT_ALIAS(temp0)), buf, SENSOR_CHAN_AMBIENT_TEMP, 0);

        while (1) {
                /* Blocking read */
                rc = sensor_read(temp_iodev, &temp_ctx, buf, sizeof(buf));

                if (rc != 0) {
                        printk("sensor_read() failed %d\n", rc);
                }

                /* Decode the data into a single q31 */
                sensor_decode(&temp_decoder, &temp_data, 1);

                printk("Temperature " PRIsensor_q31_data "\n",
                       PRIsensor_q31_data_arg(temp_data, 0));

                k_msleep(1);
        }
}
```

Polling Read with Multiple Sensors One of the benefits of Read and Decode is the ability to concurrently read many sensors with many channels in one thread. Effectively read requests are started asynchronously for all sensors and their channels. When each read completes we then decode the sensor data. Examples speak loudly and so a sample showing how this might work with multiple temperature sensors with multiple temperature channels:

```
/*
 * Copyright (c) 2024 Intel Corporation
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <zephyr/drivers/sensor.h>

#define TEMP_CHANNELS                                   \
        { SENSOR_CHAN_AMBIENT_TEMP, 0 },        \
        { SENSOR_CHAN_AMBIENT_TEMP, 1 }
#define TEMP_ALIAS(id) DT_ALIAS(CONCAT(temp, id))
#define TEMP_IODEV_SYM(id) CONCAT(temp_iodev, id)
#define TEMP_IODEV_PTR(id) &TEMP_IODEV_SYM(id)
#define TEMP_DEFINE_IODEV(id)                           \
        SENSOR_DT_READ_IODEV(                           \
                TEMP_IODEV_SYM(id),                \
                TEMP_ALIAS(id),                       \
                TEMP_CHANNELS                           \
                );

#define NUM_SENSORS 2

LISTIFY(NUM_SENSORS, TEMP_DEFINE_IODEV, (;));

struct sensor_iodev *iodevs[NUM_SENSORS] = { LISTIFY(NUM_SENSORS, TEMP_IODEV_PTR, (,)) };

RTIO_DEFINE_WITH_MEMPOOL(temp_ctx, NUM_SENSORS, NUM_SENSORS, NUM_SENSORS, 8, sizeof(void␣
↪*));

int main(void)
```

(continues on next page)

(continued from previous page)

```
{
        int rc;
        uint32_t temp_frame_iter = 0;
        struct sensor_q31_data temp_data[2] = {0};
        struct sensor_decoder_api *decoder;
        struct rtio_cqe *cqe;
        uint8_t *buf;
        uint32_t buf_len;

        while (1) {
                /* Non-Blocking read for each sensor */
                for (int i = 0; i < NUM_SENSORS; i++) {
                        rc = sensor_read_async_mempool(iodevs[i], &temp_ctx, iodevs[i]);

                        if (rc != 0) {
                                printk("sensor_read() failed %d\n", rc);
                                return;
                        }
                }

                /* Wait for read completions */
                for (int i = 0; i < NUM_SENSORS; i++) {
                        cqe = rtio_cqe_consume_block(&temp_ctx);

                        if (cqe->result != 0) {
                                printk("async read failed %d\n", cqe->result);
                                return;
                        }

                        /* Get the associated mempool buffer with the completion */
                        rc = rtio_cqe_get_mempool_buffer(&temp_ctx, cqe, &buf, &buf_len);

                        if (rc != 0) {
                                printk("get mempool buffer failed %d\n", rc);
                                return;
                        }

                        struct device *sensor = ((struct sensor_read_config *)
                                ((struct rtio_iodev *)cqe->userdata)->data)->sensor;

                        /* Done with the completion event, release it */
                        rtio_cqe_release(&temp_ctx, cqe);

                        rc = sensor_get_decoder(sensor, &decoder);
                        if (rc != 0) {
                                printk("sensor_get_decoder failed %d\n", rc);
                                return;
                        }

                        /* Frame iterators, one per channel we are decoding */
                        uint32_t temp_fits[2] = { 0, 0 };

                        decoder->decode(buf, {SENSOR_CHAN_AMBIENT_TEMP, 0},
                                        &temp_fits[0], 1, &temp_data[0]);
                        decoder->decode(buf, {SENSOR_CHAN_AMBIENT_TEMP, 1},
                                        &temp_fits[1], 1, &temp_data[1]);

                        /* Done with the buffer, release it */
                        rtio_release_buffer(&temp_ctx, buf, buf_len);

                        printk("Temperature for %s channel 0 " PRIsensor_q31_data ",␣
```

(continues on next page)

(continued from previous page)

```
↪channel 1 "
                        PRIsensor_q31_data "\n",
                        dev->name,
                        PRIsensor_q31_data_arg(temp_data[0], 0),
                        PRIsensor_q31_data_arg(temp_data[1], 0));
                }
        }

        k_msleep(1);
}
```

Streaming Handling triggers with *Read and Decode* works by setting up a stream I/O device configuration. A stream specifies the set of triggers to capture and if data should be captured with the event.

```
/*
 * Copyright (c) 2024 Intel Corporation
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <zephyr/drivers/sensor.h>

#define ACCEL_TRIGGERS                                          \
        { SENSOR_TRIG_DRDY, SENSOR_STREAM_DATA_INCLUDE },       \
        { SENSOR_TRIG_TAP, SENSOR_STREAM_DATA_NOP }
#define ACCEL_ALIAS(id) DT_ALIAS(CONCAT(accel, id))
#define ACCEL_IODEV_SYM(id) CONCAT(accel_iodev, id)
#define ACCEL_IODEV_PTR(id) &ACCEL_IODEV_SYM(id)
#define ACCEL_DEFINE_IODEV(id)                                  \
        SENSOR_DT_STREAM_IODEV(                                  \
                ACCEL_IODEV_SYM(id),                            \
                ACCEL_ALIAS(id),                                \
                ACCEL_TRIGGERS                                  \
                );

#define NUM_SENSORS 2

LISTIFY(NUM_SENSORS, ACCEL_DEFINE_IODEV, (;));

struct sensor_iodev *iodevs[NUM_SENSORS] = { LISTIFY(NUM_SENSORS, ACCEL_IODEV_PTR, (,)) };

RTIO_DEFINE_WITH_MEMPOOL(accel_ctx, NUM_SENSORS, NUM_SENSORS, NUM_SENSORS, 16, sizeof(void␣
↪*));

int main(void)
{
        int rc;
        uint32_t accel_frame_iter = 0;
        struct sensor_three_axis_data accel_data[2] = {0};
        struct sensor_decoder_api *decoder;
        struct rtio_cqe *cqe;
        uint8_t *buf;
        uint32_t buf_len;
        struct rtio_sqe *handles[2];

        /* Start the streams */
        for (int i = 0; i < NUM_SENSORS; i++) {
                sensor_stream(iodevs[i], &accel_ctx, NULL, &handles[i]);
```

(continues on next page)

(continued from previous page)

```
        }

        while (1) {
                cqe = rtio_cqe_consume_block(&accel_ctx);

                if (cqe->result != 0) {
                        printk("async read failed %d\n", cqe->result);
                        return;
                }

                rc = rtio_cqe_get_mempool_buffer(&accel_ctx, cqe, &buf, &buf_len);

                if (rc != 0) {
                        printk("get mempool buffer failed %d\n", rc);
                        return;
                }

                struct device *sensor = ((struct sensor_read_config *)
                                         ((struct rtio_iodev *)cqe->userdata)->data)->
↪sensor;

                rtio_cqe_release(&accel_ctx, cqe);

                rc = sensor_get_decoder(sensor, &decoder);

                if (rc != 0) {
                        printk("sensor_get_decoder failed %d\n", rc);
                        return;
                }

                /* Frame iterator values when data comes from a FIFO */
                uint32_t accel_fit = 0;

                /* Number of accelerometer data frames */
                uint32_t frame_count;

                rc = decoder->get_frame_count(buf, {SENSOR_CHAN_ACCEL_XYZ, 0},
                                              &frame_count);
                if (rc != 0) {
                        printk("sensor_get_decoder failed %d\n", rc);
                        return;
                }

                /* If a tap has occurred lets print it out */
                if (decoder->has_trigger(buf, SENSOR_TRIG_TAP)) {
                        printk("Tap! Sensor %s\n", dev->name);
                }

                /* Decode all available accelerometer sample frames */
                for (int i = 0; i < frame_count; i++) {
                        decoder->decode(buf, {SENSOR_CHAN_ACCEL_XYZ, 0},
                                        accel_fit, 1, &accel_data);
                        printk("Accel data for %s " PRIsensor_three_axis_data "\n",
                               dev->name,
                               PRIsensor_three_axis_data_arg(accel_data, 0));
                }

                rtio_release_buffer(&accel_ctx, buf, buf_len);
        }
}
```

Implementing Sensor Drivers

> **Note**
>
> Implementing the driver side of the sensor API requires an understanding how the sensor APIs are used. Please read through *Using Sensors* first!

Implementing Attributes

- SHOULD implement attribute setting in a blocking manner.
- SHOULD provide the ability to get and set channel scale if supported by the device.
- SHOULD provide the ability to get and set channel sampling rate if supported by the device.

Implementing Fetch and Get

- SHOULD implement *sensor_sample_fetch_t* as a blocking call that stashes the specified channels (or all sensor channels) as driver instance data.
- SHOULD implement *sensor_channel_get_t* without side effects manipulating driver state returning stashed sensor readings.
- SHOULD implement *sensor_trigger_set_t* storing the address of the *sensor_trigger* rather than copying the contents. This is so *CONTAINER_OF* may be used for trigger callback context.

Implementing Read and Decode

- MUST implement *sensor_submit_t* as a non-blocking call.
- SHOULD implement *sensor_submit_t* using *Real Time I/O (RTIO)* to do non-blocking bus transfers if possible.
- MAY implement *sensor_submit_t* using a work queue if *Real Time I/O (RTIO)* is unsupported by the bus.
- SHOULD implement *sensor_submit_t* checking the *rtio_sqe* is of type `RTIO_SQE_RX` (read request).
- SHOULD implement *sensor_submit_t* checking all requested channels supported or respond with an error if not.
- SHOULD implement *sensor_submit_t* checking the provided buffer is large enough for the requested channels.
- SHOULD implement *sensor_submit_t* in a way that directly reads into the provided buffer avoiding copying of any kind, with few exceptions.
- MUST implement *sensor_decoder_api* with pure stateless functions. All state needed to convert the raw sensor readings into fixed point SI united values must be in the provided buffer.
- MUST implement *sensor_get_decoder_t* returning the *sensor_decoder_api* for that device type.

API Reference

Related code samples

LVGL line chart with accelerometer data
Display acceleration data on a real-time chart using LVGL.

Secure MQTT Sensor/Actuator
Implement an MQTT-based IoT sensor/actuator device

X-NUCLEO-53L0A1 shield
Interact with the 7-segment display and VL53L0X ranging sensor of an X-NUCLEO-53L0A1 shield.

X-NUCLEO-IKS01A1 shield
Interact with all the sensors of an X-NUCLEO-IKS01A1 shield.

X-NUCLEO-IKS01A2 shield - SensorHub (Mode 2)
Interact with all the sensors of an X-NUCLEO-IKS01A2 shield using Sensor Hub mode.

X-NUCLEO-IKS01A2 shield - Standard (Mode 1)
Interact with all the sensors of an X-NUCLEO-IKS01A2 shield using Standard Mode.

X-NUCLEO-IKS01A3 shield - SensorHub (Mode 2)
Interact with all the sensors of an X-NUCLEO-IKS01A3 shield using Sensor Hub mode.

X-NUCLEO-IKS01A3 shield - Standard (Mode 1)
Interact with all the sensors of an X-NUCLEO-IKS01A3 shield using Standard mode.

X-NUCLEO-IKS02A1 shield - SensorHub (Mode 2)
Interact with all the sensors of an X-NUCLEO-IKS02A1 shield using Sensor Hub mode.

X-NUCLEO-IKS02A1 shield - Standard (Mode 1)
Interact with all the sensors of an X-NUCLEO-IKS02A1 shield using Standard mode.

group `sensor_interface`

Sensor Interface.

Since
1.2

Version
1.0.0

Defines

`SENSOR_DECODE_CONTEXT_INIT`(decoder_, buffer_, channel_type_, channel_index_)
Initialize a *sensor_decode_context*.

`SENSOR_STREAM_TRIGGER_PREP`(_trigger, _opt)

`SENSOR_DT_READ_IODEV`(name, dt_node, ...)
Define a reading instance of a sensor.

Use this macro to generate a *rtio_iodev* for reading specific channels. Example:

```
 (.c)
SENSOR_DT_READ_IODEV(icm42688_accelgyro, DT_NODELABEL(icm42688),
    { SENSOR_CHAN_ACCEL_XYZ, 0 },
```

(continues on next page)

(continued from previous page)

```
    { SENSOR_CHAN_GYRO_XYZ, 0 });

int main(void) {
  sensor_read_async_mempool(&icm42688_accelgyro, &rtio);
}
```

SENSOR_DT_STREAM_IODEV(name, dt_node, ...)

Define a stream instance of a sensor.

Use this macro to generate a *rtio_iodev* for starting a stream that's triggered by specific interrupts. Example:

```
 (.c)
SENSOR_DT_STREAM_IODEV(imu_stream, DT_ALIAS(imu),
    {SENSOR_TRIG_FIFO_WATERMARK, SENSOR_STREAM_DATA_INCLUDE},
    {SENSOR_TRIG_FIFO_FULL, SENSOR_STREAM_DATA_NOP});

int main(void) {
  struct rtio_sqe *handle;
  sensor_stream(&imu_stream, &rtio, NULL, &handle);
  k_msleep(1000);
  rtio_sqe_cancel(handle);
}
```

SENSOR_CHANNEL_3_AXIS(chan)

checks if a given channel is a 3-axis channel

Parameters

- chan – **[in]** The channel to check

Return values

- true – if chan is any of *SENSOR_CHAN_ACCEL_XYZ*, *SENSOR_CHAN_GYRO_XYZ*, or *SENSOR_CHAN_MAGN_XYZ*, or *SENSOR_CHAN_POS_DXYZ*
- false – otherwise

SENSOR_G

The value of gravitational constant in micro m/s^2.

SENSOR_PI

The value of constant PI in micros.

SENSOR_INFO_DEFINE(name, ...)

SENSOR_INFO_DT_DEFINE(node_id)

SENSOR_DEVICE_DT_DEFINE(node_id, init_fn, pm_device, data_ptr, cfg_ptr, level, prio, api_ptr, ...)

Like *DEVICE_DT_DEFINE()* with sensor specifics.

Defines a device which implements the sensor API. May define an element in the sensor info iterable section used to enumerate all sensor devices.

Parameters

- node_id – The devicetree node identifier.
- init_fn – Name of the init function of the driver.

- **pm_device** – PM device resources reference (NULL if device does not use PM).
- **data_ptr** – Pointer to the device's private data.
- **cfg_ptr** – The address to the structure containing the configuration information for this instance of the driver.
- **level** – The initialization level. See SYS_INIT() for details.
- **prio** – Priority within the selected initialization level. See SYS_INIT() for details.
- **api_ptr** – Provides an initial pointer to the API function struct used by the driver. Can be NULL.

SENSOR_DEVICE_DT_INST_DEFINE(inst, ...)

Like *SENSOR_DEVICE_DT_DEFINE()* for an instance of a DT_DRV_COMPAT compatible.

Parameters

- **inst** – instance number. This is replaced by `DT_DRV_COMPAT(inst)` in the call to *SENSOR_DEVICE_DT_DEFINE()*.
- **...** – other parameters as expected by *SENSOR_DEVICE_DT_DEFINE()*.

Typedefs

typedef void (***sensor_trigger_handler_t**)(const struct *device* *dev, const struct *sensor_trigger* *trigger)

Callback API upon firing of a trigger.

Param dev
Pointer to the sensor device

Param trigger
The trigger

typedef int (***sensor_attr_set_t**)(const struct *device* *dev, enum *sensor_channel* chan, enum *sensor_attribute* attr, const struct *sensor_value* *val)

Callback API upon setting a sensor's attributes.

See *sensor_attr_set()* for argument description

typedef int (***sensor_attr_get_t**)(const struct *device* *dev, enum *sensor_channel* chan, enum *sensor_attribute* attr, struct *sensor_value* *val)

Callback API upon getting a sensor's attributes.

See *sensor_attr_get()* for argument description

typedef int (***sensor_trigger_set_t**)(const struct *device* *dev, const struct *sensor_trigger* *trig, *sensor_trigger_handler_t* handler)

Callback API for setting a sensor's trigger and handler.

See *sensor_trigger_set()* for argument description

typedef int (***sensor_sample_fetch_t**)(const struct *device* *dev, enum *sensor_channel* chan)

Callback API for fetching data from a sensor.

See *sensor_sample_fetch()* for argument description

typedef int (*sensor_channel_get_t)(const struct *device* *dev, enum *sensor_channel* chan, struct *sensor_value* *val)

Callback API for getting a reading from a sensor.

See *sensor_channel_get()* for argument description

typedef int (*sensor_get_decoder_t)(const struct *device* *dev, const struct *sensor_decoder_api* **api)

Get the decoder associate with the given device.

See also

sensor_get_decoder for more details

typedef void (*sensor_submit_t)(const struct *device* *sensor, struct *rtio_iodev_sqe* *sqe)

typedef void (*sensor_processing_callback_t)(int result, uint8_t *buf, uint32_t buf_len, void *userdata)

Callback function used with the helper processing function.

See also

sensor_processing_with_callback

Param result
[in] The result code of the read (0 being success)

Param buf
[in] The data buffer holding the sensor data

Param buf_len
[in] The length (in bytes) of the buf

Param userdata
[in] The optional userdata passed to *sensor_read_async_mempool()*

Enums

enum sensor_channel

Sensor channels.

Values:

enumerator SENSOR_CHAN_ACCEL_X

Acceleration on the X axis, in m/s^2.

enumerator SENSOR_CHAN_ACCEL_Y

Acceleration on the Y axis, in m/s^2.

enumerator SENSOR_CHAN_ACCEL_Z
Acceleration on the Z axis, in m/s^2.

enumerator SENSOR_CHAN_ACCEL_XYZ
Acceleration on the X, Y and Z axes.

enumerator SENSOR_CHAN_GYRO_X
Angular velocity around the X axis, in radians/s.

enumerator SENSOR_CHAN_GYRO_Y
Angular velocity around the Y axis, in radians/s.

enumerator SENSOR_CHAN_GYRO_Z
Angular velocity around the Z axis, in radians/s.

enumerator SENSOR_CHAN_GYRO_XYZ
Angular velocity around the X, Y and Z axes.

enumerator SENSOR_CHAN_MAGN_X
Magnetic field on the X axis, in Gauss.

enumerator SENSOR_CHAN_MAGN_Y
Magnetic field on the Y axis, in Gauss.

enumerator SENSOR_CHAN_MAGN_Z
Magnetic field on the Z axis, in Gauss.

enumerator SENSOR_CHAN_MAGN_XYZ
Magnetic field on the X, Y and Z axes.

enumerator SENSOR_CHAN_DIE_TEMP
Device die temperature in degrees Celsius.

enumerator SENSOR_CHAN_AMBIENT_TEMP
Ambient temperature in degrees Celsius.

enumerator SENSOR_CHAN_PRESS
Pressure in kilopascal.

enumerator SENSOR_CHAN_PROX
Proximity.
Adimensional. A value of 1 indicates that an object is close.

enumerator SENSOR_CHAN_HUMIDITY
Humidity, in percent.

enumerator SENSOR_CHAN_LIGHT
Illuminance in visible spectrum, in lux.

enumerator SENSOR_CHAN_IR
Illuminance in infra-red spectrum, in lux.

enumerator SENSOR_CHAN_RED
Illuminance in red spectrum, in lux.

enumerator SENSOR_CHAN_GREEN
Illuminance in green spectrum, in lux.

enumerator SENSOR_CHAN_BLUE
Illuminance in blue spectrum, in lux.

enumerator SENSOR_CHAN_ALTITUDE
Altitude, in meters.

enumerator SENSOR_CHAN_PM_1_0
1.0 micro-meters Particulate Matter, in ug/m^3

enumerator SENSOR_CHAN_PM_2_5
2.5 micro-meters Particulate Matter, in ug/m^3

enumerator SENSOR_CHAN_PM_10
10 micro-meters Particulate Matter, in ug/m^3

enumerator SENSOR_CHAN_DISTANCE
Distance.
From sensor to target, in meters

enumerator SENSOR_CHAN_CO2
CO2 level, in parts per million (ppm)

enumerator SENSOR_CHAN_O2
O2 level, in parts per million (ppm)

enumerator SENSOR_CHAN_VOC
VOC level, in parts per billion (ppb)

enumerator SENSOR_CHAN_GAS_RES
Gas sensor resistance in ohms.

enumerator SENSOR_CHAN_VOLTAGE
Voltage, in volts.

enumerator SENSOR_CHAN_VSHUNT
Current Shunt Voltage in milli-volts.

enumerator SENSOR_CHAN_CURRENT
Current, in amps.

enumerator SENSOR_CHAN_POWER
Power in watts.

enumerator SENSOR_CHAN_RESISTANCE
Resistance , in Ohm.

enumerator SENSOR_CHAN_ROTATION
Angular rotation, in degrees.

enumerator SENSOR_CHAN_POS_DX
Position change on the X axis, in points.

enumerator SENSOR_CHAN_POS_DY
Position change on the Y axis, in points.

enumerator SENSOR_CHAN_POS_DZ
Position change on the Z axis, in points.

enumerator SENSOR_CHAN_POS_DXYZ
Position change on the X, Y and Z axis, in points.

enumerator SENSOR_CHAN_RPM
Revolutions per minute, in RPM.

enumerator SENSOR_CHAN_GAUGE_VOLTAGE
Voltage, in volts.

enumerator SENSOR_CHAN_GAUGE_AVG_CURRENT
Average current, in amps.

enumerator SENSOR_CHAN_GAUGE_STDBY_CURRENT
Standby current, in amps.

enumerator SENSOR_CHAN_GAUGE_MAX_LOAD_CURRENT
Max load current, in amps.

enumerator SENSOR_CHAN_GAUGE_TEMP
Gauge temperature

enumerator SENSOR_CHAN_GAUGE_STATE_OF_CHARGE
State of charge measurement in %.

enumerator SENSOR_CHAN_GAUGE_FULL_CHARGE_CAPACITY
Full Charge Capacity in mAh.

enumerator SENSOR_CHAN_GAUGE_REMAINING_CHARGE_CAPACITY
Remaining Charge Capacity in mAh.

enumerator SENSOR_CHAN_GAUGE_NOM_AVAIL_CAPACITY
Nominal Available Capacity in mAh.

enumerator SENSOR_CHAN_GAUGE_FULL_AVAIL_CAPACITY
Full Available Capacity in mAh.

enumerator SENSOR_CHAN_GAUGE_AVG_POWER
Average power in mW.

enumerator SENSOR_CHAN_GAUGE_STATE_OF_HEALTH
State of health measurement in %.

enumerator SENSOR_CHAN_GAUGE_TIME_TO_EMPTY
Time to empty in minutes.

enumerator SENSOR_CHAN_GAUGE_TIME_TO_FULL
Time to full in minutes.

enumerator SENSOR_CHAN_GAUGE_CYCLE_COUNT
Cycle count (total number of charge/discharge cycles)

enumerator SENSOR_CHAN_GAUGE_DESIGN_VOLTAGE
Design voltage of cell in V (max voltage)

enumerator SENSOR_CHAN_GAUGE_DESIRED_VOLTAGE
Desired voltage of cell in V (nominal voltage)

enumerator SENSOR_CHAN_GAUGE_DESIRED_CHARGING_CURRENT
Desired charging current in mA.

enumerator SENSOR_CHAN_ALL
All channels.

enumerator SENSOR_CHAN_COMMON_COUNT
Number of all common sensor channels.

enumerator SENSOR_CHAN_PRIV_START = *SENSOR_CHAN_COMMON_COUNT*
This and higher values are sensor specific.

Refer to the sensor header file.

enumerator SENSOR_CHAN_MAX = INT16_MAX
Maximum value describing a sensor channel type.

enum sensor_trigger_type
Sensor trigger types.

Values:

enumerator SENSOR_TRIG_TIMER

Timer-based trigger, useful when the sensor does not have an interrupt line.

enumerator SENSOR_TRIG_DATA_READY

Trigger fires whenever new data is ready.

enumerator SENSOR_TRIG_DELTA

Trigger fires when the selected channel varies significantly.

This includes any-motion detection when the channel is acceleration or gyro. If detection is based on slope between successive channel readings, the slope threshold is configured via the *SENSOR_ATTR_SLOPE_TH* and *SENSOR_ATTR_SLOPE_DUR* attributes.

enumerator SENSOR_TRIG_NEAR_FAR

Trigger fires when a near/far event is detected.

enumerator SENSOR_TRIG_THRESHOLD

Trigger fires when channel reading transitions configured thresholds.

The thresholds are configured via the *SENSOR_ATTR_LOWER_THRESH*, *SENSOR_ATTR_UPPER_THRESH*, and *SENSOR_ATTR_HYSTERESIS* attributes.

enumerator SENSOR_TRIG_TAP

Trigger fires when a single tap is detected.

enumerator SENSOR_TRIG_DOUBLE_TAP

Trigger fires when a double tap is detected.

enumerator SENSOR_TRIG_FREEFALL

Trigger fires when a free fall is detected.

enumerator SENSOR_TRIG_MOTION

Trigger fires when motion is detected.

enumerator SENSOR_TRIG_STATIONARY

Trigger fires when no motion has been detected for a while.

enumerator SENSOR_TRIG_FIFO_WATERMARK

Trigger fires when the FIFO watermark has been reached.

enumerator SENSOR_TRIG_FIFO_FULL

Trigger fires when the FIFO becomes full.

enumerator SENSOR_TRIG_COMMON_COUNT

Number of all common sensor triggers.

enumerator SENSOR_TRIG_PRIV_START = *SENSOR_TRIG_COMMON_COUNT*

This and higher values are sensor specific.

Refer to the sensor header file.

enumerator SENSOR_TRIG_MAX = INT16_MAX

Maximum value describing a sensor trigger type.

enum sensor_attribute

Sensor attribute types.

Values:

enumerator SENSOR_ATTR_SAMPLING_FREQUENCY

Sensor sampling frequency, i.e.

how many times a second the sensor takes a measurement.

enumerator SENSOR_ATTR_LOWER_THRESH

Lower threshold for trigger.

enumerator SENSOR_ATTR_UPPER_THRESH

Upper threshold for trigger.

enumerator SENSOR_ATTR_SLOPE_TH

Threshold for any-motion (slope) trigger.

enumerator SENSOR_ATTR_SLOPE_DUR

Duration for which the slope values needs to be outside the threshold for the trigger to fire.

enumerator SENSOR_ATTR_HYSTERESIS

enumerator SENSOR_ATTR_OVERSAMPLING

Oversampling factor.

enumerator SENSOR_ATTR_FULL_SCALE

Sensor range, in SI units.

enumerator SENSOR_ATTR_OFFSET

The sensor value returned will be altered by the amount indicated by offset: final_value = *sensor_value* + offset.

enumerator SENSOR_ATTR_CALIB_TARGET

Calibration target.

This will be used by the internal chip's algorithms to calibrate itself on a certain axis, or all of them.

enumerator SENSOR_ATTR_CONFIGURATION

Configure the operating modes of a sensor.

enumerator SENSOR_ATTR_CALIBRATION

Set a calibration value needed by a sensor.

enumerator SENSOR_ATTR_FEATURE_MASK
Enable/disable sensor features.

enumerator SENSOR_ATTR_ALERT
Alert threshold or alert enable/disable.

enumerator SENSOR_ATTR_FF_DUR
Free-fall duration represented in milliseconds.
If the sampling frequency is changed during runtime, this attribute should be set to adjust freefall duration to the new sampling frequency.

enumerator SENSOR_ATTR_BATCH_DURATION
Hardware batch duration in ticks.

enumerator SENSOR_ATTR_COMMON_COUNT
Number of all common sensor attributes.

enumerator SENSOR_ATTR_PRIV_START = *SENSOR_ATTR_COMMON_COUNT*
This and higher values are sensor specific.
Refer to the sensor header file.

enumerator SENSOR_ATTR_MAX = INT16_MAX
Maximum value describing a sensor attribute type.

enum sensor_stream_data_opt
Options for what to do with the associated data when a trigger is consumed.
Values:

enumerator SENSOR_STREAM_DATA_INCLUDE = 0
Include whatever data is associated with the trigger.

enumerator SENSOR_STREAM_DATA_NOP = 1
Do nothing with the associated trigger data, it may be consumed later.

enumerator SENSOR_STREAM_DATA_DROP = 2
Flush/clear whatever data is associated with the trigger.

Functions

static inline bool sensor_chan_spec_eq(struct *sensor_chan_spec* chan_spec0, struct *sensor_chan_spec* chan_spec1)
Check if channel specs are equivalent.

Parameters

- chan_spec0 – First chan spec
- chan_spec1 – Second chan spec

Return values

- true – If equivalent

- `false` – If not equivalent

static inline int **sensor_decode**(struct *sensor_decode_context* *ctx, void *out, uint16_t max_count)

Decode N frames using a *sensor_decode_context*.

Parameters

- `ctx` – **[inout]** The context to use for decoding
- `out` – **[out]** The output buffer
- `max_count` – **[in]** Maximum number of frames to decode

Returns

The decode result from *sensor_decoder_api*'s decode function

int **sensor_natively_supported_channel_size_info**(struct *sensor_chan_spec* channel, size_t *base_size, size_t *frame_size)

int **sensor_attr_set**(const struct *device* *dev, enum *sensor_channel* chan, enum *sensor_attribute* attr, const struct *sensor_value* *val)

Set an attribute for a sensor.

Parameters

- `dev` – Pointer to the sensor device
- `chan` – The channel the attribute belongs to, if any. Some attributes may only be set for all channels of a device, depending on device capabilities.
- `attr` – The attribute to set
- `val` – The value to set the attribute to

Returns

0 if successful, negative errno code if failure.

int **sensor_attr_get**(const struct *device* *dev, enum *sensor_channel* chan, enum *sensor_attribute* attr, struct *sensor_value* *val)

Get an attribute for a sensor.

Parameters

- `dev` – Pointer to the sensor device
- `chan` – The channel the attribute belongs to, if any. Some attributes may only be set for all channels of a device, depending on device capabilities.
- `attr` – The attribute to get
- `val` – Pointer to where to store the attribute

Returns

0 if successful, negative errno code if failure.

static inline int **sensor_trigger_set**(const struct *device* *dev, const struct *sensor_trigger* *trig, *sensor_trigger_handler_t* handler)

Activate a sensor's trigger and set the trigger handler.

The handler will be called from a thread, so I2C or SPI operations are safe. However, the thread's stack is limited and defined by the driver. It is currently up to the caller to ensure that the handler does not overflow the stack.

The user-allocated trigger will be stored by the driver as a pointer, rather than a copy, and passed back to the handler. This enables the handler to use CONTAINER_OF to retrieve a context pointer when the trigger is embedded in a larger struct and requires that the trigger is not allocated on the stack.

Function properties (list may not be complete)
supervisor

Parameters

- `dev` – Pointer to the sensor device
- `trig` – The trigger to activate
- `handler` – The function that should be called when the trigger fires

Returns
0 if successful, negative errno code if failure.

int `sensor_sample_fetch`(const struct *device* *dev)

Fetch a sample from the sensor and store it in an internal driver buffer.

Read all of a sensor's active channels and, if necessary, perform any additional operations necessary to make the values useful. The user may then get individual channel values by calling *sensor_channel_get*.

The function blocks until the fetch operation is complete.

Since the function communicates with the sensor device, it is unsafe to call it in an ISR if the device is connected via I2C or SPI.

Parameters

- `dev` – Pointer to the sensor device

Returns
0 if successful, negative errno code if failure.

int `sensor_sample_fetch_chan`(const struct *device* *dev, enum *sensor_channel* type)

Fetch a sample from the sensor and store it in an internal driver buffer.

Read and compute compensation for one type of sensor data (magnetometer, accelerometer, etc). The user may then get individual channel values by calling *sensor_channel_get*.

This is mostly implemented by multi function devices enabling reading at different sampling rates.

The function blocks until the fetch operation is complete.

Since the function communicates with the sensor device, it is unsafe to call it in an ISR if the device is connected via I2C or SPI.

Parameters

- `dev` – Pointer to the sensor device
- `type` – The channel that needs updated

Returns
0 if successful, negative errno code if failure.

int `sensor_channel_get`(const struct *device* *dev, enum *sensor_channel* chan, struct *sensor_value* *val)

Get a reading from a sensor device.

Return a useful value for a particular channel, from the driver's internal data. Before calling this function, a sample must be obtained by calling *sensor_sample_fetch* or *sensor_sample_fetch_chan*. It is guaranteed that two subsequent calls of this function for the same channels will yield the same value, if *sensor_sample_fetch* or *sensor_sample_fetch_chan* has not been called in the meantime.

For vectorial data samples you can request all axes in just one call by passing the specific channel with _XYZ suffix. The sample will be returned at val[0], val[1] and val[2] (X, Y and Z in that order).

Parameters

- `dev` – Pointer to the sensor device
- `chan` – The channel to read
- `val` – Where to store the value

Returns
0 if successful, negative errno code if failure.

int `sensor_get_decoder`(const struct *device* *dev, const struct *sensor_decoder_api* **decoder)

Get the sensor's decoder API.

Parameters

- `dev` – **[in]** The sensor device
- `decoder` – **[in]** Pointer to the decoder which will be set upon success

Returns
0 on success

Returns
< 0 on error

int `sensor_reconfigure_read_iodev`(struct *rtio_iodev* *iodev, const struct *device* *sensor, const struct *sensor_chan_spec* *channels, size_t num_channels)

Reconfigure a reading iodev.

Allows a reconfiguration of the iodev associated with reading a sample from a sensor.

Important: If the iodev is currently servicing a read operation, the data will likely be invalid. Please be sure the flush or wait for all pending operations to complete before using the data after a configuration change.

It is also important that the `data` field of the iodev is a *sensor_read_config*.

Parameters

- `iodev` – **[in]** The iodev to reconfigure
- `sensor` – **[in]** The sensor to read from
- `channels` – **[in]** One or more channels to read
- `num_channels` – **[in]** The number of channels in `channels`

Returns
0 on success

Returns
< 0 on error

static inline int `sensor_stream`(struct *rtio_iodev* *iodev, struct *rtio* *ctx, void *userdata, struct *rtio_sqe* **handle)

static inline int `sensor_read`(struct *rtio_iodev* *iodev, struct *rtio* *ctx, uint8_t *buf, size_t buf_len)

Blocking one shot read of samples from a sensor into a buffer.

Using `cfg`, read data from the device by using the provided RTIO context `ctx`. This call will generate a *rtio_sqe* that will be given the provided buffer. The call will wait for the read to complete before returning to the caller.

Parameters

- `iodev` – **[in]** The iodev created by *SENSOR_DT_READ_IODEV*
- `ctx` – **[in]** The RTIO context to service the read
- `buf` – **[in]** Pointer to memory to read sample data into
- `buf_len` – **[in]** Size in bytes of the given memory that are valid to read into

Returns
0 on success

Returns
< 0 on error

static inline int **sensor_read_async_mempool**(struct *rtio_iodev* *iodev, struct *rtio* *ctx, void *userdata)

One shot non-blocking read with pool allocated buffer.

Using `cfg`, read one snapshot of data from the device by using the provided RTIO context `ctx`. This call will generate a *rtio_sqe* that will leverage the RTIO's internal mempool when the time comes to service the read.

Parameters

- `iodev` – **[in]** The iodev created by *SENSOR_DT_READ_IODEV*
- `ctx` – **[in]** The RTIO context to service the read
- `userdata` – **[in]** Optional userdata that will be available when the read is complete

Returns
0 on success

Returns
< 0 on error

void **sensor_processing_with_callback**(struct *rtio* *ctx, *sensor_processing_callback_t* cb)

Helper function for common processing of sensor data.

This function can be called in a blocking manner after *sensor_read()* or in a standalone thread dedicated to processing. It will wait for a cqe from the RTIO context, once received, it will decode the userdata and call the `cb`. Once the `cb` returns, the buffer will be released back into `ctx`'s mempool if available.

Parameters

- `ctx` – **[in]** The RTIO context to wait on
- `cb` – **[in]** Callback to call when data is ready for processing

static inline int32_t **sensor_ms2_to_g**(const struct *sensor_value* *ms2)

Helper function to convert acceleration from m/s^2 to Gs.

Parameters

- `ms2` – A pointer to a *sensor_value* struct holding the acceleration, in m/s^2.

Returns
The converted value, in Gs.

static inline void **sensor_g_to_ms2**(int32_t g, struct *sensor_value* *ms2)

Helper function to convert acceleration from Gs to m/s^2.

Parameters

- `g` – The G value to be converted.

- **ms2** – A pointer to a *sensor_value* struct, where the result is stored.

static inline int32_t **sensor_ms2_to_ug**(const struct *sensor_value* *ms2)

Helper function to convert acceleration from m/s^2 to micro Gs.

Parameters

- **ms2** – A pointer to a *sensor_value* struct holding the acceleration, in m/s^2.

Returns

The converted value, in micro Gs.

static inline void **sensor_ug_to_ms2**(int32_t ug, struct *sensor_value* *ms2)

Helper function to convert acceleration from micro Gs to m/s^2.

Parameters

- **ug** – The micro G value to be converted.
- **ms2** – A pointer to a *sensor_value* struct, where the result is stored.

static inline int32_t **sensor_rad_to_degrees**(const struct *sensor_value* *rad)

Helper function for converting radians to degrees.

Parameters

- **rad** – A pointer to a *sensor_value* struct, holding the value in radians.

Returns

The converted value, in degrees.

static inline void **sensor_degrees_to_rad**(int32_t d, struct *sensor_value* *rad)

Helper function for converting degrees to radians.

Parameters

- **d** – The value (in degrees) to be converted.
- **rad** – A pointer to a *sensor_value* struct, where the result is stored.

static inline int32_t **sensor_rad_to_10udegrees**(const struct *sensor_value* *rad)

Helper function for converting radians to 10 micro degrees.

When the unit is 1 micro degree, the range that the int32_t can represent is +/-2147.483 degrees. In order to increase this range, here we use 10 micro degrees as the unit.

Parameters

- **rad** – A pointer to a *sensor_value* struct, holding the value in radians.

Returns

The converted value, in 10 micro degrees.

static inline void **sensor_10udegrees_to_rad**(int32_t d, struct *sensor_value* *rad)

Helper function for converting 10 micro degrees to radians.

Parameters

- **d** – The value (in 10 micro degrees) to be converted.
- **rad** – A pointer to a *sensor_value* struct, where the result is stored.

static inline double **sensor_value_to_double**(const struct *sensor_value* *val)

Helper function for converting struct *sensor_value* to double.

Parameters

- **val** – A pointer to a *sensor_value* struct.

Returns

The converted value.

static inline float **sensor_value_to_float**(const struct *sensor_value* *val)

Helper function for converting struct *sensor_value* to float.

Parameters

- **val** – A pointer to a *sensor_value* struct.

Returns

The converted value.

static inline int **sensor_value_from_double**(struct *sensor_value* *val, double inp)

Helper function for converting double to struct *sensor_value*.

Parameters

- **val** – A pointer to a *sensor_value* struct.
- **inp** – The converted value.

Returns

0 if successful, negative errno code if failure.

static inline int **sensor_value_from_float**(struct *sensor_value* *val, float inp)

Helper function for converting float to struct *sensor_value*.

Parameters

- **val** – A pointer to a *sensor_value* struct.
- **inp** – The converted value.

Returns

0 if successful, negative errno code if failure.

static inline int64_t **sensor_value_to_milli**(const struct *sensor_value* *val)

Helper function for converting struct *sensor_value* to integer milli units.

Parameters

- **val** – A pointer to a *sensor_value* struct.

Returns

The converted value.

static inline int64_t **sensor_value_to_micro**(const struct *sensor_value* *val)

Helper function for converting struct *sensor_value* to integer micro units.

Parameters

- **val** – A pointer to a *sensor_value* struct.

Returns

The converted value.

static inline int **sensor_value_from_milli**(struct *sensor_value* *val, int64_t milli)

Helper function for converting integer milli units to struct *sensor_value*.

Parameters

- **val** – A pointer to a *sensor_value* struct.
- **milli** – The converted value.

Returns

0 if successful, negative errno code if failure.

static inline int **sensor_value_from_micro**(struct *sensor_value* *val, int64_t micro)

Helper function for converting integer micro units to struct *sensor_value*.

Parameters

- `val` – A pointer to a *sensor_value* struct.
- `micro` – The converted value.

Returns
0 if successful, negative errno code if failure.

struct `sensor_value`

#include <*sensor.h*> Representation of a sensor readout value.

The value is represented as having an integer and a fractional part, and can be obtained using the formula val1 + val2 * 10^(-6). Negative values also adhere to the above formula, but may need special attention. Here are some examples of the value representation:

```
 0.5: val1 =  0, val2 =  500000
-0.5: val1 =  0, val2 = -500000
-1.0: val1 = -1, val2 =  0
-1.5: val1 = -1, val2 = -500000
```

Public Members

int32_t `val1`

Integer part of the value.

int32_t `val2`

Fractional part of the value (in one-millionth parts).

struct `sensor_trigger`

#include <*sensor.h*> Sensor trigger spec.

Public Members

enum *sensor_trigger_type* `type`

Trigger type.

enum *sensor_channel* `chan`

Channel the trigger is set on.

struct `sensor_chan_spec`

#include <*sensor.h*> Sensor Channel Specification.

A sensor channel specification is a unique identifier per sensor device describing a measurement channel.

Note

Typically passed by value as the size of a *sensor_chan_spec* is a single word.

Public Members

uint16_t **chan_type**

A sensor channel type.

uint16_t **chan_idx**

A sensor channel index.

struct **sensor_decoder_api**

#include <*sensor.h*> Decodes a single raw data buffer.

Data buffers are provided on the *RTIO* context that's supplied to *sensor_read*.

Public Members

int (***get_frame_count**)(const uint8_t *buffer, struct *sensor_chan_spec* channel, uint16_t *frame_count)

Get the number of frames in the current buffer.

Param buffer
[in] The buffer provided on the *RTIO* context.
Param channel
[in] The channel to get the count for
Param frame_count
[out] The number of frames on the buffer (at least 1)
Return
0 on success
Return
-ENOTSUP if the channel/channel_idx aren't found

int (***get_size_info**)(struct *sensor_chan_spec* channel, size_t *base_size, size_t *frame_size)

Get the size required to decode a given channel.

When decoding a single frame, use `base_size`. For every additional frame, add another `frame_size`. As an example, to decode 3 frames use: 'base_size + 2 * frame_size'.

Param channel
[in] The channel to query
Param base_size
[out] The size of decoding the first frame
Param frame_size
[out] The additional size of every additional frame
Return
0 on success
Return
-ENOTSUP if the channel is not supported

int (***decode**)(const uint8_t *buffer, struct *sensor_chan_spec* channel, uint32_t *fit, uint16_t max_count, void *data_out)

Decode up to `max_count` samples from the buffer.

Decode samples of channel *sensor_channel* across multiple frames. If there exist multiple instances of the same channel, `channel_index` is used to differentiate them. As an example, assume a sensor provides 2 distance measurements:

```
// Decode the first channel instance of 'distance'
decoder->decode(buffer, SENSOR_CHAN_DISTANCE, 0, &fit, 5, out);
...

// Decode the second channel instance of 'distance'
decoder->decode(buffer, SENSOR_CHAN_DISTANCE, 1, &fit, 5, out);
```

Param buffer
[in] The buffer provided on the *RTIO* context
Param channel
[in] The channel to decode
Param fit
[inout] The current frame iterator
Param max_count
[in] The maximum number of channels to decode.
Param data_out
[out] The decoded data
Return
0 no more samples to decode
Return
>0 the number of decoded frames
Return
<0 on error

bool (***has_trigger**)(const uint8_t *buffer, enum *sensor_trigger_type* trigger)

Check if the given trigger type is present.
Param buffer
[in] The buffer provided on the *RTIO* context
Param trigger
[in] The trigger type in question
Return
Whether the trigger is present in the buffer

struct **sensor_decode_context**

#include <*sensor.h*> Used for iterating over the data frames via the *sensor_decoder_api*.

Example usage:

```
(.c)
   struct sensor_decode_context ctx = SENSOR_DECODE_CONTEXT_INIT(
       decoder, buffer, SENSOR_CHAN_ACCEL_XYZ, 0);

   while (true) {
     struct sensor_three_axis_data accel_out_data;

     num_decoded_channels = sensor_decode(ctx, &accel_out_data, 1);

     if (num_decoded_channels <= 0) {
       printk("Done decoding buffer\n");
       break;
     }

     printk("Decoded (%" PRId32 ", %" PRId32 ", %" PRId32 ")\n", accel_out_data.
↪readings[0].x,
         accel_out_data.readings[0].y, accel_out_data.readings[0].z);
   }
```

struct **sensor_stream_trigger**

#include <*sensor.h*>

struct **sensor_read_config**

#include <sensor.h>

struct **sensor_driver_api**

#include <sensor.h>

struct **sensor_data_generic_header**

#include <sensor.h>

group **sensor_emulator_backend**

Sensor emulator backend API.

Functions

static inline bool **emul_sensor_backend_is_supported**(const struct *emul* *target)

Check if a given sensor emulator supports the backend API.

Parameters

- **target** – Pointer to emulator instance to query

Returns

True if supported, false if unsupported or if `target` is NULL.

static inline int **emul_sensor_backend_set_channel**(const struct *emul* *target, struct *sensor_chan_spec* ch, const *q31_t* *value, int8_t shift)

Set an expected value for a given channel on a given sensor emulator.

Parameters

- **target** – Pointer to emulator instance to operate on
- **ch** – Sensor channel to set expected value for
- **value** – Expected value in fixed-point format using standard SI unit for sensor type
- **shift** – Shift value (scaling factor) applied to `value`

Returns

0 if successful

Returns

-ENOTSUP if no backend API or if channel not supported by emul

Returns

-ERANGE if provided value is not in the sensor's supported range

static inline int **emul_sensor_backend_get_sample_range**(const struct *emul* *target, struct *sensor_chan_spec* ch, *q31_t* *lower, *q31_t* *upper, *q31_t* *epsilon, int8_t *shift)

Query an emulator for a channel's supported sample value range and tolerance.

Parameters

- **target** – Pointer to emulator instance to operate on
- **ch** – The channel to request info for. If `ch` is unsupported, return `-ENOTSUP`

- `lower` – **[out]** Minimum supported sample value in SI units, fixed-point format
- `upper` – **[out]** Maximum supported sample value in SI units, fixed-point format
- `epsilon` – **[out]** Tolerance to use comparing expected and actual values to account for rounding and sensor precision issues. This can usually be set to the minimum sample value step size. Uses SI units and fixed-point format.
- `shift` – **[out]** The shift value (scaling factor) associated with `lower`, `upper`, and `epsilon`.

Returns
0 if successful

Returns
-ENOTSUP if no backend API or if channel not supported by emul

static inline int **emul_sensor_backend_set_attribute**(const struct *emul* *target, struct *sensor_chan_spec* ch, enum *sensor_attribute* attribute, const void *value)

Set the emulator's attribute values.

Parameters

- `target` – **[in]** Pointer to emulator instance to operate on
- `ch` – **[in]** The channel to request info for. If `ch` is unsupported, return `-ENOTSUP`
- `attribute` – **[in]** The attribute to set
- `value` – **[in]** the value to use (cast according to the channel/attribute pair)

Returns
0 is successful

Returns
< 0 on error

static inline int **emul_sensor_backend_get_attribute_metadata**(const struct *emul* *target, struct *sensor_chan_spec* ch, enum *sensor_attribute* attribute, *q31_t* *min, *q31_t* *max, *q31_t* *increment, int8_t *shift)

Get metadata about an attribute.

Information provided by this function includes the minimum/maximum values of the attribute as well as the increment (value per LSB) which can be used as an epsilon when comparing results.

Parameters

- `target` – **[in]** Pointer to emulator instance to operate on
- `ch` – **[in]** The channel to request info for. If `ch` is unsupported, return '-ENOTSUP'
- `attribute` – **[in]** The attribute to request info for. If `attribute` is unsupported, return '-ENOTSUP'

- `min` – **[out]** The minimum value the attribute can be set to
- `max` – **[out]** The maximum value the attribute can be set to
- `increment` – **[out]** The value that the attribute increases by for every LSB
- `shift` – **[out]** The shift for `min`, `max`, and `increment`

Returns
0 on SUCCESS

Returns
< 0 on error

7.6.45 Serial Peripheral Interface (SPI) Bus

Overview

API Reference

Related code samples

Enhanced Serial Peripheral Interface (eSPI)
Use eSPI to connect to a slave device and exchange virtual wire packets.

SPI bitbang
Use the bitbang SPI driver for communicating with a slave.

group `spi_interface`
SPI Interface.

Since
1.0

Version
1.0.0

SPI operational mode

`SPI_OP_MODE_MASTER`
Master mode.

`SPI_OP_MODE_SLAVE`
Slave mode.

`SPI_OP_MODE_GET`(_operation_)
Get SPI operational mode.

SPI Polarity & Phase Modes

SPI_MODE_CPOL

Clock Polarity: if set, clock idle state will be 1 and active state will be 0.

If untouched, the inverse will be true which is the default.

SPI_MODE_CPHA

Clock Phase: this dictates when is the data captured, and depends clock's polarity.

When SPI_MODE_CPOL is set and this bit as well, capture will occur on low to high transition and high to low if this bit is not set (default). This is fully reversed if CPOL is not set.

SPI_MODE_LOOP

Whatever data is transmitted is looped-back to the receiving buffer of the controller.

This is fully controller dependent as some may not support this, and can be used for testing purposes only.

SPI_MODE_GET(_mode_)

Get SPI polarity and phase mode bits.

SPI Transfer modes (host controller dependent)

SPI_TRANSFER_MSB

Most significant bit first.

SPI_TRANSFER_LSB

Least significant bit first.

SPI word size

SPI_WORD_SIZE_GET(_operation_)

Get SPI word size.

SPI_WORD_SET(_word_size_)

Set SPI word size.

Specific SPI devices control bits

SPI_HOLD_ON_CS

Requests - if possible - to keep CS asserted after the transaction.

SPI_LOCK_ON

Keep the device locked after the transaction for the current config.

Use this with extreme caution (see *spi_release()* below) as it will prevent other callers to access the SPI device until *spi_release()* is properly called.

SPI_CS_ACTIVE_HIGH
: Active high logic on CS.

  Usually, and by default, CS logic is active low. However, some devices may require the reverse logic: active high. This bit will request the controller to use that logic. Note that not all controllers are able to handle that natively. In this case deferring the CS control to a gpio line through struct *spi_cs_control* would be the solution.

SPI MISO lines

Some controllers support dual, quad or octal MISO lines connected to slaves.

Default is single, which is the case most of the time. Without CONFIG_SPI_EXTENDED_MODES being enabled, single is the only supported one.

SPI_LINES_SINGLE
: Single line.

SPI_LINES_DUAL
: Dual lines.

SPI_LINES_QUAD
: Quad lines.

SPI_LINES_OCTAL
: Octal lines.

SPI_LINES_MASK
: Mask for MISO lines in spi_operation_t.

SPI duplex mode

Some controllers support half duplex transfer, which results in 3-wire usage.

By default, full duplex will prevail.

SPI_FULL_DUPLEX

SPI_HALF_DUPLEX

SPI Frame Format

2 frame formats are exposed: Motorola and TI.

The main difference is the behavior of the CS line. In Motorola it stays active the whole transfer. In TI, it's active only one serial clock period prior to actually make the transfer, it is thus inactive during the transfer, which ends when the clocks ends as well. By default, as it is the most commonly used, the Motorola frame format will prevail.

SPI_FRAME_FORMAT_MOTOROLA

SPI_FRAME_FORMAT_TI

Defines

SPI_CS_GPIOS_DT_SPEC_GET(spi_dev)

Get a `struct` *gpio_dt_spec* for a SPI device's chip select pin.

Example devicetree fragment:

```
gpio1: gpio@abcd0001 { ... };

gpio2: gpio@abcd0002 { ... };

spi@abcd0003 {
        compatible = "vnd,spi";
        cs-gpios = <&gpio1 10 GPIO_ACTIVE_LOW>,
                   <&gpio2 20 GPIO_ACTIVE_LOW>;

        a: spi-dev-a@0 {
                reg = <0>;
        };

        b: spi-dev-b@1 {
                reg = <1>;
        };
};
```

Example usage:

```
SPI_CS_GPIOS_DT_SPEC_GET(DT_NODELABEL(a)) \
      // { DEVICE_DT_GET(DT_NODELABEL(gpio1)), 10, GPIO_ACTIVE_LOW }
SPI_CS_GPIOS_DT_SPEC_GET(DT_NODELABEL(b)) \
      // { DEVICE_DT_GET(DT_NODELABEL(gpio2)), 20, GPIO_ACTIVE_LOW }
```

Parameters

- **spi_dev** – a SPI device node identifier

Returns

gpio_dt_spec struct corresponding with spi_dev's chip select

SPI_CS_GPIOS_DT_SPEC_INST_GET(inst)

Get a `struct` *gpio_dt_spec* for a SPI device's chip select pin.

This is equivalent to *SPI_CS_GPIOS_DT_SPEC_GET(DT_DRV_INST(inst))*.

Parameters

- **inst** – Devicetree instance number

Returns

gpio_dt_spec struct corresponding with spi_dev's chip select

SPI_CS_CONTROL_INIT(node_id, delay_)

Initialize and get a pointer to a *spi_cs_control* from a devicetree node identifier.

This helper is useful for initializing a device on a SPI bus. It initializes a struct *spi_cs_control* and returns a pointer to it. Here, `node_id` is a node identifier for a SPI device, not a SPI controller.

Example devicetree fragment:

```
spi@abcd0001 {
        cs-gpios = <&gpio0 1 GPIO_ACTIVE_LOW>;
        spidev: spi-device@0 { ... };
};
```

Example usage:

```
struct spi_cs_control ctrl =
        SPI_CS_CONTROL_INIT(DT_NODELABEL(spidev), 2);
```

This example is equivalent to:

```
struct spi_cs_control ctrl = {
        .gpio = SPI_CS_GPIOS_DT_SPEC_GET(DT_NODELABEL(spidev)),
        .delay = 2,
};
```

Parameters

- **node_id** – Devicetree node identifier for a device on a SPI bus
- **delay_** – The `delay` field to set in the *spi_cs_control*

Returns
a pointer to the *spi_cs_control* structure

`SPI_CS_CONTROL_INIT_INST`(inst, delay_)

Get a pointer to a *spi_cs_control* from a devicetree node.

This is equivalent to *SPI_CS_CONTROL_INIT(DT_DRV_INST(inst), delay)*.

Therefore, `DT_DRV_COMPAT` must already be defined before using this macro.

Parameters

- **inst** – Devicetree node instance number
- **delay_** – The `delay` field to set in the *spi_cs_control*

Returns
a pointer to the *spi_cs_control* structure

`SPI_CONFIG_DT`(node_id, operation_, delay_)

Structure initializer for *spi_config* from devicetree.

This helper macro expands to a static initializer for a `struct` *spi_config* by reading the relevant `frequency`, `slave`, and `cs` data from the devicetree.

Parameters

- **node_id** – Devicetree node identifier for the SPI device whose struct *spi_config* to create an initializer for
- **operation_** – the desired `operation` field in the struct *spi_config*
- **delay_** – the desired `delay` field in the struct *spi_config*'s *spi_cs_control*, if there is one

`SPI_CONFIG_DT_INST`(inst, operation_, delay_)

Structure initializer for *spi_config* from devicetree instance.

This is equivalent to *SPI_CONFIG_DT(DT_DRV_INST(inst), operation_, delay_)*.

Parameters

- **inst** – Devicetree instance number
- **operation_** – the desired `operation` field in the struct *spi_config*
- **delay_** – the desired `delay` field in the struct *spi_config*'s *spi_cs_control*, if there is one

SPI_DT_SPEC_GET(node_id, operation_, delay_)

Structure initializer for *spi_dt_spec* from devicetree.

This helper macro expands to a static initializer for a `struct` *spi_dt_spec* by reading the relevant bus, frequency, slave, and cs data from the devicetree.

Important: multiple fields are automatically constructed by this macro which must be checked before use. *spi_is_ready_dt* performs the required *device_is_ready* checks.

Parameters

- **node_id** – Devicetree node identifier for the SPI device whose struct *spi_dt_spec* to create an initializer for
- **operation_** – the desired `operation` field in the struct *spi_config*
- **delay_** – the desired `delay` field in the struct *spi_config*'s *spi_cs_control*, if there is one

SPI_DT_SPEC_INST_GET(inst, operation_, delay_)

Structure initializer for *spi_dt_spec* from devicetree instance.

This is equivalent to *SPI_DT_SPEC_GET(DT_DRV_INST(inst), operation_, delay_)*.

Parameters

- **inst** – Devicetree instance number
- **operation_** – the desired `operation` field in the struct *spi_config*
- **delay_** – the desired `delay` field in the struct *spi_config*'s *spi_cs_control*, if there is one

SPI_DEVICE_DT_DEFINE(node_id, init_fn, pm, data, config, level, prio, api, ...)

SPI_STATS_RX_BYTES_INC(dev_)

SPI_STATS_TX_BYTES_INC(dev_)

SPI_STATS_TRANSFER_ERROR_INC(dev_)

spi_transceive_stats(dev, error, tx_bufs, rx_bufs)

SPI_DT_IODEV_DEFINE(name, node_id, operation_, delay_)

Define an iodev for a given dt node on the bus.

These do not need to be shared globally but doing so will save a small amount of memory.

Parameters

- **name** – Symbolic name to use for defining the iodev
- **node_id** – Devicetree node identifier
- **operation_** – SPI operational mode
- **delay_** – Chip select delay in microseconds

Typedefs

typedef uint16_t **spi_operation_t**

Opaque type to hold the SPI operation flags.

typedef int (*spi_api_io)(const struct *device* *dev, const struct *spi_config* *config, const struct *spi_buf_set* *tx_bufs, const struct *spi_buf_set* *rx_bufs)

Callback API for I/O See *spi_transceive()* for argument descriptions.

Callback API for asynchronous I/O See *spi_transceive_signal()* for argument descriptions.

typedef void (*spi_callback_t)(const struct *device* *dev, int result, void *data)

SPI callback for asynchronous transfer requests.

Param dev
SPI device which is notifying of transfer completion or error

Param result
Result code of the transfer request. 0 is success, -errno for failure.

Param data
Transfer requester supplied data which is passed along to the callback.

typedef int (*spi_api_io_async)(const struct *device* *dev, const struct *spi_config* *config, const struct *spi_buf_set* *tx_bufs, const struct *spi_buf_set* *rx_bufs, *spi_callback_t* cb, void *userdata)

typedef int (*spi_api_release)(const struct *device* *dev, const struct *spi_config* *config)

Callback API for unlocking SPI device.

See *spi_release()* for argument descriptions

Functions

static inline bool spi_cs_is_gpio(const struct *spi_config* *config)

Check if SPI CS is controlled using a GPIO.

Parameters

- config – SPI configuration.

Returns
true If CS is controlled using a GPIO.

Returns
false If CS is controlled by hardware or any other means.

static inline bool spi_cs_is_gpio_dt(const struct *spi_dt_spec* *spec)

Check if SPI CS in *spi_dt_spec* is controlled using a GPIO.

Parameters

- spec – SPI specification from devicetree.

Returns
true If CS is controlled using a GPIO.

Returns
false If CS is controlled by hardware or any other means.

static inline bool spi_is_ready_dt(const struct *spi_dt_spec* *spec)

Validate that SPI bus (and CS gpio if defined) is ready.

Parameters

- spec – SPI specification from devicetree

Return values

- `true` – if the SPI bus is ready for use.
- `false` – if the SPI bus (or the CS gpio defined) is not ready for use.

int **spi_transceive**(const struct *device* *dev, const struct *spi_config* *config, const struct *spi_buf_set* *tx_bufs, const struct *spi_buf_set* *rx_bufs)

Read/write the specified amount of data from the SPI driver.

> **Note**
>
> This function is synchronous.

Parameters

- `dev` – Pointer to the device structure for the driver instance
- `config` – Pointer to a valid *spi_config* structure instance. Pointer-comparison may be used to detect changes from previous operations.
- `tx_bufs` – Buffer array where data to be sent originates from, or NULL if none.
- `rx_bufs` – Buffer array where data to be read will be written to, or NULL if none.

Return values

- `frames` – Positive number of frames received in slave mode.
- `0` – If successful in master mode.
- `-errno` – Negative errno code on failure.

static inline int **spi_transceive_dt**(const struct *spi_dt_spec* *spec, const struct *spi_buf_set* *tx_bufs, const struct *spi_buf_set* *rx_bufs)

Read/write data from an SPI bus specified in *spi_dt_spec*.

This is equivalent to:

```
spi_transceive(spec->bus, &spec->config, tx_bufs, rx_bufs);
```

Parameters

- `spec` – SPI specification from devicetree
- `tx_bufs` – Buffer array where data to be sent originates from, or NULL if none.
- `rx_bufs` – Buffer array where data to be read will be written to, or NULL if none.

Returns
a value from *spi_transceive()*.

static inline int **spi_read**(const struct *device* *dev, const struct *spi_config* *config, const struct *spi_buf_set* *rx_bufs)

Read the specified amount of data from the SPI driver.

> **Note**
>
> This function is synchronous.

Note

This function is a helper function calling spi_transceive.

Parameters

- `dev` – Pointer to the device structure for the driver instance
- `config` – Pointer to a valid *spi_config* structure instance. Pointer-comparison may be used to detect changes from previous operations.
- `rx_bufs` – Buffer array where data to be read will be written to.

Return values

- `frames` – Positive number of frames received in slave mode.
- `0` – If successful.
- `-errno` – Negative errno code on failure.

static inline int **spi_read_dt**(const struct *spi_dt_spec* *spec, const struct *spi_buf_set* *rx_bufs)

Read data from a SPI bus specified in *spi_dt_spec*.

This is equivalent to:

```
spi_read(spec->bus, &spec->config, rx_bufs);
```

Parameters

- `spec` – SPI specification from devicetree
- `rx_bufs` – Buffer array where data to be read will be written to.

Returns

a value from *spi_read()*.

static inline int **spi_write**(const struct *device* *dev, const struct *spi_config* *config, const struct *spi_buf_set* *tx_bufs)

Write the specified amount of data from the SPI driver.

Note

This function is synchronous.

Note

This function is a helper function calling spi_transceive.

Parameters

- `dev` – Pointer to the device structure for the driver instance
- `config` – Pointer to a valid *spi_config* structure instance. Pointer-comparison may be used to detect changes from previous operations.
- `tx_bufs` – Buffer array where data to be sent originates from.

Return values

- `0` – If successful.
- `-errno` – Negative errno code on failure.

static inline int **spi_write_dt**(const struct *spi_dt_spec* *spec, const struct *spi_buf_set* *tx_bufs)

Write data to a SPI bus specified in *spi_dt_spec*.

This is equivalent to:

```
spi_write(spec->bus, &spec->config, tx_bufs);
```

Parameters

- `spec` – SPI specification from devicetree
- `tx_bufs` – Buffer array where data to be sent originates from.

Returns

a value from *spi_write()*.

static inline int **spi_transceive_cb**(const struct *device* *dev, const struct *spi_config* *config, const struct *spi_buf_set* *tx_bufs, const struct *spi_buf_set* *rx_bufs, *spi_callback_t* callback, void *userdata)

Read/write the specified amount of data from the SPI driver.

Note

This function is asynchronous.

Note

This function is available only if `CONFIG_SPI_ASYNC` is selected.

Parameters

- `dev` – Pointer to the device structure for the driver instance
- `config` – Pointer to a valid *spi_config* structure instance. Pointer-comparison may be used to detect changes from previous operations.
- `tx_bufs` – Buffer array where data to be sent originates from, or NULL if none.
- `rx_bufs` – Buffer array where data to be read will be written to, or NULL if none.
- `callback` – Function pointer to completion callback. (Note: if NULL this function will not notify the end of the transaction, and whether it went successfully or not).
- `userdata` – Userdata passed to callback

Return values

- `frames` – Positive number of frames received in slave mode.
- `0` – If successful in master mode.
- `-errno` – Negative errno code on failure.

static inline int **spi_transceive_signal**(const struct *device* *dev, const struct *spi_config* *config, const struct *spi_buf_set* *tx_bufs, const struct *spi_buf_set* *rx_bufs, struct *k_poll_signal* *sig)

Read/write the specified amount of data from the SPI driver.

Note

This function is asynchronous.

Note

This function is available only if `CONFIG_SPI_ASYNC` and `CONFIG_POLL` are selected.

Parameters

- **dev** – Pointer to the device structure for the driver instance
- **config** – Pointer to a valid *spi_config* structure instance. Pointer-comparison may be used to detect changes from previous operations.
- **tx_bufs** – Buffer array where data to be sent originates from, or NULL if none.
- **rx_bufs** – Buffer array where data to be read will be written to, or NULL if none.
- **sig** – A pointer to a valid and ready to be signaled struct *k_poll_signal*. (Note: if NULL this function will not notify the end of the transaction, and whether it went successfully or not).

Return values

- **frames** – Positive number of frames received in slave mode.
- **0** – If successful in master mode.
- **-errno** – Negative errno code on failure.

static inline int **spi_read_signal**(const struct *device* *dev, const struct *spi_config* *config, const struct *spi_buf_set* *rx_bufs, struct *k_poll_signal* *sig)

Read the specified amount of data from the SPI driver.

Note

This function is asynchronous.

Note

This function is a helper function calling spi_transceive_signal.

Note

This function is available only if `CONFIG_SPI_ASYNC` and `CONFIG_POLL` are selected.

Parameters

- `dev` – Pointer to the device structure for the driver instance
- `config` – Pointer to a valid *spi_config* structure instance. Pointer-comparison may be used to detect changes from previous operations.
- `rx_bufs` – Buffer array where data to be read will be written to.
- `sig` – A pointer to a valid and ready to be signaled struct *k_poll_signal*. (Note: if NULL this function will not notify the end of the transaction, and whether it went successfully or not).

Return values

- `frames` – Positive number of frames received in slave mode.
- `0` – If successful
- `-errno` – Negative errno code on failure.

static inline int **spi_write_signal**(const struct *device* *dev, const struct *spi_config* *config, const struct *spi_buf_set* *tx_bufs, struct *k_poll_signal* *sig)

Write the specified amount of data from the SPI driver.

Note

This function is asynchronous.

Note

This function is a helper function calling spi_transceive_signal.

Note

This function is available only if `CONFIG_SPI_ASYNC` and `CONFIG_POLL` are selected.

Parameters

- `dev` – Pointer to the device structure for the driver instance
- `config` – Pointer to a valid *spi_config* structure instance. Pointer-comparison may be used to detect changes from previous operations.
- `tx_bufs` – Buffer array where data to be sent originates from.
- `sig` – A pointer to a valid and ready to be signaled struct *k_poll_signal*. (Note: if NULL this function will not notify the end of the transaction, and whether it went successfully or not).

Return values

- `0` – If successful.
- `-errno` – Negative errno code on failure.

static inline void **spi_iodev_submit**(struct *rtio_iodev_sqe* *iodev_sqe)

Submit a SPI device with a request.

Parameters

- **iodev_sqe** – Prepared submissions queue entry connected to an iodev defined by SPI_IODEV_DEFINE. Must live as long as the request is in flight.

static inline bool **spi_is_ready_iodev**(const struct *rtio_iodev* *spi_iodev)

Validate that SPI bus (and CS gpio if defined) is ready.

Parameters

- **spi_iodev** – SPI iodev defined with SPI_DT_IODEV_DEFINE

Return values

- **true** – if the SPI bus is ready for use.
- **false** – if the SPI bus (or the CS gpio defined) is not ready for use.

static inline int **spi_rtio_copy**(struct *rtio* *r, struct *rtio_iodev* *iodev, const struct *spi_buf_set* *tx_bufs, const struct *spi_buf_set* *rx_bufs, struct *rtio_sqe* **last_sqe)

Copy the tx_bufs and rx_bufs into a set of RTIO requests.

Parameters

- **r** – **[in]** rtio context
- **iodev** – **[in]** iodev to transceive with
- **tx_bufs** – **[in]** transmit buffer set
- **rx_bufs** – **[in]** receive buffer set
- **last_sqe** – **[out]** last sqe submitted, NULL if not enough memory

Return values

- **Number** – of submission queue entries
- **-ENOMEM** – out of memory

int **spi_release**(const struct *device* *dev, const struct *spi_config* *config)

Release the SPI device locked on and/or the CS by the current config.

Note: This synchronous function is used to release either the lock on the SPI device and/or the CS line that was kept if, and if only, given config parameter was the last one to be used (in any of the above functions) and if it has the SPI_LOCK_ON bit set and/or the SPI_HOLD_ON_CS bit set into its operation bits field. This can be used if the caller needs to keep its hand on the SPI device for consecutive transactions and/or if it needs the device to stay selected. Usually both bits will be used along each other, so the the device is locked and stays on until another operation is necessary or until it gets released with the present function.

Parameters

- **dev** – Pointer to the device structure for the driver instance
- **config** – Pointer to a valid *spi_config* structure instance.

Return values

- **0** – If successful.
- **-errno** – Negative errno code on failure.

static inline int **spi_release_dt**(const struct *spi_dt_spec* *spec)

Release the SPI device specified in *spi_dt_spec*.

This is equivalent to:

```
spi_release(spec->bus, &spec->config);
```

Parameters

- **spec** – SPI specification from devicetree

Returns
a value from *spi_release()*.

Variables

const struct *rtio_iodev_api* **spi_iodev_api**

struct **spi_cs_control**

#include <*spi.h*> SPI Chip Select control structure.

This can be used to control a CS line via a GPIO line, instead of using the controller inner CS logic.

Public Members

struct *gpio_dt_spec* **gpio**

GPIO devicetree specification of CS GPIO.

The device pointer can be set to NULL to fully inhibit CS control if necessary. The GPIO flags GPIO_ACTIVE_LOW/GPIO_ACTIVE_HIGH should be equivalent to SPI_CS_ACTIVE_HIGH/SPI_CS_ACTIVE_LOW options in struct *spi_config*.

uint32_t **delay**

Delay in microseconds to wait before starting the transmission and before releasing the CS line.

struct **spi_config**

#include <*spi.h*> SPI controller configuration structure.

Public Members

uint32_t **frequency**

Bus frequency in Hertz.

spi_operation_t **operation**

Operation flags.

It is a bit field with the following parts:

- 0: Master or slave.
- 1..3: Polarity, phase and loop mode.
- 4: LSB or MSB first.
- 5..10: Size of a data frame in bits.
- 11: Full/half duplex.
- 12: Hold on the CS line if possible.
- 13: Keep resource locked for the caller.
- 14: Active high CS logic.
- 15: Motorola or TI frame format (optional).

If CONFIG_SPI_EXTENDED_MODES is enabled:

- 16..17: MISO lines (Single/Dual/Quad/Octal).
- 18..31: Reserved for future use.

uint16_t **slave**

Slave number from 0 to host controller slave limit.

struct *spi_cs_control* **cs**

GPIO chip-select line (optional, must be initialized to zero if not used).

struct **spi_dt_spec**

#include <*spi.h*> Complete SPI DT information.

Public Members

const struct *device* ***bus**

SPI bus.

struct *spi_config* **config**

Slave specific configuration.

struct **spi_buf**

#include <*spi.h*> SPI buffer structure.

Public Members

void ***buf**

Valid pointer to a data buffer, or NULL otherwise.

size_t **len**

Length of the buffer *buf*.

If *buf* is NULL, length which as to be sent as dummy bytes (as TX buffer) or the length of bytes that should be skipped (as RX buffer).

struct **spi_buf_set**

#include <*spi.h*> SPI buffer array structure.

Public Members

const struct *spi_buf* ***buffers**

Pointer to an array of *spi_buf*, or NULL.

size_t **count**

Length of the array pointed by *buffers*.

struct `spi_driver_api`

#include <*spi.h*> SPI driver API This is the mandatory API any SPI driver needs to expose.

7.6.46 System Management Bus (SMBus)

- *Overview*
- *SMBus Controller API*
- *Configuration Options*
- *API Reference*

Overview

System Management Bus (SMBus) is derived from I2C for communication with devices on the motherboard. A system may use SMBus to communicate with the peripherals on the motherboard without using dedicated control lines. SMBus peripherals can provide various manufacturer information, report errors, accept control parameters, etc.

Devices on the bus can operate in three roles: as a Controller that initiates transactions and controls the clock, a Peripheral that responds to transaction commands, or a Host, which is a specialized Controller, that provides the main interface to the system's CPU. Zephyr has API for the Controller role.

SMBus peripheral devices can initiate communication with Controller with two methods:

- **Host Notify protocol**: Peripheral device that supports the Host Notify protocol behaves as a Controller to perform the notification. It writes a three-bytes message to a special address "SMBus Host (0x08)" with own address and two bytes of relevant data.
- **SMBALERT# signal**: Peripheral device uses special signal SMBALERT# to request attention from the Controller. The Controller needs to read one byte from the special "SMBus Alert Response Address (ARA) (0x0c)". The peripheral device responds with a data byte containing its own address.

Currently, the API is based on SMBus Specification version 2.0

Note

See *Rule A.2: Inclusive Language* for information about the terminology used in this API.

SMBus Controller API

Zephyr's SMBus controller API is used when an SMBus device controls the bus, particularly the start and stop conditions and the clock. This is the most common mode used to interact with SMBus peripherals.

Configuration Options

Related configuration options:

- CONFIG_SMBUS

API Reference

Related code samples

SMBus shell
Interact with SMBus peripherals using shell commands.

group `smbus_interface`

SMBus Interface.

Since
3.4

Version
0.1.0

SMBus read / write direction

enum `smbus_direction`

SMBus read / write direction.

Values:

enumerator `SMBUS_MSG_WRITE` = 0

Write a message to SMBus peripheral.

enumerator `SMBUS_MSG_READ` = 1

Read a message from SMBus peripheral.

SMBus Protocol commands

SMBus Specification defines the following SMBus protocols operations

`SMBUS_CMD_QUICK`

SMBus Quick protocol is a very simple command with no data sent or received.

Peripheral may denote only R/W bit, which can still be used for the peripheral management, for example to switch peripheral On/Off. Quick protocol can also be used for peripheral devices scanning.

```
0                   1
0 1 2 3 4 5 6 7 8 9 0
+-+-+-+-+-+-+-+-+-+-+
|S| Periph Addr |D|A|P|
+-+-+-+-+-+-+-+-+-+-+
```

SMBUS_CMD_BYTE

SMBus Byte protocol can send or receive one byte of data.

```
Byte Write

0                   1                   2
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|S| Periph Addr |W|A| Command code  |A|P|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Byte Read

0                   1                   2
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|S| Periph Addr |R|A| Byte received |N|P|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

SMBUS_CMD_BYTE_DATA

SMBus Byte Data protocol sends the first byte (command) followed by read or write one byte.

```
Byte Data Write

0                   1                   2
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|S| Periph Addr |W|A|  Command code |A|   Data Write  |A|P|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Byte Data Read

0                   1                   2
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|S| Periph Addr |W|A|  Command code |A|S| Periph Addr |R|A|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Data Read   |N|P|
+-+-+-+-+-+-+-+-+-+
```

SMBUS_CMD_WORD_DATA

SMBus Word Data protocol sends the first byte (command) followed by read or write two bytes.

```
Word Data Write

0                   1                   2
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|S| Periph Addr |W|A|  Command code |A| Data Write Low|A|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Data Write Hi |A|P|
+-+-+-+-+-+-+-+-+-+

Word Data Read

0                   1                   2
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7
```

(continues on next page)

(continued from previous page)

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|S| Periph Addr |W|A|  Command code |A|S| Periph Addr |R|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|A| Data Read Low |A|  Data Read Hi |N|P|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

SMBUS_CMD_PROC_CALL

SMBus Process Call protocol is Write Word followed by Read Word.

It is named so because the command sends data and waits for the peripheral to return a reply.

```
0                   1                   2
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|S| Periph Addr |W|A|  Command code |A| Data Write Low|A|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Data Write Hi |A|S| Periph Addr |R|A| Data Read Low |A|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Data Read Hi  |N|P|
+-+-+-+-+-+-+-+-+-+
```

SMBUS_CMD_BLOCK

SMBus Block protocol reads or writes a block of data up to 32 bytes.

The Count byte specifies the amount of data.

```
SMBus Block Write

0                   1                   2
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|S| Periph Addr |W|A|  Command code |A| Send Count=N  |A|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data Write 1 |A|      ...      |A|  Data Write N |A|P|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

SMBus Block Read

0                   1                   2
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|S| Periph Addr |W|A|  Command code |A|S| Periph Addr |R|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|A|  Recv Count=N |A|  Data Read 1  |A|      ...      |A|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data Read N  |N|P|
+-+-+-+-+-+-+-+-+-+
```

SMBUS_CMD_BLOCK_PROC

SMBus Block Write - Block Read Process Call protocol is Block Write followed by Block Read.

```
0                   1                   2
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|S| Periph Addr |W|A|  Command code |A|   Count = N   |A|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

(continues on next page)

(continued from previous page)

```
|  Data Write 1 |A|      ...      |A|  Data Write N |A|S|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Periph Addr |R|A|  Recv Count=N |A|  Data Read 1  |A| |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     ...     |A|  Data Read N  |N|P|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

SMBus device functionality

The following parameters describe the functionality of the SMBus device

`SMBUS_MODE_CONTROLLER`
: Peripheral to act as Controller.

`SMBUS_MODE_PEC`
: Support Packet Error Code (PEC) checking.

`SMBUS_MODE_HOST_NOTIFY`
: Support Host Notify functionality.

`SMBUS_MODE_SMBALERT`
: Support SMBALERT signal functionality.

SMBus special reserved addresses

The following addresses are reserved by SMBus specification

`SMBUS_ADDRESS_ARA`
: Alert Response Address (ARA)

  A broadcast address used by the system host as part of the Alert Response Protocol.

Defines

`SMBUS_BLOCK_BYTES_MAX`
: Maximum number of bytes in SMBus Block protocol.

`SMBUS_DT_SPEC_GET`(node_id)
: Structure initializer for *smbus_dt_spec* from devicetree.

  This helper macro expands to a static initializer for a `struct` *smbus_dt_spec* by reading the relevant bus and address data from the devicetree.

  Parameters

  - **node_id** – Devicetree node identifier for the SMBus device whose struct *smbus_dt_spec* to create an initializer for

`SMBUS_DT_SPEC_INST_GET`(inst)
: Structure initializer for *smbus_dt_spec* from devicetree instance.

  This is equivalent to *SMBUS_DT_SPEC_GET(DT_DRV_INST(inst))*.

Parameters

- `inst` – Devicetree instance number

`SMBUS_DEVICE_DT_DEFINE`(node_id, init_fn, pm_device, data_ptr, cfg_ptr, level, prio, api_ptr, ...)

Like *DEVICE_DT_DEFINE()* with SMBus specifics.

Defines a device which implements the SMBus API. May generate a custom *device_state* container struct and init_fn wrapper when needed depending on SMBus `CONFIG_SMBUS_STATS`.

Parameters

- `node_id` – The devicetree node identifier.
- `init_fn` – Name of the init function of the driver.
- `pm_device` – PM device resources reference (NULL if device does not use PM).
- `data_ptr` – Pointer to the device's private data.
- `cfg_ptr` – The address to the structure containing the configuration information for this instance of the driver.
- `level` – The initialization level. See SYS_INIT() for details.
- `prio` – Priority within the selected initialization level. See SYS_INIT() for details.
- `api_ptr` – Provides an initial pointer to the API function struct used by the driver. Can be NULL.

`SMBUS_DEVICE_DT_INST_DEFINE`(inst, ...)

Like *SMBUS_DEVICE_DT_DEFINE()* for an instance of a DT_DRV_COMPAT compatible.

Parameters

- `inst` – instance number. This is replaced by `DT_DRV_COMPAT(inst)` in the call to *SMBUS_DEVICE_DT_DEFINE()*.
- `...` – other parameters as expected by *SMBUS_DEVICE_DT_DEFINE()*.

Typedefs

typedef void (*`smbus_callback_handler_t`)(const struct *device* *dev, struct *smbus_callback* *cb, uint8_t addr)

Define SMBus callback handler function signature.

Param dev
Pointer to the device structure for the SMBus driver instance.

Param cb
Structure *smbus_callback* owning this handler.

Param addr
Address of the SMBus peripheral device.

Functions

static inline void **smbus_xfer_stats**(const struct *device* *dev, uint8_t sent, uint8_t recv)

Updates the SMBus stats.

Parameters

- **dev** – Pointer to the device structure for the SMBus driver instance to update stats for.
- **sent** – Number of bytes sent
- **recv** – Number of bytes received

int **smbus_configure**(const struct *device* *dev, uint32_t dev_config)

Configure operation of a SMBus host controller.

Parameters

- **dev** – Pointer to the device structure for the SMBus driver instance.
- **dev_config** – Bit-packed 32-bit value to the device runtime configuration for the SMBus controller.

Return values

- **0** – If successful.
- **-EIO** – General input / output error.

int **smbus_get_config**(const struct *device* *dev, uint32_t *dev_config)

Get configuration of a SMBus host controller.

This routine provides a way to get current configuration. It is allowed to call the function before smbus_configure, because some SMBus ports can be configured during init process. However, if the SMBus port is not configured, smbus_get_config returns an error.

smbus_get_config can return cached config or probe hardware, but it has to be up to date with current configuration.

Parameters

- **dev** – Pointer to the device structure for the SMBus driver instance.
- **dev_config** – Pointer to return bit-packed 32-bit value of the SMBus controller configuration.

Return values

- **0** – If successful.
- **-EIO** – General input / output error.
- **-ENOSYS** – If function *smbus_get_config()* is not implemented by the driver.

static inline int **smbus_smbalert_set_cb**(const struct *device* *dev, struct *smbus_callback* *cb)

Add SMBUSALERT callback for a SMBus host controller.

Parameters

- **dev** – Pointer to the device structure for the SMBus driver instance.
- **cb** – Pointer to a callback structure.

Return values

- **0** – If successful.
- **-EIO** – General input / output error.
- **-ENOSYS** – If function *smbus_smbalert_set_cb()* is not implemented by the driver.

int **smbus_smbalert_remove_cb**(const struct *device* *dev, struct *smbus_callback* *cb)

Remove SMBUSALERT callback from a SMBus host controller.

Parameters

- **dev** – Pointer to the device structure for the SMBus driver instance.
- **cb** – Pointer to a callback structure.

Return values

- **0** – If successful.
- **-EIO** – General input / output error.
- **-ENOSYS** – If function *smbus_smbalert_remove_cb()* is not implemented by the driver.

static inline int **smbus_host_notify_set_cb**(const struct *device* *dev, struct *smbus_callback* *cb)

Add Host Notify callback for a SMBus host controller.

Parameters

- **dev** – Pointer to the device structure for the SMBus driver instance.
- **cb** – Pointer to a callback structure.

Return values

- **0** – If successful.
- **-EIO** – General input / output error.
- **-ENOSYS** – If function *smbus_host_notify_set_cb()* is not implemented by the driver.

int **smbus_host_notify_remove_cb**(const struct *device* *dev, struct *smbus_callback* *cb)

Remove Host Notify callback from a SMBus host controller.

Parameters

- **dev** – Pointer to the device structure for the SMBus driver instance.
- **cb** – Pointer to a callback structure.

Return values

- **0** – If successful.
- **-EIO** – General input / output error.
- **-ENOSYS** – If function *smbus_host_notify_remove_cb()* is not implemented by the driver.

int **smbus_quick**(const struct *device* *dev, uint16_t addr, enum *smbus_direction* direction)

Perform SMBus Quick operation.

This routine provides a generic interface to perform SMBus Quick operation.

Parameters

- **dev** – Pointer to the device structure for the SMBus driver instance. driver configured in controller mode.
- **addr** – Address of the SMBus peripheral device.
- **direction** – Direction Read or Write.

Return values

- **0** – If successful.

- `-EIO` – General input / output error.
- `-ENOSYS` – If function *smbus_quick()* is not implemented by the driver.

int `smbus_byte_write`(const struct *device* *dev, uint16_t addr, uint8_t byte)

Perform SMBus Byte Write operation.

This routine provides a generic interface to perform SMBus Byte Write operation.

Parameters

- `dev` – Pointer to the device structure for the SMBus driver instance.
- `addr` – Address of the SMBus peripheral device.
- `byte` – Byte to be sent to the peripheral device.

Return values

- `0` – If successful.
- `-EIO` – General input / output error.
- `-ENOSYS` – If function *smbus_byte_write()* is not implemented by the driver.

int `smbus_byte_read`(const struct *device* *dev, uint16_t addr, uint8_t *byte)

Perform SMBus Byte Read operation.

This routine provides a generic interface to perform SMBus Byte Read operation.

Parameters

- `dev` – Pointer to the device structure for the SMBus driver instance.
- `addr` – Address of the SMBus peripheral device.
- `byte` – Byte received from the peripheral device.

Return values

- `0` – If successful.
- `-EIO` – General input / output error.
- `-ENOSYS` – If function *smbus_byte_read()* is not implemented by the driver.

int `smbus_byte_data_write`(const struct *device* *dev, uint16_t addr, uint8_t cmd, uint8_t byte)

Perform SMBus Byte Data Write operation.

This routine provides a generic interface to perform SMBus Byte Data Write operation.

Parameters

- `dev` – Pointer to the device structure for the SMBus driver instance.
- `addr` – Address of the SMBus peripheral device.
- `cmd` – Command byte which is sent to peripheral device first.
- `byte` – Byte to be sent to the peripheral device.

Return values

- `0` – If successful.
- `-EIO` – General input / output error.
- `-ENOSYS` – If function *smbus_byte_data_write()* is not implemented by the driver.

int **smbus_byte_data_read**(const struct *device* *dev, uint16_t addr, uint8_t cmd, uint8_t *byte)

Perform SMBus Byte Data Read operation.

This routine provides a generic interface to perform SMBus Byte Data Read operation.

Parameters

- `dev` – Pointer to the device structure for the SMBus driver instance.
- `addr` – Address of the SMBus peripheral device.
- `cmd` – Command byte which is sent to peripheral device first.
- `byte` – Byte received from the peripheral device.

Return values

- `0` – If successful.
- `-EIO` – General input / output error.
- `-ENOSYS` – If function *smbus_byte_data_read()* is not implemented by the driver.

int **smbus_word_data_write**(const struct *device* *dev, uint16_t addr, uint8_t cmd, uint16_t word)

Perform SMBus Word Data Write operation.

This routine provides a generic interface to perform SMBus Word Data Write operation.

Parameters

- `dev` – Pointer to the device structure for the SMBus driver instance.
- `addr` – Address of the SMBus peripheral device.
- `cmd` – Command byte which is sent to peripheral device first.
- `word` – Word (16-bit) to be sent to the peripheral device.

Return values

- `0` – If successful.
- `-EIO` – General input / output error.
- `-ENOSYS` – If function *smbus_word_data_write()* is not implemented by the driver.

int **smbus_word_data_read**(const struct *device* *dev, uint16_t addr, uint8_t cmd, uint16_t *word)

Perform SMBus Word Data Read operation.

This routine provides a generic interface to perform SMBus Word Data Read operation.

Parameters

- `dev` – Pointer to the device structure for the SMBus driver instance.
- `addr` – Address of the SMBus peripheral device.
- `cmd` – Command byte which is sent to peripheral device first.
- `word` – Word (16-bit) received from the peripheral device.

Return values

- `0` – If successful.
- `-EIO` – General input / output error.

- `-ENOSYS` – If function *smbus_word_data_read()* is not implemented by the driver.

int `smbus_pcall`(const struct *device* *dev, uint16_t addr, uint8_t cmd, uint16_t send_word, uint16_t *recv_word)

Perform SMBus Process Call operation.

This routine provides a generic interface to perform SMBus Process Call operation, which means Write 2 bytes following by Read 2 bytes.

Parameters

- `dev` – Pointer to the device structure for the SMBus driver instance.
- `addr` – Address of the SMBus peripheral device.
- `cmd` – Command byte which is sent to peripheral device first.
- `send_word` – Word (16-bit) to be sent to the peripheral device.
- `recv_word` – Word (16-bit) received from the peripheral device.

Return values

- `0` – If successful.
- `-EIO` – General input / output error.
- `-ENOSYS` – If function *smbus_pcall()* is not implemented by the driver.

int `smbus_block_write`(const struct *device* *dev, uint16_t addr, uint8_t cmd, uint8_t count, uint8_t *buf)

Perform SMBus Block Write operation.

This routine provides a generic interface to perform SMBus Block Write operation.

Parameters

- `dev` – Pointer to the device structure for the SMBus driver instance.
- `addr` – Address of the SMBus peripheral device.
- `cmd` – Command byte which is sent to peripheral device first.
- `count` – Size of the data block buffer. Maximum 32 bytes.
- `buf` – Data block buffer to be sent to the peripheral device.

Return values

- `0` – If successful.
- `-EIO` – General input / output error.
- `-ENOSYS` – If function *smbus_block_write()* is not implemented by the driver.

int `smbus_block_read`(const struct *device* *dev, uint16_t addr, uint8_t cmd, uint8_t *count, uint8_t *buf)

Perform SMBus Block Read operation.

This routine provides a generic interface to perform SMBus Block Read operation.

Parameters

- `dev` – Pointer to the device structure for the SMBus driver instance.
- `addr` – Address of the SMBus peripheral device.
- `cmd` – Command byte which is sent to peripheral device first.
- `count` – Size of the data peripheral sent. Maximum 32 bytes.

- `buf` – Data block buffer received from the peripheral device.

Return values

- `0` – If successful.
- `-EIO` – General input / output error.
- `-ENOSYS` – If function *smbus_block_read()* is not implemented by the driver.

int `smbus_block_pcall`(const struct *device* *dev, uint16_t addr, uint8_t cmd, uint8_t snd_count, uint8_t *snd_buf, uint8_t *rcv_count, uint8_t *rcv_buf)

Perform SMBus Block Process Call operation.

This routine provides a generic interface to perform SMBus Block Process Call operation. This operation is basically Block Write followed by Block Read.

Parameters

- `dev` – Pointer to the device structure for the SMBus driver instance.
- `addr` – Address of the SMBus peripheral device.
- `cmd` – Command byte which is sent to peripheral device first.
- `snd_count` – Size of the data block buffer to send.
- `snd_buf` – Data block buffer send to the peripheral device.
- `rcv_count` – Size of the data peripheral sent.
- `rcv_buf` – Data block buffer received from the peripheral device.

Return values

- `0` – If successful.
- `-EIO` – General input / output error.
- `-ENOSYS` – If function *smbus_block_pcall()* is not implemented by the driver.

struct `smbus_callback`

#include <*smbus.h*> SMBus callback structure.

Used to register a callback in the driver instance callback list. As many callbacks as needed can be added as long as each of them is a unique pointer of struct *smbus_callback*.

Note: Such struct should not be allocated on stack.

Public Members

sys_snode_t `node`

This should be used in driver for a callback list management.

smbus_callback_handler_t `handler`

Actual callback function being called when relevant.

uint8_t `addr`

Peripheral device address.

struct **smbus_dt_spec**

#include <*smbus.h*> Complete SMBus DT information.

Public Members

const struct *device* ***bus**

SMBus bus.

uint16_t **addr**

Address of the SMBus peripheral device.

7.6.47 Universal Asynchronous Receiver-Transmitter (UART)

Overview

Zephyr provides three different ways to access the UART peripheral. Depending on the method, different API functions are used according to below sections:

1. *Polling API*
2. *Interrupt-driven API*
3. *Asynchronous API* using *Direct Memory Access (DMA)*

Polling is the most basic method to access the UART peripheral. The reading function, *uart_poll_in*, is a non-blocking function and returns a character or *-1* when no valid data is available. The writing function, *uart_poll_out*, is a blocking function and the thread waits until the given character is sent.

With the Interrupt-driven API, possibly slow communication can happen in the background while the thread continues with other tasks. The Kernel's *Data Passing* features can be used to communicate between the thread and the UART driver.

The Asynchronous API allows to read and write data in the background using DMA without interrupting the MCU at all. However, the setup is more complex than the other methods.

Warning

Interrupt-driven API and the Asynchronous API should NOT be used at the same time for the same hardware peripheral, since both APIs require hardware interrupts to function properly. Using the callbacks for both APIs would result in interference between each other. `CONFIG_UART_EXCLUSIVE_API_CALLBACKS` is enabled by default so that only the callbacks associated with one API is active at a time.

Configuration Options

Most importantly, the Kconfig options define whether the polling API (default), the interrupt-driven API or the asynchronous API can be used. Only enable the features you need in order to minimize memory footprint.

Related configuration options:

- `CONFIG_SERIAL`
- `CONFIG_UART_INTERRUPT_DRIVEN`

- CONFIG_UART_ASYNC_API
- CONFIG_UART_WIDE_DATA
- CONFIG_UART_USE_RUNTIME_CONFIGURE
- CONFIG_UART_LINE_CTRL
- CONFIG_UART_DRV_CMD

API Reference

Related code samples

802.15.4 "serial-radio"
Implement a slip-radio device for Contiki-based border routers.

Native TTY UART
Use native TTY driver to send and receive data between two UART-to-USB bridge dongles.

STM32 single-wire UART
Use single-wire/half-duplex UART functionality of STM32 devices.

UART Passthrough
Pass data directly between the console and another UART interface.

UART echo
Read data from the console and echo it back.

USB CDC-ACM
Use USB CDC-ACM driver to implement a serial port echo.

group `uart_interface`

UART Interface.

Since
1.0

Version
1.0.0

Enums

enum `uart_line_ctrl`

Line control signals.

Values:

enumerator `UART_LINE_CTRL_BAUD_RATE` = *BIT*(0)

Baud rate.

enumerator `UART_LINE_CTRL_RTS` = *BIT*(1)

Request To Send (RTS)

enumerator UART_LINE_CTRL_DTR = *BIT*(2)

Data Terminal Ready (DTR)

enumerator UART_LINE_CTRL_DCD = *BIT*(3)

Data Carrier Detect (DCD)

enumerator UART_LINE_CTRL_DSR = *BIT*(4)

Data Set Ready (DSR)

enum uart_rx_stop_reason

Reception stop reasons.

Values that correspond to events or errors responsible for stopping receiving.

Values:

enumerator UART_ERROR_OVERRUN = (1 « 0)

Overrun error.

enumerator UART_ERROR_PARITY = (1 « 1)

Parity error.

enumerator UART_ERROR_FRAMING = (1 « 2)

Framing error.

enumerator UART_BREAK = (1 « 3)

Break interrupt.

A break interrupt was received. This happens when the serial input is held at a logic '0' state for longer than the sum of start time + data bits + parity + stop bits.

enumerator UART_ERROR_COLLISION = (1 « 4)

Collision error.

This error is raised when transmitted data does not match received data. Typically this is useful in scenarios where the TX and RX lines maybe connected together such as RS-485 half-duplex. This error is only valid on UARTs that support collision checking.

enumerator UART_ERROR_NOISE = (1 « 5)

Noise error.

enum uart_config_parity

Parity modes.

Values:

enumerator UART_CFG_PARITY_NONE

No parity.

enumerator UART_CFG_PARITY_ODD

Odd parity.

enumerator UART_CFG_PARITY_EVEN
Even parity.

enumerator UART_CFG_PARITY_MARK
Mark parity.

enumerator UART_CFG_PARITY_SPACE
Space parity.

enum uart_config_stop_bits
Number of stop bits.

Values:

enumerator UART_CFG_STOP_BITS_0_5
0.5 stop bit

enumerator UART_CFG_STOP_BITS_1
1 stop bit

enumerator UART_CFG_STOP_BITS_1_5
1.5 stop bits

enumerator UART_CFG_STOP_BITS_2
2 stop bits

enum uart_config_data_bits
Number of data bits.

Values:

enumerator UART_CFG_DATA_BITS_5
5 data bits

enumerator UART_CFG_DATA_BITS_6
6 data bits

enumerator UART_CFG_DATA_BITS_7
7 data bits

enumerator UART_CFG_DATA_BITS_8
8 data bits

enumerator UART_CFG_DATA_BITS_9
9 data bits

enum uart_config_flow_control
Hardware flow control options.

With flow control set to none, any operations related to flow control signals can be managed by user with uart_line_ctrl functions. In other cases, flow control is managed by hardware/driver.

Values:

enumerator `UART_CFG_FLOW_CTRL_NONE`
No flow control.

enumerator `UART_CFG_FLOW_CTRL_RTS_CTS`
RTS/CTS flow control.

enumerator `UART_CFG_FLOW_CTRL_DTR_DSR`
DTR/DSR flow control.

enumerator `UART_CFG_FLOW_CTRL_RS485`
RS485 flow control.

Functions

int `uart_err_check`(const struct *device* *dev)
Check whether an error was detected.

Parameters

- `dev` – UART device instance.

Return values

- `0` – If no error was detected.
- `err` – Error flags as defined in *uart_rx_stop_reason*
- `-ENOSYS` – If not implemented.

int `uart_configure`(const struct *device* *dev, const struct *uart_config* *cfg)
Set UART configuration.

Sets UART configuration using data from *cfg.

Parameters

- `dev` – UART device instance.
- `cfg` – UART configuration structure.

Return values

- `0` – If successful.
- `-errno` – Negative errno code in case of failure.
- `-ENOSYS` – If configuration is not supported by device or driver does not support setting configuration in runtime.
- `-ENOTSUP` – If API is not enabled.

int `uart_config_get`(const struct *device* *dev, struct *uart_config* *cfg)
Get UART configuration.

Stores current UART configuration to *cfg, can be used to retrieve initial configuration after device was initialized using data from DTS.

Parameters

- `dev` – UART device instance.
- `cfg` – UART configuration structure.

Return values

- `0` – If successful.
- `-errno` – Negative errno code in case of failure.
- `-ENOSYS` – If driver does not support getting current configuration.
- `-ENOTSUP` – If API is not enabled.

int `uart_line_ctrl_set`(const struct *device* *dev, uint32_t ctrl, uint32_t val)

Manipulate line control for UART.

Parameters

- `dev` – UART device instance.
- `ctrl` – The line control to manipulate (see enum uart_line_ctrl).
- `val` – Value to set to the line control.

Return values

- `0` – If successful.
- `-ENOSYS` – If this function is not implemented.
- `-ENOTSUP` – If API is not enabled.
- `-errno` – Other negative errno value in case of failure.

int `uart_line_ctrl_get`(const struct *device* *dev, uint32_t ctrl, uint32_t *val)

Retrieve line control for UART.

Parameters

- `dev` – UART device instance.
- `ctrl` – The line control to retrieve (see enum uart_line_ctrl).
- `val` – Pointer to variable where to store the line control value.

Return values

- `0` – If successful.
- `-ENOSYS` – If this function is not implemented.
- `-ENOTSUP` – If API is not enabled.
- `-errno` – Other negative errno value in case of failure.

int `uart_drv_cmd`(const struct *device* *dev, uint32_t cmd, uint32_t p)

Send extra command to driver.

Implementation and accepted commands are driver specific. Refer to the drivers for more information.

Parameters

- `dev` – UART device instance.
- `cmd` – Command to driver.
- `p` – Parameter to the command.

Return values

- `0` – If successful.
- `-ENOSYS` – If this function is not implemented.
- `-ENOTSUP` – If API is not enabled.
- `-errno` – Other negative errno value in case of failure.

struct `uart_config`

#include <*uart.h*> UART controller configuration structure.

Public Members

uint32_t `baudrate`

Baudrate setting in bps.

uint8_t `parity`

Parity bit, use *uart_config_parity*.

uint8_t `stop_bits`

Stop bits, use *uart_config_stop_bits*.

uint8_t `data_bits`

Data bits, use *uart_config_data_bits*.

uint8_t `flow_ctrl`

Flow control setting, use *uart_config_flow_control*.

Polling API

group `uart_polling`

Functions

int `uart_poll_in`(const struct *device* *dev, unsigned char *p_char)

Read a character from the device for input.

This routine checks if the receiver has valid data. When the receiver has valid data, it reads a character from the device, stores to the location pointed to by p_char, and returns 0 to the calling thread. It returns -1, otherwise. This function is a non-blocking call.

Parameters

- `dev` – UART device instance.
- `p_char` – Pointer to character.

Return values

- `0` – If a character arrived.
- `-1` – If no character was available to read (i.e. the UART input buffer was empty).
- `-ENOSYS` – If the operation is not implemented.
- `-EBUSY` – If async reception was enabled using *uart_rx_enable*

int `uart_poll_in_u16`(const struct *device* *dev, uint16_t *p_u16)

Read a 16-bit datum from the device for input.

This routine checks if the receiver has valid data. When the receiver has valid data, it reads a 16-bit datum from the device, stores to the location pointed to by p_u16, and returns 0 to the calling thread. It returns -1, otherwise. This function is a non-blocking call.

Parameters

- `dev` – UART device instance.
- `p_u16` – Pointer to 16-bit data.

Return values

- `0` – If data arrived.
- `-1` – If no data was available to read (i.e., the UART input buffer was empty).
- `-ENOTSUP` – If API is not enabled.
- `-ENOSYS` – If the function is not implemented.
- `-EBUSY` – If async reception was enabled using *uart_rx_enable*

void `uart_poll_out`(const struct *device* *dev, unsigned char out_char)

Write a character to the device for output.

This routine checks if the transmitter is full. When the transmitter is not full, it writes a character to the data register. It waits and blocks the calling thread, otherwise. This function is a blocking call.

To send a character when hardware flow control is enabled, the handshake signal CTS must be asserted.

Parameters

- `dev` – UART device instance.
- `out_char` – Character to send.

void `uart_poll_out_u16`(const struct *device* *dev, uint16_t out_u16)

Write a 16-bit datum to the device for output.

This routine checks if the transmitter is full. When the transmitter is not full, it writes a 16-bit datum to the data register. It waits and blocks the calling thread, otherwise. This function is a blocking call.

To send a datum when hardware flow control is enabled, the handshake signal CTS must be asserted.

Parameters

- `dev` – UART device instance.
- `out_u16` – Wide data to send.

Interrupt-driven API

group `uart_interrupt`

Typedefs

typedef void (*`uart_irq_callback_user_data_t`)(const struct *device* *dev, void *user_data)

Define the application callback function signature for *uart_irq_callback_user_data_set()* function.

Param dev
UART device instance.

Param user_data
Arbitrary user data.

typedef void (*`uart_irq_config_func_t`)(const struct *device* *dev)

For configuring IRQ on each individual UART device.

Param dev
UART device instance.

Functions

static inline int `uart_fifo_fill`(const struct *device* *dev, const uint8_t *tx_data, int size)

Fill FIFO with data.

This function is expected to be called from UART interrupt handler (ISR), if *uart_irq_tx_ready()* returns true. Result of calling this function not from an ISR is undefined (hardware-dependent). Likewise, *not* calling this function from an ISR if *uart_irq_tx_ready()* returns true may lead to undefined behavior, e.g. infinite interrupt loops. It's mandatory to test return value of this function, as different hardware has different FIFO depth (oftentimes just 1).

Parameters

- `dev` – UART device instance.
- `tx_data` – Data to transmit.
- `size` – Number of bytes to send.

Return values

- `-ENOSYS` – if this function is not supported
- `-ENOTSUP` – If API is not enabled.

Returns
Number of bytes sent.

static inline int `uart_fifo_fill_u16`(const struct *device* *dev, const uint16_t *tx_data, int size)

Fill FIFO with wide data.

This function is expected to be called from UART interrupt handler (ISR), if *uart_irq_tx_ready()* returns true. Result of calling this function not from an ISR is undefined (hardware-dependent). Likewise, *not* calling this function from an ISR if *uart_irq_tx_ready()* returns true may lead to undefined behavior, e.g. infinite interrupt loops. It's mandatory to test return value of this function, as different hardware has different FIFO depth (oftentimes just 1).

Parameters

- `dev` – UART device instance.
- `tx_data` – Wide data to transmit.

- `size` – Number of datum to send.

Return values

- `-ENOSYS` – If this function is not implemented
- `-ENOTSUP` – If API is not enabled.

Returns
Number of datum sent.

static inline int `uart_fifo_read`(const struct *device* *dev, uint8_t *rx_data, const int size)

Read data from FIFO.

This function is expected to be called from UART interrupt handler (ISR), if *uart_irq_rx_ready()* returns true. Result of calling this function not from an ISR is undefined (hardware-dependent). It's unspecified whether "RX ready" condition as returned by *uart_irq_rx_ready()* is level- or edge- triggered. That means that once *uart_irq_rx_ready()* is detected, *uart_fifo_read()* must be called until it reads all available data in the FIFO (i.e. until it returns less data than was requested).

Parameters

- `dev` – UART device instance.
- `rx_data` – Data container.
- `size` – Container size.

Return values

- `-ENOSYS` – If this function is not implemented.
- `-ENOTSUP` – If API is not enabled.

Returns
Number of bytes read.

static inline int `uart_fifo_read_u16`(const struct *device* *dev, uint16_t *rx_data, const int size)

Read wide data from FIFO.

This function is expected to be called from UART interrupt handler (ISR), if *uart_irq_rx_ready()* returns true. Result of calling this function not from an ISR is undefined (hardware-dependent). It's unspecified whether "RX ready" condition as returned by *uart_irq_rx_ready()* is level- or edge- triggered. That means that once *uart_irq_rx_ready()* is detected, *uart_fifo_read()* must be called until it reads all available data in the FIFO (i.e. until it returns less data than was requested).

Parameters

- `dev` – UART device instance.
- `rx_data` – Wide data container.
- `size` – Container size.

Return values

- `-ENOSYS` – If this function is not implemented.
- `-ENOTSUP` – If API is not enabled.

Returns
Number of datum read.

void `uart_irq_tx_enable`(const struct *device* *dev)

Enable TX interrupt in IER.

Parameters

- `dev` – UART device instance.

void `uart_irq_tx_disable`(const struct *device* *dev)

Disable TX interrupt in IER.

Parameters

- `dev` – UART device instance.

static inline int `uart_irq_tx_ready`(const struct *device* *dev)

Check if UART TX buffer can accept a new char.

Check if UART TX buffer can accept at least one character for transmission (i.e. *uart_fifo_fill()* will succeed and return non-zero). This function must be called in a UART interrupt handler, or its result is undefined. Before calling this function in the interrupt handler, *uart_irq_update()* must be called once per the handler invocation.

Parameters

- `dev` – UART device instance.

Return values

- `1` – If TX interrupt is enabled and at least one char can be written to UART.
- `0` – If device is not ready to write a new byte.
- `-ENOSYS` – If this function is not implemented.
- `-ENOTSUP` – If API is not enabled.

void `uart_irq_rx_enable`(const struct *device* *dev)

Enable RX interrupt.

Parameters

- `dev` – UART device instance.

void `uart_irq_rx_disable`(const struct *device* *dev)

Disable RX interrupt.

Parameters

- `dev` – UART device instance.

static inline int `uart_irq_tx_complete`(const struct *device* *dev)

Check if UART TX block finished transmission.

Check if any outgoing data buffered in UART TX block was fully transmitted and TX block is idle. When this condition is true, UART device (or whole system) can be power off. Note that this function is *not* useful to check if UART TX can accept more data, use *uart_irq_tx_ready()* for that. This function must be called in a UART interrupt handler, or its result is undefined. Before calling this function in the interrupt handler, *uart_irq_update()* must be called once per the handler invocation.

Parameters

- `dev` – UART device instance.

Return values

- `1` – If nothing remains to be transmitted.
- `0` – If transmission is not completed.
- `-ENOSYS` – If this function is not implemented.
- `-ENOTSUP` – If API is not enabled.

static inline int `uart_irq_rx_ready`(const struct *device* *dev)

Check if UART RX buffer has a received char.

Check if UART RX buffer has at least one pending character (i.e. *uart_fifo_read()* will succeed and return non-zero). This function must be called in a UART interrupt handler, or its result is undefined. Before calling this function in the interrupt handler, *uart_irq_update()* must be called once per the handler invocation. It's unspecified whether condition as returned by this function is level- or edge- triggered (i.e. if this function returns true when RX FIFO is non-empty, or when a new char was received since last call to it). See description of *uart_fifo_read()* for implication of this.

Parameters

- `dev` – UART device instance.

Return values

- `1` – If a received char is ready.
- `0` – If a received char is not ready.
- `-ENOSYS` – If this function is not implemented.
- `-ENOTSUP` – If API is not enabled.

void `uart_irq_err_enable`(const struct *device* *dev)

Enable error interrupt.

Parameters

- `dev` – UART device instance.

void `uart_irq_err_disable`(const struct *device* *dev)

Disable error interrupt.

Parameters

- `dev` – UART device instance.

int `uart_irq_is_pending`(const struct *device* *dev)

Check if any IRQs is pending.

Parameters

- `dev` – UART device instance.

Return values

- `1` – If an IRQ is pending.
- `0` – If an IRQ is not pending.
- `-ENOSYS` – If this function is not implemented.
- `-ENOTSUP` – If API is not enabled.

int `uart_irq_update`(const struct *device* *dev)

Start processing interrupts in ISR.

This function should be called the first thing in the ISR. Calling *uart_irq_rx_ready()*, *uart_irq_tx_ready()*, *uart_irq_tx_complete()* allowed only after this.

The purpose of this function is:

- For devices with auto-acknowledge of interrupt status on register read to cache the value of this register (rx_ready, etc. then use this case).
- For devices with explicit acknowledgment of interrupts, to ack any pending interrupts and likewise to cache the original value.

- For devices with implicit acknowledgment, this function will be empty. But the ISR must perform the actions needs to ack the interrupts (usually, call *uart_fifo_read()* on rx_ready, and *uart_fifo_fill()* on tx_ready).

Parameters

- `dev` – UART device instance.

Return values

- `1` – On success.
- `-ENOSYS` – If this function is not implemented.
- `-ENOTSUP` – If API is not enabled.

static inline int `uart_irq_callback_user_data_set`(const struct *device* *dev, *uart_irq_callback_user_data_t* cb, void *user_data)

Set the IRQ callback function pointer.

This sets up the callback for IRQ. When an IRQ is triggered, the specified function will be called with specified user data. See description of *uart_irq_update()* for the requirements on ISR.

Parameters

- `dev` – UART device instance.
- `cb` – Pointer to the callback function.
- `user_data` – Data to pass to callback function.

Return values

- `0` – On success.
- `-ENOSYS` – If this function is not implemented.
- `-ENOTSUP` – If API is not enabled.

static inline int `uart_irq_callback_set`(const struct *device* *dev, *uart_irq_callback_user_data_t* cb)

Set the IRQ callback function pointer (legacy).

This sets up the callback for IRQ. When an IRQ is triggered, the specified function will be called with the device pointer.

Parameters

- `dev` – UART device instance.
- `cb` – Pointer to the callback function.

Return values

- `0` – On success.
- `-ENOSYS` – If this function is not implemented.
- `-ENOTSUP` – If API is not enabled.

Asynchronous API

group `uart_async`

Since
1.14

Version
0.8.0

Typedefs

typedef void (*`uart_callback_t`)(const struct *device* *dev, struct *uart_event* *evt, void *user_data)

Define the application callback function signature for *uart_callback_set()* function.

Param dev
UART device instance.

Param evt
Pointer to *uart_event* instance.

Param user_data
Pointer to data specified by user.

Enums

enum `uart_event_type`

Types of events passed to callback in UART_ASYNC_API.

Receiving:

a. To start receiving, uart_rx_enable has to be called with first buffer

b. When receiving starts to current buffer, *UART_RX_BUF_REQUEST* will be generated, in response to that user can either:

- Provide second buffer using uart_rx_buf_rsp, when first buffer is filled, receiving will automatically start to second buffer.
- Ignore the event, this way when current buffer is filled *UART_RX_RDY* event will be generated and receiving will be stopped.

c. If some data was received and timeout occurred *UART_RX_RDY* event will be generated. It can happen multiples times for the same buffer. RX timeout is counted from last byte received i.e. if no data was received, there won't be any timeout event.

d. *UART_RX_BUF_RELEASED* event will be generated when the current buffer is no longer used by the driver. It will immediately follow *UART_RX_RDY* event. Depending on the implementation buffer may be released when it is completely or partially filled.

e. If there was second buffer provided, it will become current buffer and we start again at point 2. If no second buffer was specified receiving is stopped and *UART_RX_DISABLED* event is generated. After that whole process can be repeated.

Any time during reception *UART_RX_STOPPED* event can occur. if there is any data received, *UART_RX_RDY* event will be generated. It will be followed by *UART_RX_BUF_RELEASED* event for every buffer currently passed to driver and finally by *UART_RX_DISABLED* event.

Receiving can be disabled using uart_rx_disable, after calling that function, if there is any data received, *UART_RX_RDY* event will be generated. *UART_RX_BUF_RELEASED* event will be generated for every buffer currently passed to driver and finally *UART_RX_DISABLED* event will occur.

Transmitting:

a. Transmitting starts by uart_tx function.

b. If whole buffer was transmitted *UART_TX_DONE* is generated. If timeout occurred *UART_TX_ABORTED* will be generated.

Transmitting can be aborted using *uart_tx_abort*, after calling that function *UART_TX_ABORTED* event will be generated.

Values:

enumerator `UART_TX_DONE`

Whole TX buffer was transmitted.

enumerator `UART_TX_ABORTED`

Transmitting aborted due to timeout or uart_tx_abort call.

When flow control is enabled, there is a possibility that TX transfer won't finish in the allotted time. Some data may have been transferred, information about it can be found in event data.

enumerator `UART_RX_RDY`

Received data is ready for processing.

This event is generated in the following cases:
- When RX timeout occurred, and data was stored in provided buffer. This can happen multiple times in the same buffer.
- When provided buffer is full.
- After *uart_rx_disable()*.
- After stopping due to external event (*UART_RX_STOPPED*).

enumerator `UART_RX_BUF_REQUEST`

Driver requests next buffer for continuous reception.

This event is triggered when receiving has started for a new buffer, i.e. it's time to provide a next buffer for a seamless switchover to it. For continuous reliable receiving, user should provide another RX buffer in response to this event, using uart_rx_buf_rsp function

If uart_rx_buf_rsp is not called before current buffer is filled up, receiving will stop.

enumerator `UART_RX_BUF_RELEASED`

Buffer is no longer used by UART driver.

enumerator `UART_RX_DISABLED`

RX has been disabled and can be reenabled.

This event is generated whenever receiver has been stopped, disabled or finished its operation and can be enabled again using uart_rx_enable

enumerator `UART_RX_STOPPED`

RX has stopped due to external event.

Reason is one of uart_rx_stop_reason.

Functions

static inline int **uart_callback_set**(const struct *device* *dev, *uart_callback_t* callback, void *user_data)

Set event handler function.

Since it is mandatory to set callback to use other asynchronous functions, it can be used to detect if the device supports asynchronous API. Remaining API does not have that detection.

Parameters

- **dev** – UART device instance.
- **callback** – Event handler.
- **user_data** – Data to pass to event handler function.

Return values

- **0** – If successful.
- **-ENOSYS** – If not supported by the device.
- **-ENOTSUP** – If API not enabled.

int **uart_tx**(const struct *device* *dev, const uint8_t *buf, size_t len, int32_t timeout)

Send given number of bytes from buffer through UART.

Function returns immediately and event handler, set using *uart_callback_set*, is called after transfer is finished.

Parameters

- **dev** – UART device instance.
- **buf** – Pointer to transmit buffer.
- **len** – Length of transmit buffer.
- **timeout** – Timeout in microseconds. Valid only if flow control is enabled. SYS_FOREVER_US disables timeout.

Return values

- **0** – If successful.
- **-ENOTSUP** – If API is not enabled.
- **-EBUSY** – If There is already an ongoing transfer.
- **-errno** – Other negative errno value in case of failure.

int **uart_tx_u16**(const struct *device* *dev, const uint16_t *buf, size_t len, int32_t timeout)

Send given number of datum from buffer through UART.

Function returns immediately and event handler, set using *uart_callback_set*, is called after transfer is finished.

Parameters

- **dev** – UART device instance.
- **buf** – Pointer to wide data transmit buffer.
- **len** – Length of wide data transmit buffer.
- **timeout** – Timeout in milliseconds. Valid only if flow control is enabled. SYS_FOREVER_MS disables timeout.

Return values

- **0** – If successful.
- **-ENOTSUP** – If API is not enabled.

- `-EBUSY` – If there is already an ongoing transfer.
- `-errno` – Other negative errno value in case of failure.

int `uart_tx_abort`(const struct *device* *dev)

Abort current TX transmission.

UART_TX_DONE event will be generated with amount of data sent.

Parameters

- `dev` – UART device instance.

Return values

- `0` – If successful.
- `-ENOTSUP` – If API is not enabled.
- `-EFAULT` – There is no active transmission.
- `-errno` – Other negative errno value in case of failure.

int `uart_rx_enable`(const struct *device* *dev, uint8_t *buf, size_t len, int32_t timeout)

Start receiving data through UART.

Function sets given buffer as first buffer for receiving and returns immediately. After that event handler, set using *uart_callback_set*, is called with *UART_RX_RDY* or *UART_RX_BUF_REQUEST* events.

Parameters

- `dev` – UART device instance.
- `buf` – Pointer to receive buffer.
- `len` – Buffer length.
- `timeout` – Inactivity period after receiving at least a byte which triggers *UART_RX_RDY* event. Given in microseconds. SYS_FOREVER_US disables timeout. See *uart_event_type* for details.

Return values

- `0` – If successful.
- `-ENOTSUP` – If API is not enabled.
- `-EBUSY` – RX already in progress.
- `-errno` – Other negative errno value in case of failure.

int `uart_rx_enable_u16`(const struct *device* *dev, uint16_t *buf, size_t len, int32_t timeout)

Start receiving wide data through UART.

Function sets given buffer as first buffer for receiving and returns immediately. After that event handler, set using *uart_callback_set*, is called with *UART_RX_RDY* or *UART_RX_BUF_REQUEST* events.

Parameters

- `dev` – UART device instance.
- `buf` – Pointer to wide data receive buffer.
- `len` – Buffer length.
- `timeout` – Inactivity period after receiving at least a byte which triggers *UART_RX_RDY* event. Given in milliseconds. SYS_FOREVER_MS disables timeout. See *uart_event_type* for details.

Return values

- `0` – If successful.
- `-ENOTSUP` – If API is not enabled.
- `-EBUSY` – RX already in progress.
- `-errno` – Other negative errno value in case of failure.

static inline int `uart_rx_buf_rsp`(const struct *device* *dev, uint8_t *buf, size_t len)

Provide receive buffer in response to *UART_RX_BUF_REQUEST* event.

Provide pointer to RX buffer, which will be used when current buffer is filled.

> **Note**
>
> Providing buffer that is already in usage by driver leads to undefined behavior. Buffer can be reused when it has been released by driver.

Parameters

- `dev` – UART device instance.
- `buf` – Pointer to receive buffer.
- `len` – Buffer length.

Return values

- `0` – If successful.
- `-ENOTSUP` – If API is not enabled.
- `-EBUSY` – Next buffer already set.
- `-EACCES` – Receiver is already disabled (function called too late?).
- `-errno` – Other negative errno value in case of failure.

static inline int `uart_rx_buf_rsp_u16`(const struct *device* *dev, uint16_t *buf, size_t len)

Provide wide data receive buffer in response to *UART_RX_BUF_REQUEST* event.

Provide pointer to RX buffer, which will be used when current buffer is filled.

> **Note**
>
> Providing buffer that is already in usage by driver leads to undefined behavior. Buffer can be reused when it has been released by driver.

Parameters

- `dev` – UART device instance.
- `buf` – Pointer to wide data receive buffer.
- `len` – Buffer length.

Return values

- `0` – If successful.
- `-ENOTSUP` – If API is not enabled
- `-EBUSY` – Next buffer already set.
- `-EACCES` – Receiver is already disabled (function called too late?).
- `-errno` – Other negative errno value in case of failure.

int `uart_rx_disable`(const struct *device* *dev)

Disable RX.

UART_RX_BUF_RELEASED event will be generated for every buffer scheduled, after that *UART_RX_DISABLED* event will be generated. Additionally, if there is any pending received data, the *UART_RX_RDY* event for that data will be generated before the *UART_RX_BUF_RELEASED* events.

Parameters

- `dev` – UART device instance.

Return values

- `0` – If successful.
- `-ENOTSUP` – If API is not enabled.
- `-EFAULT` – There is no active reception.
- `-errno` – Other negative errno value in case of failure.

struct `uart_event_tx`

#include <*uart.h*> UART TX event data.

Public Members

const uint8_t *`buf`

Pointer to current buffer.

size_t `len`

Number of bytes sent.

struct `uart_event_rx`

#include <*uart.h*> UART RX event data.

The data represented by the event is stored in rx.buf[rx.offset] to rx.buf[rx.offset+rx.len]. That is, the length is relative to the offset.

Public Members

uint8_t *`buf`

Pointer to current buffer.

size_t `offset`

Currently received data offset in bytes.

size_t `len`

Number of new bytes received.

struct `uart_event_rx_buf`

#include <*uart.h*> UART RX buffer released event data.

Public Members

uint8_t *buf
Pointer to buffer that is no longer in use.

struct uart_event_rx_stop
#include <*uart.h*> UART RX stopped data.

Public Members

enum *uart_rx_stop_reason* reason
Reason why receiving stopped.

struct *uart_event_rx* data
Last received data.

struct uart_event
#include <*uart.h*> Structure containing information about current event.

Public Members

enum *uart_event_type* type
Type of event.

union uart_event_data
#include <*uart.h*> Event data.

Public Members

struct *uart_event_tx* tx
UART_TX_DONE and *UART_TX_ABORTED* events data.

struct *uart_event_rx* rx
UART_RX_RDY event data.

struct *uart_event_rx_buf* rx_buf
UART_RX_BUF_RELEASED event data.

struct *uart_event_rx_stop* rx_stop
UART_RX_STOPPED event data.

7.6.48 USB-C VBUS

Overview

USB-C VBUS is the line in a USB Type-C connection that delivers power from a Source to a Sink device.

USB-C VBUS API The USB-C VBUS device driver presents an API that's used to control and measure VBUS.

Configuration Options

Related configuration options:

- `CONFIG_USBC_VBUS_DRIVER`

API Reference

group `usbc_vbus_api`

USB-C VBUS API.

Since
3.3

Version
0.1.0

Functions

static inline bool `usbc_vbus_check_level`(const struct *device* *dev, enum *tc_vbus_level* level)

Checks if VBUS is at a particular level.

Parameters

- `dev` – Runtime device structure
- `level` – The level voltage to check against

Return values

- `true` – if VBUS is at the level voltage
- `false` – if VBUS is not at that level voltage

static inline int `usbc_vbus_measure`(const struct *device* *dev, int *meas)

Reads and returns VBUS measured in mV.

Parameters

- `dev` – Runtime device structure
- `meas` – pointer where the measured VBUS voltage is stored

Return values

- `0` – on success

- `-EIO` – on failure

static inline int **usbc_vbus_discharge**(const struct *device* *dev, bool enable)

Controls a pin that discharges VBUS.

Parameters

- `dev` – Runtime device structure
- `enable` – Discharge VBUS when true

Return values

- `0` – on success
- `-EIO` – on failure
- `-ENOENT` – if discharge pin isn't defined

static inline int **usbc_vbus_enable**(const struct *device* *dev, bool enable)

Controls a pin that enables VBUS measurements.

Parameters

- `dev` – Runtime device structure
- `enable` – enable VBUS measurements when true

Return values

- `0` – on success
- `-EIO` – on failure
- `-ENOENT` – if enable pin isn't defined

struct **usbc_vbus_driver_api**

#include <usbc_vbus.h>

7.6.49 USB Type-C Port Controller (TCPC)

Overview

TCPC (USB Type-C Port Controller) The TCPC is a device used to simplify the implementation of a USB-C system by providing the following three function:

- VBUS and VCONN control USB Type-C: The TCPC may provide a Source device, the mechanism to control VBUS sourcing, and a Sink device, the mechanism to control VBUS sinking. A similar mechanism is provided for the control of VCONN.
- CC control and sensing: The TCPC implements logic for controlling the CC pin pull-up and pull-down resistors. It also provides a way to sense and report what resistors are present on the CC pin.
- Power Delivery message reception and transmission USB Power Delivery: The TCPC sends and receives messages constructed in the TCPM and places them on the CC lines.

TCPC API The TCPC device driver functions as the liaison between the TCPC device and the application software; this is accomplished by the Zephyr's API provided by the device driver that's used to communicate with and control the TCPC device.

Configuration Options

Related configuration options:

- CONFIG_USBC_TCPC_DRIVER

API Reference

group usb_type_c

USB Type-C.

Defines

TC_V_SINK_DISCONNECT_MIN_MV

VBUS minimum for a sink disconnect detection.

See Table 4-3 VBUS Sink Characteristics

TC_V_SINK_DISCONNECT_MAX_MV

VBUS maximum for a sink disconnect detection.

See Table 4-3 VBUS Sink Characteristics

TC_T_VBUS_ON_MAX_MS

From entry to Attached.SRC until VBUS reaches the minimum vSafe5V threshold as measured at the source's receptacle See Table 4-29 VBUS and VCONN Timing Parameters.

TC_T_VBUS_OFF_MAX_MS

From the time the Sink is detached until the Source removes VBUS and reaches vSafe0V (See USB PD).

See Table 4-29 VBUS and VCONN Timing Parameters

TC_T_VCONN_ON_MAX_MS

From the time the Source supplied VBUS in the Attached.SRC state.

See Table 4-29 VBUS and VCONN Timing Parameters

TC_T_VCONN_ON_PA_MAX_MS

From the time a Sink with accessory support enters the PoweredAccessory state until the Sink sources minimum VCONN voltage (see Table 4-5) See Table 4-29 VBUS and VCONN Timing Parameters.

TC_T_VCONN_OFF_MAX_MS

From the time that a Sink is detached or as directed until the VCONN supply is disconnected.

See Table 4-29 VBUS and VCONN Timing Parameters

TC_T_SINK_ADJ_MAX_MS

Response time for a Sink to adjust its current consumption to be in the specified range due to a change in USB Type-C Current advertisement See Table 4-29 VBUS and VCONN Timing Parameters.

TC_T_DRP_MIN_MS

The minimum period a DRP shall complete a Source to Sink and back advertisement See Table 4-30 DRP Timing Parameters.

TC_T_DRP_MAX_MS

The maximum period a DRP shall complete a Source to Sink and back advertisement See Table 4-30 DRP Timing Parameters.

TC_T_DRP_TRANSITION_MIN_MS

The minimum time a DRP shall complete transitions between Source and Sink roles during role resolution See Table 4-30 DRP Timing Parameters.

TC_T_DRP_TRANSITION_MAX_MS

The maximum time a DRP shall complete transitions between Source and Sink roles during role resolution See Table 4-30 DRP Timing Parameters.

TC_T_DRP_TRY_MIN_MS

Minimum wait time associated with the Try.SRC state.

See Table 4-30 DRP Timing Parameters

TC_T_DRP_TRY_MAX_MS

Maximum wait time associated with the Try.SRC state.

See Table 4-30 DRP Timing Parameters

TC_T_DRP_TRY_WAIT_MIN_MS

Minimum wait time associated with the Try.SNK state.

See Table 4-30 DRP Timing Parameters

TC_T_DRP_TRY_WAIT_MAX_MS

Maximum wait time associated with the Try.SNK state.

See Table 4-30 DRP Timing Parameters

TC_T_TRY_TIMEOUT_MIN_MS

Minimum timeout for transition from Try.SRC to TryWait.SNK.

See Table 4-30 DRP Timing Parameters

TC_T_TRY_TIMEOUT_MAX_MS

Maximum timeout for transition from Try.SRC to TryWait.SNK.

See Table 4-30 DRP Timing Parameters

TC_T_VPD_DETACH_MIN_MS

Minimum Time for a DRP to detect that the connected Charge-Through VCONNPowered USB Device has been detached, after VBUS has been removed.

See Table 4-30 DRP Timing Parameters

`TC_T_VPD_DETACH_MAX_MS`

Maximum Time for a DRP to detect that the connected Charge-Through VCONNPowered USB Device has been detached, after VBUS has been removed.

See Table 4-30 DRP Timing Parameters

`TC_T_CC_DEBOUNCE_MIN_MS`

Minimum time a port shall wait before it can determine it is attached See Table 4-31 CC Timing.

`TC_T_CC_DEBOUNCE_MAX_MS`

Maximum time a port shall wait before it can determine it is attached See Table 4-31 CC Timing.

`TC_T_PD_DEBOUNCE_MIN_MS`

Minimum time a Sink port shall wait before it can determine it is detached due to the potential for USB PD signaling on CC as described in the state definitions.

See Table 4-31 CC Timing

`TC_T_PD_DEBOUNCE_MAX_MS`

Maximum time a Sink port shall wait before it can determine it is detached due to the potential for USB PD signaling on CC as described in the state definitions.

See Table 4-31 CC Timing

`TC_T_TRY_CC_DEBOUNCE_MIN_MS`

Minimum Time a port shall wait before it can determine it is re-attached during the try-wait process.

See Table 4-31 CC Timing

`TC_T_TRY_CC_DEBOUNCE_MAX_MS`

Maximum Time a port shall wait before it can determine it is re-attached during the try-wait process.

See Table 4-31 CC Timing

`TC_T_ERROR_RECOVERY_SELF_POWERED_MIN_MS`

Minimum time a self-powered port shall remain in the ErrorRecovery state.

See Table 4-31 CC Timing

`TC_T_ERROR_RECOVERY_SOURCE_MIN_MS`

Minimum time a source shall remain in the ErrorRecovery state if it was sourcing VCONN in the previous state.

See Table 4-31 CC Timing

`TC_T_RP_VALUE_CHANGE_MIN_MS`

Minimum time a Sink port shall wait before it can determine there has been a change in Rp where CC is not BMC Idle or the port is unable to detect BMC Idle.

See Table 4-31 CC Timing

TC_T_RP_VALUE_CHANGE_MAX_MS

Maximum time a Sink port shall wait before it can determine there has been a change in Rp where CC is not BMC Idle or the port is unable to detect BMC Idle.

See Table 4-31 CC Timing

TC_T_SRC_DISCONNECT_MIN_MS

Minimum time a Source shall detect the SRC.Open state.

The Source should detect the SRC.Open state as quickly as practical. See Table 4-31 CC Timing

TC_T_SRC_DISCONNECT_MAX_MS

Maximum time a Source shall detect the SRC.Open state.

The Source should detect the SRC.Open state as quickly as practical. See Table 4-31 CC Timing

TC_T_NO_TOGGLE_CONNECT_MIN_MS

Minimum time to detect connection when neither Port Partner is toggling.

See Table 4-31 CC Timing

TC_T_NO_TOGGLE_CONNECT_MAX_MS

Maximum time to detect connection when neither Port Partner is toggling.

See Table 4-31 CC Timing

TC_T_ONE_PORT_TOGGLE_CONNECT_MIN_MS

Minimum time to detect connection when one Port Partner is toggling 0ms ... dcSRC.DRP max * tDRP max + 2 * tNoToggleConnect).

See Table 4-31 CC Timing

TC_T_ONE_PORT_TOGGLE_CONNECT_MAX_MS

Maximum time to detect connection when one Port Partner is toggling 0ms ... dcSRC.DRP max * tDRP max + 2 * tNoToggleConnect).

See Table 4-31 CC Timing

TC_T_TWO_PORT_TOGGLE_CONNECT_MIN_MS

Minimum time to detect connection when both Port Partners are toggling (0ms ... 5 * tDRP max + 2 * tNoToggleConnect).

See Table 4-31 CC Timing

TC_T_TWO_PORT_TOGGLE_CONNECT_MAX_MS

Maximum time to detect connection when both Port Partners are toggling (0ms ... 5 * tDRP max + 2 * tNoToggleConnect).

See Table 4-31 CC Timing

TC_T_VPDCTDD_MIN_US

Minimum time for a Charge-Through VCONN-Powered USB Device to detect that the Charge-Through source has disconnected from CC after VBUS has been removed, transition to CTUnattached.VPD, and re-apply its Rp termination advertising 3.0 A on the host port CC.

See Table 4-31 CC Timing

TC_T_VPDCTDD_MAX_MS

Maximum time for a Charge-Through VCONN-Powered USB Device to detect that the Charge-Through source has disconnected from CC after VBUS has been removed, transition to CTUnattached.VPD, and re-apply its Rp termination advertising 3.0 A on the host port CC.

See Table 4-31 CC Timing

TC_T_VPDDISABLE_MIN_MS

Minimum time for a Charge-Through VCONN-Powered USB Device shall remain in CTDisabled.VPD state.

See Table 4-31 CC Timing

Enums

enum **tc_cc_voltage_state**

CC Voltage status.

Values:

enumerator TC_CC_VOLT_OPEN = 0

No port partner connection.

enumerator TC_CC_VOLT_RA = 1

Port partner is applying Ra.

enumerator TC_CC_VOLT_RD = 2

Port partner is applying Rd.

enumerator TC_CC_VOLT_RP_DEF = 5

Port partner is applying Rp (0.5A)

enumerator TC_CC_VOLT_RP_1A5 = 6

enumerator TC_CC_VOLT_RP_3A0 = 7

Port partner is applying Rp (3.0A)

enum **tc_vbus_level**

VBUS level voltages.

Values:

enumerator TC_VBUS_SAFE0V = 0

VBUS is less than vSafe0V max.

enumerator TC_VBUS_PRESENT = 1

VBUS is at least vSafe5V min.

enumerator TC_VBUS_REMOVED = 2
VBUS is less than vSinkDisconnect max.

enum tc_rp_value
Pull-Up resistor values.
Values:

enumerator TC_RP_USB = 0
Pull-Up resistor for a current of 900mA.

enumerator TC_RP_1A5 = 1
Pull-Up resistor for a current of 1.5A.

enumerator TC_RP_3A0 = 2
Pull-Up resistor for a current of 3.0A.

enumerator TC_RP_RESERVED = 3
No Pull-Up resistor is applied.

enum tc_cc_pull
CC pull resistors.
Values:

enumerator TC_CC_RA = 0
Ra Pull-Down resistor.

enumerator TC_CC_RP = 1
Rp Pull-Up resistor.

enumerator TC_CC_RD = 2
Rd Pull-Down resistor.

enumerator TC_CC_OPEN = 3
No CC resistor.

enumerator TC_RA_RD = 4
Ra and Rd Pull-Down resistor.

enum tc_cable_plug
Cable plug.
See 6.2.1.1.7 Cable Plug. Only applies to SOP' and SOP". Replaced by pd_power_role for SOP packets.
Values:

enumerator PD_PLUG_FROM_DFP_UFP = 0

enumerator PD_PLUG_FROM_CABLE_VPD = 1

enum **tc_power_role**

Power Delivery Power Role.

Values:

enumerator **TC_ROLE_SINK** = 0

Power role is a sink.

enumerator **TC_ROLE_SOURCE** = 1

Power role is a source.

enum **tc_data_role**

Power Delivery Data Role.

Values:

enumerator **TC_ROLE_UFP** = 0

Data role is an Upstream Facing Port.

enumerator **TC_ROLE_DFP** = 1

Data role is a Downstream Facing Port.

enumerator **TC_ROLE_DISCONNECTED** = 2

Port is disconnected.

enum **tc_cc_polarity**

Polarity of the CC lines.

Values:

enumerator **TC_POLARITY_CC1** = 0

Use CC1 IO for Power Delivery communication.

enumerator **TC_POLARITY_CC2** = 1

Use CC2 IO for Power Delivery communication.

enum **tc_cc_states**

Possible port partner connections based on CC line states.

Values:

enumerator **TC_CC_NONE** = 0

No port partner attached.

enumerator **TC_CC_UFP_NONE** = 1

From DFP perspective.

No UFP accessory connected

enumerator **TC_CC_UFP_AUDIO_ACC** = 2

UFP Audio accessory connected.

enumerator TC_CC_UFP_DEBUG_ACC = 3
UFP Debug accessory connected.

enumerator TC_CC_UFP_ATTACHED = 4
Plain UFP attached.

enumerator TC_CC_DFP_ATTACHED = 5
From UFP perspective.
Plain DFP attached

enumerator TC_CC_DFP_DEBUG_ACC = 6
DFP debug accessory connected.

group usb_type_c_port_controller_api
USB Type-C Port Controller API.

Since
3.1

Version
0.1.0

Typedefs

typedef int (*tcpc_vconn_control_cb_t)(const struct *device* *dev, enum *tc_cc_polarity* pol, bool enable)

typedef int (*tcpc_vconn_discharge_cb_t)(const struct *device* *dev, enum *tc_cc_polarity* pol, bool enable)

typedef void (*tcpc_alert_handler_cb_t)(const struct *device* *dev, void *data, enum *tcpc_alert* alert)

Enums

enum tcpc_alert
TCPC Alert bits.
Values:

enumerator TCPC_ALERT_CC_STATUS
CC status changed.

enumerator TCPC_ALERT_POWER_STATUS
Power status changed.

enumerator TCPC_ALERT_MSG_STATUS
Receive Buffer register changed.

enumerator TCPC_ALERT_HARD_RESET_RECEIVED
Received Hard Reset message.

enumerator TCPC_ALERT_TRANSMIT_MSG_FAILED
SOP* message transmission not successful.

enumerator TCPC_ALERT_TRANSMIT_MSG_DISCARDED
Reset or SOP* message transmission not sent due to an incoming receive message.

enumerator TCPC_ALERT_TRANSMIT_MSG_SUCCESS
Reset or SOP* message transmission successful.

enumerator TCPC_ALERT_VBUS_ALARM_HI
A high-voltage alarm has occurred.

enumerator TCPC_ALERT_VBUS_ALARM_LO
A low-voltage alarm has occurred.

enumerator TCPC_ALERT_FAULT_STATUS
A fault has occurred.
Read the FAULT_STATUS register

enumerator TCPC_ALERT_RX_BUFFER_OVERFLOW
TCPC RX buffer has overflowed.

enumerator TCPC_ALERT_VBUS_SNK_DISCONNECT
The TCPC in Attached.SNK state has detected a sink disconnect.

enumerator TCPC_ALERT_BEGINNING_MSG_STATUS
Receive buffer register changed.

enumerator TCPC_ALERT_EXTENDED_STATUS
Extended status changed.

enumerator TCPC_ALERT_EXTENDED
An extended interrupt event has occurred.
Read the alert_extended register

enumerator TCPC_ALERT_VENDOR_DEFINED
A vendor defined alert has been detected.

enum tcpc_status_reg
TCPC Status register.
Values:

enumerator TCPC_CC_STATUS
The CC Status register.

enumerator TCPC_POWER_STATUS
The Power Status register.

enumerator TCPC_FAULT_STATUS
The Fault Status register.

enumerator TCPC_EXTENDED_STATUS
The Extended Status register.

enumerator TCPC_EXTENDED_ALERT_STATUS
The Extended Alert Status register.

enumerator TCPC_VENDOR_DEFINED_STATUS
The Vendor Defined Status register.

Functions

static inline int **tcpc_is_cc_rp**(enum *tc_cc_voltage_state* cc)
Returns whether the sink has detected a Rp resistor on the other side.

static inline int **tcpc_is_cc_open**(enum *tc_cc_voltage_state* cc1, enum *tc_cc_voltage_state* cc2)
Returns true if both CC lines are completely open.

static inline int **tcpc_is_cc_snk_dbg_acc**(enum *tc_cc_voltage_state* cc1, enum *tc_cc_voltage_state* cc2)
Returns true if we detect the port partner is a snk debug accessory.

static inline int **tcpc_is_cc_src_dbg_acc**(enum *tc_cc_voltage_state* cc1, enum *tc_cc_voltage_state* cc2)
Returns true if we detect the port partner is a src debug accessory.

static inline int **tcpc_is_cc_audio_acc**(enum *tc_cc_voltage_state* cc1, enum *tc_cc_voltage_state* cc2)
Returns true if the port partner is an audio accessory.

static inline int **tcpc_is_cc_at_least_one_rd**(enum *tc_cc_voltage_state* cc1, enum *tc_cc_voltage_state* cc2)
Returns true if the port partner is presenting at least one Rd.

static inline int **tcpc_is_cc_only_one_rd**(enum *tc_cc_voltage_state* cc1, enum *tc_cc_voltage_state* cc2)
Returns true if the port partner is presenting Rd on only one CC line.

static inline int **tcpc_init**(const struct *device* *dev)
Initializes the TCPC.

Parameters

- `dev` – Runtime device structure

Return values

- `0` – on success
- `-EIO` – on failure
- `-EAGAIN` – if initialization should be postponed

static inline int **tcpc_get_cc**(const struct *device* *dev, enum *tc_cc_voltage_state* *cc1, enum *tc_cc_voltage_state* *cc2)

Reads the status of the CC lines.

Parameters

- `dev` – Runtime device structure
- `cc1` – A pointer where the CC1 status is written
- `cc2` – A pointer where the CC2 status is written

Return values

- `0` – on success
- `-EIO` – on failure
- `-ENOSYS` – if not implemented

static inline int **tcpc_select_rp_value**(const struct *device* *dev, enum *tc_rp_value* rp)

Sets the value of CC pull up resistor used when operating as a Source.

Parameters

- `dev` – Runtime device structure
- `rp` – Value of the Pull-Up Resistor.

Return values

- `0` – on success
- `-ENOSYS` –
- `-EIO` – on failure

static inline int **tcpc_get_rp_value**(const struct *device* *dev, enum *tc_rp_value* *rp)

Gets the value of the CC pull up resistor used when operating as a Source.

Parameters

- `dev` – Runtime device structure
- `rp` – pointer where the value of the Pull-Up Resistor is stored

Return values

- `0` – on success
- `-ENOSYS` –
- `-EIO` – on failure

static inline int **tcpc_set_cc**(const struct *device* *dev, enum *tc_cc_pull* pull)

Sets the CC pull resistor and sets the role as either Source or Sink.

Parameters

- `dev` – Runtime device structure
- `pull` – The pull resistor to set

Return values

- `0` – on success
- `-EIO` – on failure

static inline void **tcpc_set_vconn_cb**(const struct *device* *dev, *tcpc_vconn_control_cb_t* vconn_cb)

Sets a callback that can enable or disable VCONN if the TCPC is unable to or the system is configured in a way that does not use the VCONN control capabilities of the TCPC.

The callback is called in the tcpc_set_vconn function if vconn_cb isn't NULL

Parameters

- **dev** – Runtime device structure
- **vconn_cb** – pointer to the callback function that controls vconn

static inline void **tcpc_set_vconn_discharge_cb**(const struct *device* *dev, *tcpc_vconn_discharge_cb_t* cb)

Sets a callback that can enable or discharge VCONN if the TCPC is unable to or the system is configured in a way that does not use the VCONN control capabilities of the TCPC.

The callback is called in the tcpc_vconn_discharge function if cb isn't NULL

Parameters

- **dev** – Runtime device structure
- **cb** – pointer to the callback function that discharges vconn

static inline int **tcpc_vconn_discharge**(const struct *device* *dev, bool enable)

Discharges VCONN.

This function uses the TCPC to discharge VCONN if possible or calls the callback function set by tcpc_set_vconn_cb

Parameters

- **dev** – Runtime device structure
- **enable** – VCONN discharge is enabled when true, it's disabled

Return values

- **0** – on success
- **-EIO** – on failure
- **-ENOSYS** – if not implemented

static inline int **tcpc_set_vconn**(const struct *device* *dev, bool enable)

Enables or disables VCONN.

This function uses the TCPC to measure VCONN if possible or calls the callback function set by tcpc_set_vconn_cb

Parameters

- **dev** – Runtime device structure
- **enable** – VCONN is enabled when true, it's disabled

Return values

- **0** – on success
- **-EIO** – on failure
- **-ENOSYS** – if not implemented

static inline int **tcpc_set_roles**(const struct *device* *dev, enum *tc_power_role* power_role, enum *tc_data_role* data_role)

Sets the Power and Data Role of the PD message header.

This function only needs to be called once per data / power role change

Parameters

- **dev** – Runtime device structure
- **power_role** – current power role
- **data_role** – current data role

Return values

- **0** – on success
- **-EIO** – on failure
- **-ENOSYS** – if not implemented

static inline int **tcpc_get_rx_pending_msg**(const struct *device* *dev, struct *pd_msg* *buf)

Retrieves the Power Delivery message from the TCPC.

If buf is NULL, then only the status is returned, where 0 means there is a message pending and -ENODATA means there is no pending message.

Parameters

- **dev** – Runtime device structure
- **buf** – pointer where the pd_buf pointer is written, NULL if only checking the status

Return values

- **Greater** – or equal to 0 is the number of bytes received if buf parameter is provided
- **0** – if there is a message pending and buf parameter is NULL
- **-EIO** – on failure
- **-ENODATA** – if no message is pending

static inline int **tcpc_set_rx_enable**(const struct *device* *dev, bool enable)

Enables the reception of SOP* message types.

Parameters

- **dev** – Runtime device structure
- **enable** – Enable Power Delivery when true, else it's disabled

Return values

- **0** – on success
- **-EIO** – on failure
- **-ENOSYS** – if not implemented

static inline int **tcpc_set_cc_polarity**(const struct *device* *dev, enum *tc_cc_polarity* polarity)

Sets the polarity of the CC lines.

Parameters

- **dev** – Runtime device structure
- **polarity** – Polarity of the cc line

Return values

- `0` – on success
- `-EIO` – on failure

static inline int **tcpc_transmit_data**(const struct *device* *dev, struct *pd_msg* *msg)

Transmits a Power Delivery message.

Parameters

- `dev` – Runtime device structure
- `msg` – Power Delivery message to transmit

Return values

- `0` – on success
- `-EIO` – on failure
- `-ENOSYS` – if not implemented

static inline int **tcpc_dump_std_reg**(const struct *device* *dev)

Dump a set of TCPC registers.

Parameters

- `dev` – Runtime device structure

Return values

- `0` – on success
- `-EIO` – on failure
- `-ENOSYS` – if not implemented

static inline int **tcpc_set_alert_handler_cb**(const struct *device* *dev, *tcpc_alert_handler_cb_t* handler, void *data)

Sets the alert function that's called when an interrupt is triggered due to an alert bit.

Calling this function enables the particular alert bit

Parameters

- `dev` – Runtime device structure
- `handler` – The callback function called when the bit is set
- `data` – user data passed to the callback

Return values

- `0` – on success
- `-EINVAL` – on failure

static inline int **tcpc_get_status_register**(const struct *device* *dev, enum *tcpc_status_reg* reg, int32_t *status)

Gets a status register.

Parameters

- `dev` – Runtime device structure
- `reg` – The status register to read
- `status` – Pointer where the status is stored

Return values

- `0` – on success

- `-EIO` – on failure
- `-ENOSYS` – if not implemented

static inline int **tcpc_clear_status_register**(const struct *device* *dev, enum *tcpc_status_reg* reg, uint32_t mask)

Clears a TCPC status register.

Parameters

- `dev` – Runtime device structure
- `reg` – The status register to read
- `mask` – A bit mask of the status register to clear. A status bit is cleared when it's set to 1.

Return values

- `0` – on success
- `-EIO` – on failure
- `-ENOSYS` – if not implemented

static inline int **tcpc_mask_status_register**(const struct *device* *dev, enum *tcpc_status_reg* reg, uint32_t mask)

Sets the mask of a TCPC status register.

Parameters

- `dev` – Runtime device structure
- `reg` – The status register to read
- `mask` – A bit mask of the status register to mask. The status bit is masked if it's 0, else it's unmasked.

Return values

- `0` – on success
- `-EIO` – on failure
- `-ENOSYS` – if not implemented

static inline int **tcpc_set_debug_accessory**(const struct *device* *dev, bool enable)

Manual control of TCPC DebugAccessory control.

Parameters

- `dev` – Runtime device structure
- `enable` – Enable Debug Accessory when true, else it's disabled

Return values

- `0` – on success
- `-EIO` – on failure
- `-ENOSYS` – if not implemented

static inline int **tcpc_set_debug_detach**(const struct *device* *dev)

Detach from a debug connection.

Parameters

- `dev` – Runtime device structure

Return values

- `0` – on success

- `-EIO` – on failure
- `-ENOSYS` – if not implemented

static inline int **tcpc_set_drp_toggle**(const struct *device* *dev, bool enable)

Enable TCPC auto dual role toggle.

Parameters

- `dev` – Runtime device structure
- `enable` – Auto dual role toggle is active when true, else it's disabled

Return values

- `0` – on success
- `-EIO` – on failure
- `-ENOSYS` – if not implemented

static inline int **tcpc_get_snk_ctrl**(const struct *device* *dev)

Queries the current sinking state of the TCPC.

Parameters

- `dev` – Runtime device structure

Return values

- `true` – if sinking power
- `false` – if not sinking power
- `-ENOSYS` – if not implemented

static inline int **tcpc_set_snk_ctrl**(const struct *device* *dev, bool enable)

Set the VBUS sinking state of the TCPC.

Parameters

- `dev` – Runtime device structure
- `enable` – True if sinking should be enabled, false if disabled

Return values

- `0` – on success
- `-ENOSYS` – if not implemented

static inline int **tcpc_get_src_ctrl**(const struct *device* *dev)

Queries the current sourcing state of the TCPC.

Parameters

- `dev` – Runtime device structure

Return values

- `true` – if sourcing power
- `false` – if not sourcing power
- `-ENOSYS` – if not implemented

static inline int **tcpc_set_src_ctrl**(const struct *device* *dev, bool enable)

Set the VBUS sourcing state of the TCPC.

Parameters

- `dev` – Runtime device structure
- `enable` – True if sourcing should be enabled, false if disabled

Return values

- `0` – on success
- `-ENOSYS` – if not implemented

static inline int **tcpc_set_bist_test_mode**(const struct *device* *dev, bool enable)

Controls the BIST Mode of the TCPC.

It disables RX alerts while the mode is active.

Parameters

- `dev` – Runtime device structure
- `enable` – The TCPC enters BIST TEST Mode when true

Return values

- `0` – on success
- `-EIO` – on failure
- `-ENOSYS` – if not implemented

static inline int **tcpc_get_chip_info**(const struct *device* *dev, struct *tcpc_chip_info* *chip_info)

Gets the TCPC firmware version.

Parameters

- `dev` – Runtime device structure
- `chip_info` – Pointer to TCPC chip info where the version is stored

Return values

- `0` – on success
- `-EIO` – on failure
- `-ENOSYS` – if not implemented

static inline int **tcpc_set_low_power_mode**(const struct *device* *dev, bool enable)

Instructs the TCPC to enter or exit low power mode.

Parameters

- `dev` – Runtime device structure
- `enable` – The TCPC enters low power mode when true, else it exits it

Return values

- `0` – on success
- `-EIO` – on failure
- `-ENOSYS` – if not implemented

static inline int **tcpc_sop_prime_enable**(const struct *device* *dev, bool enable)

Enables the reception of SOP Prime messages.

Parameters

- `dev` – Runtime device structure
- `enable` – Can receive SOP Prime messages when true, else it can not

Return values

- `0` – on success
- `-EIO` – on failure

- `-ENOSYS` – if not implemented

struct `tcpc_chip_info`

#include <usbc_tcpc.h> TCPC Chip Information.

Public Members

uint16_t `vendor_id`

Vendor Id.

uint16_t `product_id`

Product Id.

uint16_t `device_id`

Device Id.

uint64_t `fw_version_number`

Firmware version number.

uint8_t `min_req_fw_version_string`[8]

Minimum Required firmware version string.

uint64_t `min_req_fw_version_number`

Minimum Required firmware version number.

struct `tcpc_driver_api`

#include <usbc_tcpc.h>

group `usb_power_delivery`

USB Power Delivery.

USB PD 3.1 Rev 1.6, Table 6-70 Counter Parameters

`PD_N_CAPS_COUNT`

The CapsCounter is used to count the number of Source_Capabilities Messages which have been sent by a Source at power up or after a Hard Reset.

Parameter Name: nCapsCounter

`PD_N_HARD_RESET_COUNT`

The HardResetCounter is used to retry the Hard Reset whenever there is no response from the remote device (see Section 6.6.6) Parameter Name: nHardResetCounter.

USB PD 3.1 Rev 1.6, Table 6-68 Time Values

PD_T_NO_RESPONSE_MIN_MS

The NoResponseTimer is used by the Policy Engine in a Source to determine that its Port Partner is not responding after a Hard Reset.

Parameter Name: tNoResponseTimer

PD_T_NO_RESPONSE_MAX_MS

The NoResponseTimer is used by the Policy Engine in a Source to determine that its Port Partner is not responding after a Hard Reset.

Parameter Name: tNoResponseTimer

PD_T_PS_HARD_RESET_MIN_MS

Min time the Source waits to ensure that the Sink has had sufficient time to process Hard Reset Signaling before turning off its power supply to VBUS Parameter Name: tPSHardReset.

PD_T_PS_HARD_RESET_MAX_MS

Max time the Source waits to ensure that the Sink has had sufficient time to process Hard Reset Signaling before turning off its power supply to VBUS Parameter Name: tPSHardReset.

PD_T_SINK_TX_MIN_MS

Minimum time a Source waits after changing Rp from SinkTxOk to SinkTxNG before initiating an AMS by sending a Message.

Parameter Name: tSinkTx

PD_T_SINK_TX_MAX_MS

Maximum time a Source waits after changing Rp from SinkTxOk to SinkTxNG before initiating an AMS by sending a Message.

Parameter Name: tSinkTx

PD_T_TYPEC_SEND_SOURCE_CAP_MIN_MS

Minimum time a source shall wait before sending a Source_Capabilities message while the following is true: 1) The Port is Attached.

2) The Source is not in an active connection with a PD Sink Port. Parameter Name: tTypeCSendSourceCap

PD_T_TYPEC_SEND_SOURCE_CAP_MAX_MS

Maximum time a source shall wait before sending a Source_Capabilities message while the following is true: 1) The Port is Attached.

2) The Source is not in an active connection with a PD Sink Port. Parameter Name: tTypeCSendSourceCap

Defines

PD_MAX_EXTENDED_MSG_LEGACY_LEN

Maximum length of a non-Extended Message in bytes.

See Table 6-75 Value Parameters Parameter Name: MaxExtendedMsgLegacyLen

`PD_MAX_EXTENDED_MSG_LEN`

Maximum length of an Extended Message in bytes.

See Table 6-75 Value Parameters Parameter Name: MaxExtendedMsgLen

`PD_MAX_EXTENDED_MSG_CHUNK_LEN`

Maximum length of a Chunked Message in bytes.

When one of both Port Partners do not support Extended Messages of Data Size greater than PD_MAX_EXTENDED_MSG_LEGACY_LEN then the Protocol Layer supports a Chunking mechanism to break larger Messages into smaller Chunks of size PD_MAX_EXTENDED_MSG_CHUNK_LEN. See Table 6-75 Value Parameters Parameter Name: MaxExtendedMsgChunkLen

`PD_T_TYPEC_SINK_WAIT_CAP_MIN_MS`

Minimum time a sink shall wait for a Source_Capabilities message before sending a Hard Reset See Table 6-61 Time Values Parameter Name: tTypeCSinkWaitCap.

`PD_T_TYPEC_SINK_WAIT_CAP_MAX_MS`

Minimum time a sink shall wait for a Source_Capabilities message before sending a Hard Reset See Table 6-61 Time Values Parameter Name: tTypeCSinkWaitCap.

`PD_V_SAFE_0V_MAX_MV`

VBUS maximum safe operating voltage at “zero volts”.

See Table 7-24 Common Source/Sink Electrical Parameters Parameter Name: vSafe0V

`PD_V_SAFE_5V_MIN_MV`

VBUS minimum safe operating voltage at 5V.

See Table 7-24 Common Source/Sink Electrical Parameters Parameter Name: vSafe5V

`PD_T_SAFE_0V_MAX_MS`

Time to reach PD_V_SAFE_0V_MV max in milliseconds.

See Table 7-24 Common Source/Sink Electrical Parameters Parameter Name: tSafe0V

`PD_T_SAFE_5V_MAX_MS`

Time to reach PD_V_SAFE_5V_MV max in milliseconds.

See Table 7-24 Common Source/Sink Electrical Parameters Parameter Name: tSafe5V

`PD_T_TX_TIMEOUT_MS`

Time to wait for TCPC to complete transmit.

`PD_T_HARD_RESET_COMPLETE_MIN_MS`

Minimum time a Hard Reset must complete.

See Table 6-68 Time Values

`PD_T_HARD_RESET_COMPLETE_MAX_MS`

Maximum time a Hard Reset must complete.

See Table 6-68 Time Values

PD_T_SENDER_RESPONSE_MIN_MS

Minimum time a response must be sent from a Port Partner See Table 6-68 Time Values.

PD_T_SENDER_RESPONSE_NOM_MS

Nomiminal time a response must be sent from a Port Partner See Table 6-68 Time Values.

PD_T_SENDER_RESPONSE_MAX_MS

Maximum time a response must be sent from a Port Partner See Table 6-68 Time Values.

PD_T_SPR_PS_TRANSITION_MIN_MS

Minimum SPR Mode time for a power supply to transition to a new level See Table 6-68 Time Values.

PD_T_SPR_PS_TRANSITION_NOM_MS

Nominal SPR Mode time for a power supply to transition to a new level See Table 6-68 Time Values.

PD_T_SPR_PS_TRANSITION_MAX_MS

Maximum SPR Mode time for a power supply to transition to a new level See Table 6-68 Time Values.

PD_T_EPR_PS_TRANSITION_MIN_MS

Minimum EPR Mode time for a power supply to transition to a new level See Table 6-68 Time Values.

PD_T_EPR_PS_TRANSITION_NOM_MS

Nominal EPR Mode time for a power supply to transition to a new level See Table 6-68 Time Values.

PD_T_EPR_PS_TRANSITION_MAX_MS

Maximum EPR Mode time for a power supply to transition to a new level See Table 6-68 Time Values.

PD_T_SINK_REQUEST_MIN_MS

Minimum time to wait before sending another request after receiving a Wait message See Table 6-68 Time Values.

PD_T_CHUNKING_NOT_SUPPORTED_MIN_MS

Minimum time to wait before sending a Not_Supported message after receiving a Chunked message See Table 6-68 Time Values.

PD_T_CHUNKING_NOT_SUPPORTED_NOM_MS

Nominal time to wait before sending a Not_Supported message after receiving a Chunked message See Table 6-68 Time Values.

PD_T_CHUNKING_NOT_SUPPORTED_MAX_MS

Maximum time to wait before sending a Not_Supported message after receiving a Chunked message See Table 6-68 Time Values.

PD_CONVERT_BYTES_TO_PD_HEADER_COUNT(c)

Convert bytes to PD Header data object count, where a data object is 4-bytes.

Parameters

- **c** – number of bytes to convert

PD_CONVERT_PD_HEADER_COUNT_TO_BYTES(c)

Convert PD Header data object count to bytes.

Parameters

- **c** – number of PD Header data objects

SINK_TX_OK

Collision avoidance Rp values in REV 3.0 Sink Transmit “OK”.

SINK_TX_NG

Collision avoidance Rp values in REV 3.0 Sink Transmit “NO GO”.

PD_GET_EXT_HEADER(c)

Used to get extended header from the first 32-bit word of the message.

Parameters

- **c** – first 32-bit word of the message

PDO_MAX_DATA_OBJECTS

PDO - Power Data Object RDO - Request Data Object.

Maximum number of 32-bit data objects sent in a single request

PD_CONVERT_MA_TO_FIXED_PDO_CURRENT(c)

Convert milliamps to Fixed PDO Current in 10mA units.

Parameters

- **c** – Current in milliamps

PD_CONVERT_MV_TO_FIXED_PDO_VOLTAGE(v)

Convert millivolts to Fixed PDO Voltage in 50mV units.

Parameters

- **v** – Voltage in millivolts

PD_CONVERT_FIXED_PDO_CURRENT_TO_MA(c)

Convert a Fixed PDO Current from 10mA units to milliamps.

Parameters

- **c** – Fixed PDO current in 10mA units.

PD_CONVERT_FIXED_PDO_VOLTAGE_TO_MV(v)

Convert a Fixed PDO Voltage from 50mV units to millivolts.

Used for converting *pd_fixed_supply_pdo_source.voltage* and *pd_fixed_supply_pdo_sink.voltage*

Parameters

- **v** – Fixed PDO voltage in 50mV units.

PD_CONVERT_MA_TO_VARIABLE_PDO_CURRENT(c)

Convert milliamps to Variable PDO Current in 10ma units.

Parameters

- **c** – Current in milliamps

PD_CONVERT_MV_TO_VARIABLE_PDO_VOLTAGE(v)

Convert millivolts to Variable PDO Voltage in 50mV units.

Parameters

- **v** – Voltage in millivolts

PD_CONVERT_VARIABLE_PDO_CURRENT_TO_MA(c)

Convert a Variable PDO Current from 10mA units to milliamps.

Parameters

- **c** – Variable PDO current in 10mA units.

PD_CONVERT_VARIABLE_PDO_VOLTAGE_TO_MV(v)

Convert a Variable PDO Voltage from 50mV units to millivolts.

Parameters

- **v** – Variable PDO voltage in 50mV units.

PD_CONVERT_MW_TO_BATTERY_PDO_POWER(c)

Convert milliwatts to Battery PDO Power in 250mW units.

Parameters

- **c** – Power in milliwatts

PD_CONVERT_MV_TO_BATTERY_PDO_VOLTAGE(v)

Convert milliwatts to Battery PDO Voltage in 50mV units.

Parameters

- **v** – Voltage in millivolts

PD_CONVERT_BATTERY_PDO_POWER_TO_MW(c)

Convert a Battery PDO Power from 250mW units to milliwatts.

Parameters

- **c** – Power in 250mW units.

PD_CONVERT_BATTERY_PDO_VOLTAGE_TO_MV(v)

Convert a Battery PDO Voltage from 50mV units to millivolts.

Parameters

- **v** – Voltage in 50mV units.

PD_CONVERT_MA_TO_AUGMENTED_PDO_CURRENT(c)

Convert milliamps to Augmented PDO Current in 50mA units.

Parameters

- **c** – Current in milliamps

PD_CONVERT_MV_TO_AUGMENTED_PDO_VOLTAGE(v)

Convert millivolts to Augmented PDO Voltage in 100mV units.

Parameters

- **v** – Voltage in millivolts

PD_CONVERT_AUGMENTED_PDO_CURRENT_TO_MA(c)

Convert an Augmented PDO Current from 50mA units to milliamps.

Parameters

- c – Augmented PDO current in 50mA units.

PD_CONVERT_AUGMENTED_PDO_VOLTAGE_TO_MV(v)

Convert an Augmented PDO Voltage from 100mV units to millivolts.

Parameters

- v – Augmented PDO voltage in 100mV units.

NUM_SOP_STAR_TYPES

Number of valid Transmit Types.

Enums

enum pdo_type

Power Data Object Type Table 6-7 Power Data Object.

Values:

enumerator PDO_FIXED = 0

Fixed supply (Vmin = Vmax)

enumerator PDO_BATTERY = 1

Battery.

enumerator PDO_VARIABLE = 2

Variable Supply (non-Battery)

enumerator PDO_AUGMENTED = 3

Augmented Power Data Object (APDO)

enum pd_frs_type

Fast Role Swap Required for USB Type-C current.

Values:

enumerator FRS_NOT_SUPPORTED

Fast Swap not supported.

enumerator FRS_DEFAULT_USB_POWER

Default USB Power.

enumerator FRS_1P5A_5V

1.5A @ 5V

enumerator FRS_3P0A_5V

3.0A @ 5V

enum pd_rev_type

Protocol revision.

Values:

enumerator PD_REV10 = 0

PD revision 1.0.

enumerator PD_REV20 = 1

PD revision 2.0.

enumerator PD_REV30 = 2

PD revision 3.0.

enum pd_packet_type

Power Delivery packet type See USB Type-C Port Controller Interface Specification, Revision 2.0, Version 1.2, Table 4-38 TRANSMIT Register Definition.

Values:

enumerator PD_PACKET_SOP = 0

Port Partner message.

enumerator PD_PACKET_SOP_PRIME = 1

Cable Plug message.

enumerator PD_PACKET_PRIME_PRIME = 2

Cable Plug message far end.

enumerator PD_PACKET_DEBUG_PRIME = 3

Currently undefined in the PD specification.

enumerator PD_PACKET_DEBUG_PRIME_PRIME = 4

Currently undefined in the PD specification.

enumerator PD_PACKET_TX_HARD_RESET = 5

Hard Reset message to the Port Partner.

enumerator PD_PACKET_CABLE_RESET = 6

Cable Reset message to the Cable.

enumerator PD_PACKET_TX_BIST_MODE_2 = 7

BIST_MODE_2 message to the Port Partner.

enumerator PD_PACKET_MSG_INVALID = 0xf

USED ONLY FOR RECEPTION OF UNKNOWN MSG TYPES.

enum pd_ctrl_msg_type

Control Message type See Table 6-5 Control Message Types.

Values:

enumerator PD_CTRL_GOOD_CRC = 1

0 Reserved

GoodCRC Message

enumerator PD_CTRL_GOTO_MIN = 2

GotoMin Message.

enumerator PD_CTRL_ACCEPT = 3

Accept Message.

enumerator PD_CTRL_REJECT = 4

Reject Message.

enumerator PD_CTRL_PING = 5

Ping Message.

enumerator PD_CTRL_PS_RDY = 6

PS_RDY Message.

enumerator PD_CTRL_GET_SOURCE_CAP = 7

Get_Source_Cap Message.

enumerator PD_CTRL_GET_SINK_CAP = 8

Get_Sink_Cap Message.

enumerator PD_CTRL_DR_SWAP = 9

DR_Swap Message.

enumerator PD_CTRL_PR_SWAP = 10

PR_Swap Message.

enumerator PD_CTRL_VCONN_SWAP = 11

VCONN_Swap Message.

enumerator PD_CTRL_WAIT = 12

Wait Message.

enumerator PD_CTRL_SOFT_RESET = 13

Soft Reset Message.

enumerator PD_CTRL_DATA_RESET = 14

Used for REV 3.0.

Data_Reset Message

enumerator PD_CTRL_DATA_RESET_COMPLETE = 15

Data_Reset_Complete Message.

enumerator PD_CTRL_NOT_SUPPORTED = 16
Not_Supported Message.

enumerator PD_CTRL_GET_SOURCE_CAP_EXT = 17
Get_Source_Cap_Extended Message.

enumerator PD_CTRL_GET_STATUS = 18
Get_Status Message.

enumerator PD_CTRL_FR_SWAP = 19
FR_Swap Message.

enumerator PD_CTRL_GET_PPS_STATUS = 20
Get_PPS_Status Message.

enumerator PD_CTRL_GET_COUNTRY_CODES = 21
Get_Country_Codes Message.

enumerator PD_CTRL_GET_SINK_CAP_EXT = 22
Get_Sink_Cap_Extended Message.

enum pd_data_msg_type
Data message type See Table 6-6 Data Message Types.
Values:

enumerator PD_DATA_SOURCE_CAP = 1
0 Reserved
Source_Capabilities Message

enumerator PD_DATA_REQUEST = 2
Request Message.

enumerator PD_DATA_BIST = 3
BIST Message.

enumerator PD_DATA_SINK_CAP = 4
Sink Capabilities Message.

enumerator PD_DATA_BATTERY_STATUS = 5
5-14 Reserved for REV 2.0

enumerator PD_DATA_ALERT = 6
Alert Message.

enumerator PD_DATA_GET_COUNTRY_INFO = 7
Get Country Info Message.

enumerator PD_DATA_ENTER_USB = 8

8-14 Reserved for REV 3.0

Enter USB message

enumerator PD_DATA_VENDOR_DEF = 15

Vendor Defined Message.

enum pd_ext_msg_type

Extended message type for REV 3.0 See Table 6-48 Extended Message Types.

Values:

enumerator PD_EXT_SOURCE_CAP = 1

0 Reserved

Source_Capabilities_Extended Message

enumerator PD_EXT_STATUS = 2

Status Message.

enumerator PD_EXT_GET_BATTERY_CAP = 3

Get_Battery_Cap Message.

enumerator PD_EXT_GET_BATTERY_STATUS = 4

Get_Battery_Status Message.

enumerator PD_EXT_BATTERY_CAP = 5

Battery_Capabilities Message.

enumerator PD_EXT_GET_MANUFACTURER_INFO = 6

Get_Manufacturer_Info Message.

enumerator PD_EXT_MANUFACTURER_INFO = 7

Manufacturer_Info Message.

enumerator PD_EXT_SECURITY_REQUEST = 8

Security_Request Message.

enumerator PD_EXT_SECURITY_RESPONSE = 9

Security_Response Message.

enumerator PD_EXT_FIRMWARE_UPDATE_REQUEST = 10

Firmware_Update_Request Message.

enumerator PD_EXT_FIRMWARE_UPDATE_RESPONSE = 11

Firmware_Update_Response Message.

enumerator PD_EXT_PPS_STATUS = 12

PPS_Status Message.

enumerator PD_EXT_COUNTRY_INFO = 13
Country_Codes Message.

enumerator PD_EXT_COUNTRY_CODES = 14
Country_Info Message.

enum usbpd_cc_pin
Active PD CC pin.
Values:

enumerator USBPD_CC_PIN_1 = 0
PD is active on CC1.

enumerator USBPD_CC_PIN_2 = 1
PD is active on CC2.

union pd_header
#include <usbc_pd.h> Build a PD message header See Table 6-1 Message Header.

Public Members

uint16_t message_type
Type of message.

uint16_t port_data_role
Port Data role.

uint16_t specification_revision
Specification Revision.

uint16_t port_power_role
Port Power Role.

uint16_t message_id
Message ID.

uint16_t number_of_data_objects
Number of Data Objects.

uint16_t extended
Extended Message.

struct pd_header

uint16_t raw_value

union **pd_ext_header**

#include <usbc_pd.h> Build an extended message header See Table 6-3 Extended Message Header.

Public Members

uint16_t **data_size**

Number of total bytes in data block.

uint16_t **reserved0**

Reserved.

uint16_t **request_chunk**

1 for a chunked message, else 0

uint16_t **chunk_number**

Chunk number when chkd = 1, else 0.

uint16_t **chunked**

1 for chunked messages

struct pd_ext_header

uint16_t **raw_value**

Raw PD Ext Header value.

union **pd_fixed_supply_pdo_source**

#include <usbc_pd.h> Create a Fixed Supply PDO Source value See Table 6-9 Fixed Supply PDO - Source.

Public Members

uint32_t **max_current**

Maximum Current in 10mA units.

uint32_t **voltage**

Voltage in 50mV units.

uint32_t **peak_current**

Peak Current.

uint32_t **reserved0**

Reserved – Shall be set to zero.

uint32_t **unchunked_ext_msg_supported**

Unchunked Extended Messages Supported.

uint32_t **dual_role_data**
Dual-Role Data.

uint32_t **usb_comms_capable**
USB Communications Capable.

uint32_t **unconstrained_power**
Unconstrained Power.

uint32_t **usb_suspend_supported**
USB Suspend Supported.

uint32_t **dual_role_power**
Dual-Role Power.

enum *pdo_type* **type**
Fixed supply.
SET TO PDO_FIXED

struct pd_fixed_supply_pdo_source

uint32_t **raw_value**
Raw PDO value.

union **pd_fixed_supply_pdo_sink**
#include <usbc_pd.h> Create a Fixed Supply PDO Sink value See Table 6-14 Fixed Supply PDO - Sink.

Public Members

uint32_t **operational_current**
Operational Current in 10mA units.

uint32_t **voltage**
Voltage in 50mV units.

uint32_t **reserved0**
Reserved – Shall be set to zero.

enum *pd_frs_type* **frs_required**
Fast Role Swap required USB Type-C Current.

uint32_t **dual_role_data**
Dual-Role Data.

uint32_t **usb_comms_capable**
USB Communications Capable.

uint32_t **unconstrained_power**

Unconstrained Power.

uint32_t **higher_capability**

Higher Capability.

uint32_t **dual_role_power**

Dual-Role Power.

enum *pdo_type* **type**

Fixed supply.

SET TO PDO_FIXED

struct pd_fixed_supply_pdo_sink

uint32_t **raw_value**

Raw PDO value.

union **pd_variable_supply_pdo_source**

#include <usbc_pd.h> Create a Variable Supply PDO Source value See Table 6-11 Variable Supply (non-Battery) PDO - Source.

Public Members

uint32_t **max_current**

Maximum Current in 10mA units.

uint32_t **min_voltage**

Minimum Voltage in 50mV units.

uint32_t **max_voltage**

Maximum Voltage in 50mV units.

enum *pdo_type* **type**

Variable supply.

SET TO PDO_VARIABLE

struct pd_variable_supply_pdo_source

uint32_t **raw_value**

Raw PDO value.

union **pd_variable_supply_pdo_sink**

#include <usbc_pd.h> Create a Variable Supply PDO Sink value See Table 6-15 Variable Supply (non-Battery) PDO - Sink.

Public Members

uint32_t **operational_current**
: operational Current in 10mA units

uint32_t **min_voltage**
: Minimum Voltage in 50mV units.

uint32_t **max_voltage**
: Maximum Voltage in 50mV units.

enum *pdo_type* **type**
: Variable supply.

  SET TO PDO_VARIABLE

struct pd_variable_supply_pdo_sink

uint32_t **raw_value**
: Raw PDO value.

union **pd_battery_supply_pdo_source**
: *#include <usbc_pd.h>* Create a Battery Supply PDO Source value See Table 6-12 Battery Supply PDO - Source.

Public Members

uint32_t **max_power**
: Maximum Allowable Power in 250mW units.

uint32_t **min_voltage**
: Minimum Voltage in 50mV units.

uint32_t **max_voltage**
: Maximum Voltage in 50mV units.

enum *pdo_type* **type**
: Battery supply.

  SET TO PDO_BATTERY

struct pd_battery_supply_pdo_source

uint32_t **raw_value**
: Raw PDO value.

union **pd_battery_supply_pdo_sink**
: *#include <usbc_pd.h>* Create a Battery Supply PDO Sink value See Table 6-16 Battery Supply PDO - Sink.

Public Members

uint32_t **operational_power**
Operational Power in 250mW units.

uint32_t **min_voltage**
Minimum Voltage in 50mV units.

uint32_t **max_voltage**
Maximum Voltage in 50mV units.

enum *pdo_type* **type**
Battery supply.
SET TO PDO_BATTERY

struct pd_battery_supply_pdo_sink

uint32_t **raw_value**
Raw PDO value.

union **pd_augmented_supply_pdo_source**
#include <usbc_pd.h> Create Augmented Supply PDO Source value See Table 6-13 Programmable Power Supply APDO - Source.

Public Members

uint32_t **max_current**
Maximum Current in 50mA increments.

uint32_t **reserved0**
Reserved – Shall be set to zero.

uint32_t **min_voltage**
Minimum Voltage in 100mV increments.

uint32_t **reserved1**
Reserved – Shall be set to zero.

uint32_t **max_voltage**
Maximum Voltage in 100mV increments.

uint32_t **reserved2**
Reserved – Shall be set to zero.

uint32_t **pps_power_limited**
PPS Power Limited.

uint32_t **reserved3**

00b – Programmable Power Supply 01b...11b - Reserved, Shall Not be used Setting as reserved because it defaults to 0 when not set.

enum *pdo_type* **type**

Augmented Power Data Object (APDO).

SET TO PDO_AUGMENTED

struct pd_augmented_supply_pdo_source

uint32_t **raw_value**

Raw PDO value.

union **pd_augmented_supply_pdo_sink**

#include <usbc_pd.h> Create Augmented Supply PDO Sink value See Table 6-17 Programmable Power Supply APDO - Sink.

Public Members

uint32_t **max_current**

Maximum Current in 50mA increments.

uint32_t **reserved0**

Reserved – Shall be set to zero.

uint32_t **min_voltage**

Minimum Voltage in 100mV increments.

uint32_t **reserved1**

Reserved – Shall be set to zero.

uint32_t **max_voltage**

Maximum Voltage in 100mV increments.

uint32_t **reserved2**

Reserved – Shall be set to zero.

uint32_t **reserved3**

00b – Programmable Power Supply 01b...11b - Reserved, Shall Not be used Setting as reserved because it defaults to 0 when not set.

enum *pdo_type* **type**

Augmented Power Data Object (APDO).

SET TO PDO_AUGMENTED

struct pd_augmented_supply_pdo_sink

uint32_t **raw_value**

Raw PDO value.

union **pd_rdo**

#include <*usbc_pd.h*> The Request Data Object (RDO) Shall be returned by the Sink making a request for power.

See Section 6.4.2 Request Message

Public Members

uint32_t **min_or_max_operating_current**

Operating Current 10mA units NOTE: If Give Back Flag is zero, this field is the Maximum Operating Current.

If Give Back Flag is one, this field is the Minimum Operating Current.

uint32_t **operating_current**

Operating current in 10mA units.

Operating Current 50mA units.

uint32_t **reserved0**

Reserved - Shall be set to zero.

uint32_t **unchunked_ext_msg_supported**

Unchunked Extended Messages Supported.

uint32_t **no_usb_suspend**

No USB Suspend.

uint32_t **usb_comm_capable**

USB Communications Capable.

uint32_t **cap_mismatch**

Capability Mismatch.

uint32_t **giveback**

Give Back Flag.

uint32_t **object_pos**

Object Position (000b is Reserved and Shall Not be used)

uint32_t **reserved1**

Reserved - Shall be set to zero.

struct *pd_rdo* **fixed**

Create a Fixed RDO value See Table 6-19 Fixed and Variable Request Data Object.

struct *pd_rdo* **variable**

Create a Variable RDO value See Table 6-19 Fixed and Variable Request Data Object.

uint32_t **min_operating_power**

Minimum Operating Power in 250mW units.

uint32_t **operating_power**

Operating power in 250mW units.

struct *pd_rdo* **battery**

Create a Battery RDO value See Table 6-20 Battery Request Data Object.

uint32_t **output_voltage**

Output Voltage in 20mV units.

uint32_t **reserved2**

Reserved - Shall be set to zero.

uint32_t **reserved3**

Reserved - Shall be set to zero.

struct *pd_rdo* **augmented**

Create an Augmented RDO value See Table 6-22 Programmable Request Data Object.

uint32_t **raw_value**

Raw RDO value.

struct **pd_msg**

#include <*usbc_pd.h*> Power Delivery message.

Public Members

enum *pd_packet_type* **type**

Type of this packet.

union *pd_header* **header**

Header of this message.

uint32_t **len**

Length of bytes in data.

uint8_t **data**[260]

Message data.

7.6.50 Time-aware General-Purpose Input/Output (TGPIO)

Overview

Configuration Options

Related configuration options:

- CONFIG_TIMEAWARE_GPIO

API Reference

Related code samples

Time-aware GPIO
Synchronize clocks.

group tgpio_interface

Time-aware GPIO Interface.

Since
3.5

Version
0.1.0

Enums

enum tgpio_pin_polarity

Event polarity.

Values:

enumerator TGPIO_RISING_EDGE = 0

enumerator TGPIO_FALLING_EDGE

enumerator TGPIO_TOGGLE_EDGE

Functions

int tgpio_port_get_time(const struct *device* *dev, uint64_t *current_time)

Get time from ART timer.

Parameters

- dev – TGPIO device
- current_time – Pointer to store timer value in cycles

Returns
0 if successful

Returns
negative errno code on failure.

int **tgpio_port_get_cycles_per_second**(const struct *device* *dev, uint32_t *cycles)

Get current running rate.

Parameters

- **dev** – TGPIO device
- **cycles** – pointer to store current running frequency

Returns
0 if successful, negative errno code on failure.

int **tgpio_pin_disable**(const struct *device* *dev, uint32_t pin)

Disable operation on pin.

Parameters

- **dev** – TGPIO device
- **pin** – TGPIO pin

Returns
0 if successful, negative errno code on failure.

int **tgpio_pin_config_ext_timestamp**(const struct *device* *dev, uint32_t pin, uint32_t event_polarity)

Enable/Continue operation on pin.

Parameters

- **dev** – TGPIO device
- **pin** – TGPIO pin
- **event_polarity** – TGPIO pin event polarity

Returns
0 if successful, negative errno code on failure.

int **tgpio_pin_periodic_output**(const struct *device* *dev, uint32_t pin, uint64_t start_time, uint64_t repeat_interval, bool periodic_enable)

Enable periodic pulse generation on a pin.

Parameters

- **dev** – TGPIO device
- **pin** – TGPIO pin
- **start_time** – start_time of first pulse in hw cycles
- **repeat_interval** – repeat interval between two pulses in hw cycles
- **periodic_enable** – enables periodic mode if 'true' is passed.

Returns
0 if successful, negative errno code on failure.

int **tgpio_pin_read_ts_ec**(const struct *device* *dev, uint32_t pin, uint64_t *timestamp, uint64_t *event_count)

Read timestamp and event counter from TGPIO.

Parameters

- **dev** – TGPIO device
- **pin** – TGPIO pin

- `timestamp` – timestamp of the last pulse received
- `event_count` – number of pulses received since the pin is enabled

Returns
0 if successful, negative errno code on failure.

7.6.51 Video

The video driver API offers a generic interface to video devices.

Basic Operation

Video Device A video device is the abstraction of a hardware or software video function, which can produce, process, consume or transform video data. The video API is designed to offer flexible way to create, handle and combine various video devices.

Endpoint Each video device can have one or more endpoints. Output endpoints configure video output function and generate data. Input endpoints configure video input function and consume data.

Video Buffer A video buffer provides the transport mechanism for the data. There is no particular requirement on the content. The requirement for the content is defined by the endpoint format. A video buffer can be queued to a device endpoint for filling (input ep) or consuming (output ep) operation, once the operation is achieved, buffer can be dequeued for post-processing, release or reuse.

Controls A video control is accessed and identified by a CID (control identifier). It represents a video control property. Different devices will have different controls available which can be generic, related to a device class or vendor specific. The set/get control functions provide a generic scalable interface to handle and create controls.

Configuration Options

Related configuration options:

- `CONFIG_VIDEO`

API Reference

Related code samples

Video TCP server sink
Capture video frames and send them over the network to a TCP client.

Video capture
Use the video API to retrieve video frames from a capture device.

group `video_interface`

Video Interface.

Since
2.1

Version
1.0.0

Defines

`video_fourcc`(a, b, c, d)

Typedefs

typedef int (*`video_api_set_format_t`)(const struct *device* *dev, enum *video_endpoint_id* ep, struct *video_format* *fmt)

Set video format.

See *video_set_format()* for argument descriptions.

typedef int (*`video_api_get_format_t`)(const struct *device* *dev, enum *video_endpoint_id* ep, struct *video_format* *fmt)

Get current video format.

See *video_get_format()* for argument descriptions.

typedef int (*`video_api_enqueue_t`)(const struct *device* *dev, enum *video_endpoint_id* ep, struct *video_buffer* *buf)

Enqueue a buffer in the driver's incoming queue.

See *video_enqueue()* for argument descriptions.

typedef int (*`video_api_dequeue_t`)(const struct *device* *dev, enum *video_endpoint_id* ep, struct *video_buffer* **buf, *k_timeout_t* timeout)

Dequeue a buffer from the driver's outgoing queue.

See *video_dequeue()* for argument descriptions.

typedef int (*`video_api_flush_t`)(const struct *device* *dev, enum *video_endpoint_id* ep, bool cancel)

Flush endpoint buffers, buffer are moved from incoming queue to outgoing queue.

See *video_flush()* for argument descriptions.

typedef int (*`video_api_stream_start_t`)(const struct *device* *dev)

Start the capture or output process.

See *video_stream_start()* for argument descriptions.

typedef int (*`video_api_stream_stop_t`)(const struct *device* *dev)

Stop the capture or output process.

See *video_stream_stop()* for argument descriptions.

typedef int (*`video_api_set_ctrl_t`)(const struct *device* *dev, unsigned int cid, void *value)

Set a video control value.

See *video_set_ctrl()* for argument descriptions.

typedef int (*`video_api_get_ctrl_t`)(const struct *device* *dev, unsigned int cid, void *value)

Get a video control value.

See *video_get_ctrl()* for argument descriptions.

typedef int (*`video_api_get_caps_t`)(const struct *device* *dev, enum *video_endpoint_id* ep, struct *video_caps* *caps)

Get capabilities of a video endpoint.

See *video_get_caps()* for argument descriptions.

typedef int (*`video_api_set_signal_t`)(const struct *device* *dev, enum *video_endpoint_id* ep, struct *k_poll_signal* *signal)

Register/Unregister poll signal for buffer events.

See *video_set_signal()* for argument descriptions.

Enums

enum `video_endpoint_id`

video_endpoint_id enum

Identify the video device endpoint.

Values:

enumerator `VIDEO_EP_NONE`

enumerator `VIDEO_EP_ANY`

enumerator `VIDEO_EP_IN`

enumerator `VIDEO_EP_OUT`

enum `video_signal_result`

video_event enum

Identify video event.

Values:

enumerator `VIDEO_BUF_DONE`

enumerator `VIDEO_BUF_ABORTED`

enumerator `VIDEO_BUF_ERROR`

Functions

static inline int **video_set_format**(const struct *device* *dev, enum *video_endpoint_id* ep, struct *video_format* *fmt)

Set video format.

Configure video device with a specific format.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **ep** – Endpoint ID.
- **fmt** – Pointer to a video format struct.

Return values

- **0** – Is successful.
- **-EINVAL** – If parameters are invalid.
- **-ENOTSUP** – If format is not supported.
- **-EIO** – General input / output error.

static inline int **video_get_format**(const struct *device* *dev, enum *video_endpoint_id* ep, struct *video_format* *fmt)

Get video format.

Get video device current video format.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **ep** – Endpoint ID.
- **fmt** – Pointer to video format struct.

Return values

pointer – to video format

static inline int **video_enqueue**(const struct *device* *dev, enum *video_endpoint_id* ep, struct *video_buffer* *buf)

Enqueue a video buffer.

Enqueue an empty (capturing) or filled (output) video buffer in the driver's endpoint incoming queue.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **ep** – Endpoint ID.
- **buf** – Pointer to the video buffer.

Return values

- **0** – Is successful.
- **-EINVAL** – If parameters are invalid.
- **-EIO** – General input / output error.

static inline int **video_dequeue**(const struct *device* *dev, enum *video_endpoint_id* ep, struct *video_buffer* **buf, *k_timeout_t* timeout)

Dequeue a video buffer.

Dequeue a filled (capturing) or displayed (output) buffer from the driver's endpoint outgoing queue.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `ep` – Endpoint ID.
- `buf` – Pointer a video buffer pointer.
- `timeout` – Timeout

Return values

- `0` – Is successful.
- `-EINVAL` – If parameters are invalid.
- `-EIO` – General input / output error.

static inline int `video_flush`(const struct *device* *dev, enum *video_endpoint_id* ep, bool cancel)

Flush endpoint buffers.

A call to flush finishes when all endpoint buffers have been moved from incoming queue to outgoing queue. Either because canceled or fully processed through the video function.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `ep` – Endpoint ID.
- `cancel` – If true, cancel buffer processing instead of waiting for completion.

Return values

`0` – Is successful, -ERRNO code otherwise.

static inline int `video_stream_start`(const struct *device* *dev)

Start the video device function.

video_stream_start is called to enter 'streaming' state (capture, output...). The driver may receive buffers with *video_enqueue()* before video_stream_start is called. If driver/device needs a minimum number of buffers before being able to start streaming, then driver set the min_vbuf_count to the related endpoint capabilities.

Return values

- `0` – Is successful.
- `-EIO` – General input / output error.

static inline int `video_stream_stop`(const struct *device* *dev)

Stop the video device function.

On video_stream_stop, driver must stop any transactions or wait until they finish.

Return values

- `0` – Is successful.
- `-EIO` – General input / output error.

static inline int **video_get_caps**(const struct *device* *dev, enum *video_endpoint_id* ep, struct *video_caps* *caps)

Get the capabilities of a video endpoint.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `ep` – Endpoint ID.
- `caps` – Pointer to the *video_caps* struct to fill.

Return values

`0` – Is successful, -ERRNO code otherwise.

static inline int **video_set_ctrl**(const struct *device* *dev, unsigned int cid, void *value)

Set the value of a control.

This set the value of a video control, value type depends on control ID, and must be interpreted accordingly.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `cid` – Control ID.
- `value` – Pointer to the control value.

Return values

- `0` – Is successful.
- `-EINVAL` – If parameters are invalid.
- `-ENOTSUP` – If format is not supported.
- `-EIO` – General input / output error.

static inline int **video_get_ctrl**(const struct *device* *dev, unsigned int cid, void *value)

Get the current value of a control.

This retrieve the value of a video control, value type depends on control ID, and must be interpreted accordingly.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `cid` – Control ID.
- `value` – Pointer to the control value.

Return values

- `0` – Is successful.
- `-EINVAL` – If parameters are invalid.
- `-ENOTSUP` – If format is not supported.
- `-EIO` – General input / output error.

static inline int **video_set_signal**(const struct *device* *dev, enum *video_endpoint_id* ep, struct *k_poll_signal* *signal)

Register/Unregister k_poll signal for a video endpoint.

Register a poll signal to the endpoint, which will be signaled on frame completion (done, aborted, error). Registering a NULL poll signal unregisters any previously registered signal.

Parameters

- **dev** – Pointer to the device structure for the driver instance.
- **ep** – Endpoint ID.
- **signal** – Pointer to *k_poll_signal*

Return values
0 – Is successful, -ERRNO code otherwise.

struct *video_buffer* ***video_buffer_aligned_alloc**(size_t size, size_t align)
Allocate aligned video buffer.

Parameters

- **size** – Size of the video buffer (in bytes).
- **align** – Alignment of the requested memory, must be a power of two.

Return values
pointer – to allocated video buffer

struct *video_buffer* ***video_buffer_alloc**(size_t size)
Allocate video buffer.

Parameters

- **size** – Size of the video buffer (in bytes).

Return values
pointer – to allocated video buffer

void **video_buffer_release**(struct *video_buffer* *buf)
Release a video buffer.

Parameters

- **buf** – Pointer to the video buffer to release.

struct **video_format**
#include <*video.h*> Video format structure.

Used to configure frame format.

Public Members

uint32_t **pixelformat**
FourCC pixel format value (Video pixel formats)

uint32_t **width**
frame width in pixels.

uint32_t **height**
frame height in pixels.

uint32_t **pitch**
line stride.

This is the number of bytes that needs to be added to the address in the first pixel of a row in order to go to the address of the first pixel of the next row (>=width).

struct `video_format_cap`

#include <*video.h*> Video format capability.

Used to describe a video endpoint format capability.

Public Members

uint32_t `pixelformat`

FourCC pixel format value (Video pixel formats).

uint32_t `width_min`

minimum supported frame width in pixels.

uint32_t `width_max`

maximum supported frame width in pixels.

uint32_t `height_min`

minimum supported frame height in pixels.

uint32_t `height_max`

maximum supported frame height in pixels.

uint16_t `width_step`

width step size in pixels.

uint16_t `height_step`

height step size in pixels.

struct `video_caps`

#include <*video.h*> Video format capabilities.

Used to describe video endpoint capabilities.

Public Members

const struct *video_format_cap* *`format_caps`

list of video format capabilities (zero terminated).

uint8_t `min_vbuf_count`

minimal count of video buffers to enqueue before being able to start the stream.

struct `video_buffer`

#include <*video.h*> Video buffer structure.

Represent a video frame.

Public Members

void *driver_data
pointer to driver specific data.

uint8_t *buffer
pointer to the start of the buffer.

uint32_t size
size of the buffer in bytes.

uint32_t bytesused
number of bytes occupied by the valid data in the buffer.

uint32_t timestamp
time reference in milliseconds at which the last data byte was actually received for input endpoints or to be consumed for output endpoints.

struct video_driver_api
#include <video.h>

group video_controls
Video controls.

Control classes

VIDEO_CTRL_CLASS_GENERIC
Generic class controls.

VIDEO_CTRL_CLASS_CAMERA
Camera class controls.

VIDEO_CTRL_CLASS_MPEG
MPEG-compression controls.

VIDEO_CTRL_CLASS_JPEG
JPEG-compression controls.

VIDEO_CTRL_CLASS_VENDOR
Vendor-specific class controls.

Generic class control IDs

VIDEO_CID_HFLIP
Mirror the picture horizontally.

VIDEO_CID_VFLIP
Mirror the picture vertically.

Camera class control IDs

VIDEO_CID_CAMERA_EXPOSURE

VIDEO_CID_CAMERA_GAIN

VIDEO_CID_CAMERA_ZOOM

VIDEO_CID_CAMERA_BRIGHTNESS

VIDEO_CID_CAMERA_SATURATION

VIDEO_CID_CAMERA_WHITE_BAL

VIDEO_CID_CAMERA_CONTRAST

VIDEO_CID_CAMERA_COLORBAR

VIDEO_CID_CAMERA_QUALITY

7.6.52 Watchdog

Overview

API Reference

Related code samples

Watchdog
Use the watchdog driver API to reset the board when it gets stuck in an infinite loop.

group watchdog_interface
Watchdog Interface.

Since
1.0

Version
1.0.0

Watchdog options

WDT_OPT_PAUSE_IN_SLEEP
Pause watchdog timer when CPU is in sleep state.

`WDT_OPT_PAUSE_HALTED_BY_DBG`
: Pause watchdog timer when CPU is halted by the debugger.

Watchdog behavior flags

`WDT_FLAG_RESET_NONE`
: Reset: none.

`WDT_FLAG_RESET_CPU_CORE`
: Reset: CPU core.

`WDT_FLAG_RESET_SOC`
: Reset: SoC.

Typedefs

typedef void (*`wdt_callback_t`)(const struct *device* *dev, int channel_id)
: Watchdog callback.

  Param dev
  : Watchdog device instance.

  Param channel_id
  : Channel identifier.

Functions

int `wdt_setup`(const struct *device* *dev, uint8_t options)
: Set up watchdog instance.

  This function is used for configuring global watchdog settings that affect all timeouts. It should be called after installing timeouts. After successful return, all installed timeouts are valid and must be serviced periodically by calling *wdt_feed()*.

  Parameters

  - `dev` – Watchdog device instance.
  - `options` – Configuration options (see *WDT_OPT*).

  Return values

  - `0` – If successful.
  - `-ENOTSUP` – If any of the set options is not supported.
  - `-EBUSY` – If watchdog instance has been already setup.
  - `-errno` – In case of any other failure.

int `wdt_disable`(const struct *device* *dev)
: Disable watchdog instance.

  This function disables the watchdog instance and automatically uninstalls all timeouts. To set up a new watchdog, install timeouts and call *wdt_setup()* again. Not all watchdogs can be restarted after they are disabled.

Parameters

- `dev` – Watchdog device instance.

Return values

- `0` – If successful.
- `-EFAULT` – If watchdog instance is not enabled.
- `-EPERM` – If watchdog can not be disabled directly by application code.
- `-errno` – In case of any other failure.

static inline int **wdt_install_timeout**(const struct *device* *dev, const struct *wdt_timeout_cfg* *cfg)

Install a new timeout.

> **Note**
>
> This function must be used before *wdt_setup()*. Changes applied here have no effects until *wdt_setup()* is called.

Parameters

- `dev` – Watchdog device instance.
- `cfg` – **[in]** Timeout configuration.

Return values

- `channel_id` – If successful, a non-negative value indicating the index of the channel to which the timeout was assigned. This value is supposed to be used as the parameter in calls to *wdt_feed()*.
- `-EBUSY` – If timeout can not be installed while watchdog has already been setup.
- `-ENOMEM` – If no more timeouts can be installed.
- `-ENOTSUP` – If any of the set flags is not supported.
- `-EINVAL` – If any of the window timeout value is out of possible range. This value is also returned if watchdog supports only one timeout value for all timeouts and the supplied timeout window differs from windows for alarms installed so far.
- `-errno` – In case of any other failure.

int **wdt_feed**(const struct *device* *dev, int channel_id)

Feed specified watchdog timeout.

Parameters

- `dev` – Watchdog device instance.
- `channel_id` – Channel index.

Return values

- `0` – If successful.
- `-EAGAIN` – If completing the feed operation would stall the caller, for example due to an in-progress watchdog operation such as a previous *wdt_feed()* call.
- `-EINVAL` – If there is no installed timeout for supplied channel.

- `-errno` – In case of any other failure.

struct `wdt_window`

#include <watchdog.h> Watchdog timeout window.

Each installed timeout needs feeding within the specified time window, otherwise the watchdog will trigger. If the watchdog instance does not support window timeouts then min value must be equal to 0.

> **Note**
>
> If specified values can not be precisely set they are always rounded up.

Public Members

uint32_t `min`

Lower limit of watchdog feed timeout in milliseconds.

uint32_t `max`

Upper limit of watchdog feed timeout in milliseconds.

struct `wdt_timeout_cfg`

#include <watchdog.h> Watchdog timeout configuration.

Public Members

struct *wdt_window* `window`

Timing parameters of watchdog timeout.

wdt_callback_t `callback`

Timeout callback (can be `NULL`).

struct *wdt_timeout_cfg* *`next`

Pointer to the next timeout configuration.

This field is only available if `CONFIG_WDT_MULTISTAGE` is enabled (watchdogs with staged timeouts functionality). Value must be `NULL` for single stage timeout.

uint8_t `flags`

Flags (see *WDT_FLAGS*).

7.7 Pin Control

This is a high-level guide to pin control. See *Pin Control API* for API reference material.

7.7.1 Introduction

The hardware blocks that control pin multiplexing and pin configuration parameters such as pin direction, pull-up/down resistors, etc. are named **pin controllers**. The pin controller's main users are SoC hardware peripherals, since the controller enables exposing peripheral signals, like for example, map `I2C0 SDA` signal to pin `PX0`. Not only that, but it usually allows configuring certain pin settings that are necessary for the correct functioning of a peripheral, for example, the slew-rate depending on the operating frequency. The available configuration options are vendor/SoC dependent and can range from simple pull-up/down options to more advanced settings such as debouncing, low-power modes, etc.

The way pin control is implemented in hardware is vendor/SoC specific. It is common to find a *centralized* approach, that is, all pin configuration parameters are controlled by a single hardware block (typically named pinmux), including signal mapping. The figure below illustrates this approach. `PX0` can be mapped to `UART0_TX`, `I2C0_SCK` or `SPI0_MOSI` depending on the `AF` control bits. Other configuration parameters such as pull-up/down are controlled in the same block via `CONFIG` bits. This model is used by several SoC families, such as many from NXP and STM32.

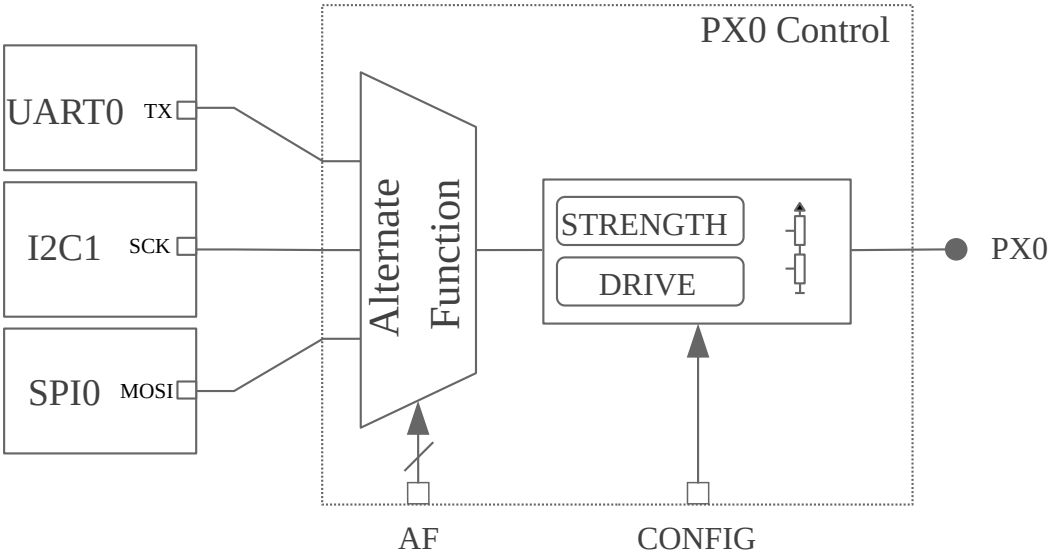


Fig. 3: Example of pin control centralized into a single per-pin block

Other vendors/SoCs use a *distributed* approach. In such case, the pin mapping and configuration are controlled by multiple hardware blocks. The figure below illustrates a distributed approach where pin mapping is controlled by peripherals, such as in Nordic nRF SoCs.

From a user perspective, there is no difference in pin controller usage regardless of the hardware implementation: a user will always apply a state. The only difference lies in the driver implementation. In general, implementing a pin controller driver for a hardware that uses a distributed approach requires more effort, since the driver needs to gather knowledge of peripheral dependent registers.

Pin control vs. GPIO

Some functionality covered by a pin controller driver overlaps with GPIO drivers. For example, pull-up/down resistors can usually be enabled by both the pin control driver and the GPIO driver. In Zephyr context, the pin control driver purpose is to perform peripheral signal multiplexing and configuration of other pin parameters required for the correct operation of that peripheral. Therefore, the main users of the pin control driver are SoC peripherals. In contrast, GPIO drivers are for general purpose control of a pin, that is, when its logic level is read or controlled manually.

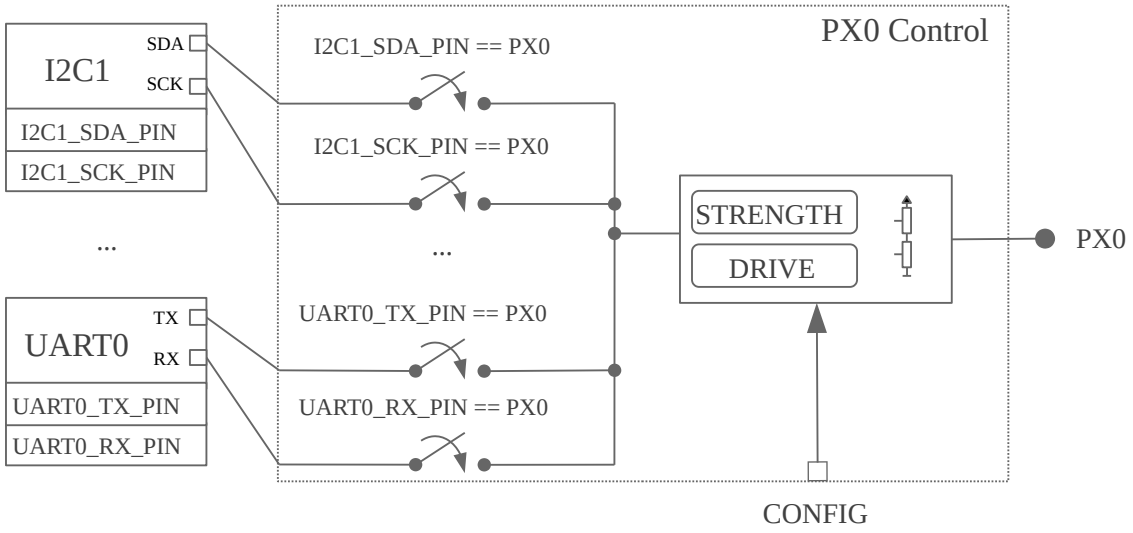


Fig. 4: Example pin control distributed between peripheral registers and per-pin block

7.7.2 State model

For a device driver to operate correctly, a certain pin configuration needs to be applied. Some device drivers require a static configuration, usually set up at initialization time. Others need to change the configuration at runtime depending on the operating conditions, for example, to enable a low-power mode when suspending the device. Such requirements are modeled using **states**, a concept that has been adapted from the one in the Linux kernel. Each device driver owns a set of states. Each state has a unique name and contains a full pin configuration set (see the figure below). This effectively means that states are independent of each other, so they do not need to be applied in any specific order. Another advantage of the state model is that it isolates device drivers from pin configuration.

Table 2: Example pin configuration encoded using the states model

`UART0` peripheral			
`default` state		`sleep` state	
TX	• Pin: PA0 • Pull: NONE • Low Power: NO	TX	• Pin: PA0 • Pull: NONE • Low Power: YES
RX	• Pin: PA1 • Pull: UP • Low Power: NO	RX	• Pin: PA1 • Pull: NONE • Low Power: YES

Standard states

The name assigned to pin control states or the number of them is up to the device driver requirements. In many cases a single state applied at initialization time will be sufficient, but in some other cases more will be required. In order to make things consistent, a naming convention

has been established for the most common use cases. The figure below details the standardized states and its purpose.

Table 3: Standardized state names

State	Identifier	Purpose
default	*PINCTRL_STATE_DEFAULT*	State of the pins when the device is in operational state
sleep	*PINCTRL_STATE_SLEEP*	State of the pins when the device is in low power or sleep modes

Note that other standard states could be introduced in the future.

Custom states

Some device drivers may require using custom states beyond the standard ones. To achieve that, the device driver needs to have in its scope definitions for the custom state identifiers named as `PINCTRL_STATE_{STATE_NAME}`, where `{STATE_NAME}` is the capitalized state name. For example, if `mystate` has to be supported, a definition named `PINCTRL_STATE_MYSTATE` needs to be in the driver's scope.

Note

It is important that custom state identifiers start from *PINCTRL_STATE_PRIV_START*

If custom states need to be accessed from outside the driver, for example to perform dynamic pin control, custom identifiers should be placed in a header that is publicly accessible.

Skipping states

In most situations, the states defined in Devicetree will be the ones used in the compiled firmware. However, there are some cases where certain states will be conditionally used depending on a compilation flag. A typical case is the `sleep` state. This state is only used in practice if `CONFIG_PM` or `CONFIG_PM_DEVICE` is enabled. If a firmware variant without these power management configurations is needed, one should in theory remove the `sleep` state from Devicetree to not waste ROM space storing such unused state.

States can be skipped by the `pinctrl` Devicetree macros if a definition named `PINCTRL_SKIP_{STATE_NAME}` expanding to `1` is present when pin control configuration is defined. In case of the `sleep` state, the `pinctrl` API already provides such definition conditional to the availability of device power management:

```
#if !defined(CONFIG_PM) && !defined(CONFIG_PM_DEVICE)
/** Out of power management configurations, ignore "sleep" state. */
#define PINCTRL_SKIP_SLEEP 1
#endif
```

7.7.3 Dynamic pin control

Dynamic pin control refers to the capability of changing pin configuration at runtime. This feature can be useful in situations where the same firmware needs to run onto slightly different boards, each having a peripheral routed at a different set of pins. This feature can be enabled by setting `CONFIG_PINCTRL_DYNAMIC`.

Note

Dynamic pin control should only be used on devices that have not been initialized. Changing pin configurations while a device is operating may lead to unexpected behavior. Since Zephyr does not support device de-initialization yet, this functionality should only be used during early boot stages.

One of the effects of enabling dynamic pin control is that *pinctrl_dev_config* will be stored in RAM instead of ROM (not states or pin configurations, though). The user can then use *pinctrl_update_states()* to update the states stored in *pinctrl_dev_config* with a new set. This effectively means that the device driver will apply the pin configurations stored in the updated states when it applies a state.

7.7.4 Devicetree representation

Because Devicetree is meant to describe hardware, it is the natural choice when it comes to storing pin control configuration. In the following sections you will find an overview on how states and pin configurations are represented in Devicetree.

States

Given a device, each of its pin control state is represented in Devicetree by `pinctrl-N` properties, being `N` the state index starting from zero. The `pinctrl-names` property is then used to assign a unique identifier for each state property by index, for example, `pinctrl-names` list entry 0 is the name for `pinctrl-0`.

```
periph0: periph@0 {
    ...
    /* state 0 ("default") */
    pinctrl-0 = <...>;
    ...
    /* state N ("mystate") */
    pinctrl-N = <...>;
    /* names for state 0 up to state N */
    pinctrl-names = "default", ..., "mystate";
    ...
};
```

Pin configuration

There are multiple ways to represent the pin configurations in Devicetree. However, all end up encoding the same information: the pin multiplexing and the pin configuration parameters. For example, `UART_RX` is mapped to `PX0` and pull-up is enabled. The representation choice largely depends on each vendor/SoC, so the Devicetree binding files for the pin control drivers are the best place to look for details.

A popular and versatile option is shown in the example below. One of the advantages of this choice is the grouping capability based on shared pin configuration. This allows to reduce the verbosity of the pin control definitions. Another advantage is that the pin configuration parameters for a particular state are enclosed in a single Devicetree node.

```
/* board.dts */
#include "board-pinctrl.dtsi"
```

(continues on next page)

(continued from previous page)

```
&periph0 {
    pinctrl-0 = <&periph0_default>;
    pinctrl-names = "default";
};
```

```
/* vnd-soc-pkgxx.h
 * File with valid mappings for a specific package (may be autogenerated).
 * This file is optional, but recommended.
 */
...
#define PERIPH0_SIGA_PX0 VNDSOC_PIN(X, 0, MUX0)
#define PERIPH0_SIGB_PY7 VNDSOC_PIN(Y, 7, MUX4)
#define PERIPH0_SIGC_PZ1 VNDSOC_PIN(Z, 1, MUX2)
...
```

```
/* board-pinctrl.dtsi */
#include <vnd-soc-pkgxx.h>

&pinctrl {
    /* Node with pin configuration for default state */
    periph0_default: periph0_default {
        group1 {
            /* Mappings: PERIPH0_SIGA -> PX0, PERIPH0_SIGC -> PZ1 */
            pinmux = <PERIPH0_SIGA_PX0>, <PERIPH0_SIGC_PZ1>;
            /* Pins PX0 and PZ1 have pull-up enabled */
            bias-pull-up;
        };
        ...
        groupN {
            /* Mappings: PERIPH0_SIGB -> PY7 */
            pinmux = <PERIPH0_SIGB_PY7>;
        };
    };
};
```

Another popular model is based on having a node for each pin configuration and state. While this model may lead to shorter board pin control files, it also requires to have one node for each pin mapping and state, since in general, nodes can not be re-used for multiple states. This method is discouraged if autogeneration is not an option.

Note

Because all Devicetree information is parsed into a C header, it is important to make sure its size is kept to a minimum. For this reason it is important to prefix pre-generated nodes with `/omit-if-no-ref/`. This prefix makes sure that the node is discarded when not used.

```
/* board.dts */
#include "board-pinctrl.dtsi"

&periph0 {
    pinctrl-0 = <&periph0_siga_px0_default &periph0_sigb_py7_default
                 &periph0_sigc_pz1_default>;
    pinctrl-names = "default";
};
```

```
/* vnd-soc-pkgxx.dtsi
 * File with valid nodes for a specific package (may be autogenerated).
 * This file is optional, but recommended.
```

(continues on next page)

(continued from previous page)

```
 */

&pinctrl {
    /* Mapping for PERIPH0_SIGA -> PX0, to be used for default state */
    /omit-if-no-ref/ periph0_siga_px0_default: periph0_siga_px0_default {
        pinmux = <VNDSOC_PIN(X, 0, MUX0)>;
    };

    /* Mapping for PERIPH0_SIGB -> PY7, to be used for default state */
    /omit-if-no-ref/ periph0_sigb_py7_default: periph0_sigb_py7_default {
        pinmux = <VNDSOC_PIN(Y, 7, MUX4)>;
    };

    /* Mapping for PERIPH0_SIGC -> PZ1, to be used for default state */
    /omit-if-no-ref/ periph0_sigc_pz1_default: periph0_sigc_pz1_default {
        pinmux = <VNDSOC_PIN(Z, 1, MUX2)>;
    };
};
```

```
/* board-pinctrl.dts */
#include <vnd-soc-pkgxx.dtsi>

/* Enable pull-up for PX0 (default state) */
&periph0_siga_px0_default {
    bias-pull-up;
};

/* Enable pull-up for PZ1 (default state) */
&periph0_sigc_pz1_default {
    bias-pull-up;
};
```

Note

It is discouraged to add pin configuration defaults in pre-defined nodes. In general, pin configurations depend on the board design or on the peripheral working conditions, so the decision should be made by the board. For example, enabling a pull-up by default may not always be desired because the board already has one or because its value depends on the operating bus speed. Another downside of defaults is that user may not be aware of them, for example:

```
/* not evident that "periph0_siga_px0_default" also implies "bias-pull-up" */
/omit-if-no-ref/ periph0_siga_px0_default: periph0_siga_px0_default {
    pinmux = <VNDSOC_PIN(X, 0, MUX0)>;
    bias-pull-up;
};
```

7.7.5 Implementation guidelines

Pin control drivers

Pin control drivers need to implement a single function: *pinctrl_configure_pins()*. This function receives an array of pin configurations that need to be applied. Furthermore, if `CONFIG_PINCTRL_STORE_REG` is set, it also receives the associated device register address for the given pins. This information may be required by some drivers to perform device specific actions.

The pin configuration is stored in an opaque type that is vendor/SoC dependent: `pinctrl_soc_pin_t`. This type needs to be defined in a header named `pinctrl_soc.h` file that is in the

Zephyr's include path. It can range from a simple integer value to a struct with multiple fields. `pinctrl_soc.h` also needs to define a macro named `Z_PINCTRL_STATE_PINS_INIT` that accepts two arguments: a node identifier and a property name (`pinctrl-N`). With this information the macro needs to define an initializer for all pin configurations contained within the `pinctrl-N` property of the given node.

Regarding Devicetree pin configuration representation, vendors can decide which option is better for their devices. However, the following guidelines should be followed:

- Use `pinctrl-N` (N=0, 1, ...) and `pinctrl-names` properties to define pin control states. These properties are defined in `dts/bindings/pinctrl/pinctrl-device.yaml`.
- Use standard pin configuration properties as defined in `dts/bindings/pinctrl/pincfg-node.yaml`.

Representations not following these guidelines may be accepted if they are already used by the same vendor in other operating systems, e.g. Linux.

Device drivers

In this section you will find some tips on how a device driver should use the `pinctrl` API to successfully configure the pins it needs.

The device compatible needs to be modified in the corresponding binding so that the `pinctrl-device.yaml` is included. For example:

```
include: [base.yaml, pinctrl-device.yaml]
```

This file is needed to add `pinctrl-N` and `pinctrl-names` properties to the device.

From a device driver perspective there are two steps that need to be performed to be able to use the `pinctrl` API. First, the pin control configuration needs to be defined. This includes all states and pins. *PINCTRL_DT_DEFINE* or *PINCTRL_DT_INST_DEFINE* macros should be used for this purpose. Second, a reference to the device instance *pinctrl_dev_config* needs to be stored, since it is required to later use the API. This can be achieved using the *PINCTRL_DT_DEV_CONFIG_GET* and *PINCTRL_DT_INST_DEV_CONFIG_GET* macros.

It is worth to note that the only relationship between a device and its associated pin control configuration is based on variable naming conventions. The way an instance of *pinctrl_dev_config* is named for a corresponding device instance allows to later obtain a reference to it given the device's Devicetree node identifier. This allows to minimize ROM usage, since only devices requiring pin control will own a reference to a pin control configuration.

Once the driver has defined the pin control configuration and kept a reference to it, it is ready to use the API. The most common way to apply a state is by using *pinctrl_apply_state()*. It is also possible to use the lower level function *pinctrl_apply_state_direct()* to skip state lookup if it is cached in advance (e.g. at init time). Since state lookup time is expected to be fast, it is recommended to use *pinctrl_apply_state()*.

The example below contains a complete example of a device driver that uses the `pinctrl` API.

```
/* A driver for the "mydev" compatible device */
#define DT_DRV_COMPAT mydev

...
#include <zephyr/drivers/pinctrl.h>
...

struct mydev_config {
    ...
    /* Reference to mydev pinctrl configuration */
    const struct pinctrl_dev_config *pcfg;
```

(continues on next page)

(continued from previous page)

```
    ...
};

...

static int mydev_init(const struct device *dev)
{
    const struct mydev_config *config = dev->config;
    int ret;
    ...
    /* Select "default" state at initialization time */
    ret = pinctrl_apply_state(config->pcfg, PINCTRL_STATE_DEFAULT);
    if (ret < 0) {
        return ret;
    }
    ...
}

#define MYDEV_DEFINE(i)                                                    \
    /* Define all pinctrl configuration for instance "i" */               \
    PINCTRL_DT_INST_DEFINE(i);                                             \
    ...                                                                    \
    static const struct mydev_config mydev_config_##i = {                 \
        ...                                                                \
        /* Keep a ref. to the pinctrl configuration for instance "i" */    \
        .pcfg = PINCTRL_DT_INST_DEV_CONFIG_GET(i),                         \
        ...                                                                \
    };                                                                     \
    ...                                                                    \
                                                                           \
    DEVICE_DT_INST_DEFINE(i, mydev_init, NULL, &mydev_data##i,             \
                          &mydev_config##i, ...);

DT_INST_FOREACH_STATUS_OKAY(MYDEV_DEFINE)
```

7.7.6 Pin Control API

group `pinctrl_interface`

Pin Controller Interface.

Since
3.0

Version
0.1.0

Pin control states

`PINCTRL_STATE_DEFAULT`

Default state (state used when the device is in operational state).

PINCTRL_STATE_SLEEP

Sleep state (state used when the device is in low power mode).

PINCTRL_STATE_PRIV_START

This and higher values refer to custom private states.

Defines

PINCTRL_REG_NONE

Utility macro to indicate no register is used.

PINCTRL_DT_DEV_CONFIG_DECLARE(node_id)

Declare pin control configuration for a given node identifier.

This macro should be used by tests or applications using runtime pin control to declare the pin control configuration for a device. *PINCTRL_DT_DEV_CONFIG_GET* can later be used to obtain a reference to such configuration.

Only available if CONFIG_PINCTRL_NON_STATIC is selected.

Parameters

- **node_id** – Node identifier.

PINCTRL_DT_DEFINE(node_id)

Define all pin control information for the given node identifier.

This helper macro should be called together with device definition. It defines and initializes the pin control configuration for the device represented by node_id. Each pin control state (pinctrl-0, ..., pinctrl-N) is also defined and initialized. Note that states marked to be skipped will not be defined (refer to Z_PINCTRL_SKIP_STATE for more details).

Parameters

- **node_id** – Node identifier.

PINCTRL_DT_INST_DEFINE(inst)

Define all pin control information for the given compatible index.

See also

PINCTRL_DT_DEFINE

Parameters

- **inst** – Instance number.

PINCTRL_DT_DEV_CONFIG_GET(node_id)

Obtain a reference to the pin control configuration given a node identifier.

Parameters

- **node_id** – Node identifier.

`PINCTRL_DT_INST_DEV_CONFIG_GET`(inst)

Obtain a reference to the pin control configuration given current compatible instance number.

See also

PINCTRL_DT_DEV_CONFIG_GET

Parameters

- `inst` – Instance number.

Functions

int `pinctrl_lookup_state`(const struct *pinctrl_dev_config* *config, uint8_t id, const struct *pinctrl_state* **state)

Find the state configuration for the given state id.

Parameters

- `config` – Pin controller configuration.
- `id` – Pin controller state id (see *PINCTRL_STATES*).
- `state` – Found state.

Return values

- `0` – If state has been found.
- `-ENOENT` – If the state has not been found.

int `pinctrl_configure_pins`(const pinctrl_soc_pin_t *pins, uint8_t pin_cnt, uintptr_t reg)

Configure a set of pins.

This function will configure the necessary hardware blocks to make the configuration immediately effective.

Warning

This function must never be used to configure pins used by an instantiated device driver.

Parameters

- `pins` – List of pins to be configured.
- `pin_cnt` – Number of pins.
- `reg` – Device register (optional, use *PINCTRL_REG_NONE* if not used).

Return values

- `0` – If succeeded
- `-errno` – Negative errno for other failures.

static inline int **pinctrl_apply_state_direct**(const struct *pinctrl_dev_config* *config, const struct *pinctrl_state* *state)

Apply a state directly from the provided state configuration.

Parameters

- `config` – Pin control configuration.
- `state` – State.

Return values

- `0` – If succeeded
- `-errno` – Negative errno for other failures.

static inline int **pinctrl_apply_state**(const struct *pinctrl_dev_config* *config, uint8_t id)

Apply a state from the given device configuration.

Parameters

- `config` – Pin control configuration.
- `id` – Id of the state to be applied (see *PINCTRL_STATES*).

Return values

- `0` – If succeeded.
- `-ENOENT` – If given state id does not exist.
- `-errno` – Negative errno for other failures.

struct **pinctrl_state**

#include <pinctrl.h> Pin control state configuration.

Public Members

const pinctrl_soc_pin_t ***pins**

Pin configurations.

uint8_t **pin_cnt**

Number of pin configurations.

uint8_t **id**

State identifier (see *PINCTRL_STATES*).

struct **pinctrl_dev_config**

#include <pinctrl.h> Pin controller configuration for a given device.

Public Members

uintptr_t **reg**

Device address (only available if `CONFIG_PINCTRL_STORE_REG` is enabled).

const struct *pinctrl_state* ***states**

List of state configurations.

uint8_t **state_cnt**

Number of state configurations.

Dynamic pin control

group **pinctrl_interface_dynamic**

Defines

PINCTRL_DT_STATE_PINS_DEFINE(node_id, prop)

Helper macro to define the pins of a pin control state from Devicetree.

The name of the defined state pins variable is the same used by `prop`. This macro is expected to be used in conjunction with *PINCTRL_DT_STATE_INIT*.

See also

PINCTRL_DT_STATE_INIT

Parameters

- **node_id** – Node identifier containing `prop`.
- **prop** – Property within `node_id` containing state configuration.

PINCTRL_DT_STATE_INIT(prop, state)

Utility macro to initialize a pin control state.

This macro should be used in conjunction with *PINCTRL_DT_STATE_PINS_DEFINE* when using dynamic pin control to define an alternative state configuration stored in Devicetree.

Example:

```
// board.dts

/{
     zephyr,user {
             // uart0_alt_default node contains alternative pin config
             uart0_alt_default = <&uart0_alt_default>;
     };
};
```

```
// application

PINCTRL_DT_STATE_PINS_DEFINE(DT_PATH(zephyr_user), uart0_alt_default);

static const struct pinctrl_state uart0_alt[] = {
    PINCTRL_DT_STATE_INIT(uart0_alt_default, PINCTRL_STATE_DEFAULT)
};
```

See also

PINCTRL_DT_STATE_PINS_DEFINE

Parameters

- **prop** – Property name in Devicetree containing state configuration.
- **state** – State represented by prop (see *PINCTRL_STATES*).

Functions

int **pinctrl_update_states**(struct *pinctrl_dev_config* *config, const struct *pinctrl_state* *states, uint8_t state_cnt)

Update states with a new set.

Note

In order to guarantee device drivers correct operation the same states have to be provided. For example, if default and sleep are in the current list of states, it is expected that the new array of states also contains both.

Parameters

- **config** – Pin control configuration.
- **states** – New states to be set.
- **state_cnt** – Number of new states to be set.

Return values

- **-EINVAL** – If the new configuration does not contain the same states as the current active configuration.
- **-ENOSYS** – If the functionality is not available.
- **0** – On success.

7.7.7 Other reference material

- Introduction to pin muxing and GPIO control under Linux

7.8 Porting

These pages document how to port Zephyr to new hardware.

7.8.1 Architecture Porting Guide

An architecture port is needed to enable Zephyr to run on an ISA (instruction set architecture) or an ABI (Application Binary Interface) that is not currently supported.

The following are examples of ISAs and ABIs that Zephyr supports:

- x86_32 ISA with System V ABI

- ARMv7-M ISA with Thumb2 instruction set and ARM Embedded ABI (aeabi)
- ARCv2 ISA

For information on Kconfig configuration, see *Setting Kconfig configuration values*. Architectures use a Kconfig configuration scheme similar to boards.

An architecture port can be divided in several parts; most are required and some are optional:

- **The early boot sequence**: each architecture has different steps it must take when the CPU comes out of reset (required).
- **Interrupt and exception handling**: each architecture handles asynchronous and unrequested events in a specific manner (required).
- **Thread context switching**: the Zephyr context switch is dependent on the ABI and each ISA has a different set of registers to save (required).
- **Thread creation and termination**: A thread's initial stack frame is ABI and architecture-dependent, and thread abortion possibly as well (required).
- **Device drivers**: most often, the system clock timer and the interrupt controller are tied to the architecture (some required, some optional).
- **Utility libraries**: some common kernel APIs rely on a architecture-specific implementation for performance reasons (required).
- **CPU idling/power management**: most architectures implement instructions for putting the CPU to sleep (partly optional, most likely very desired).
- **Fault management**: for implementing architecture-specific debug help and handling of fatal error in threads (partly optional).
- **Linker scripts and toolchains**: architecture-specific details will most likely be needed in the build system and when linking the image (required).
- **Memory Management and Memory Mapping**: for architecture-specific details on supporting memory management and memory mapping.
- **Stack Objects**: for architecture-specific details on memory protection hardware regarding stack objects.
- **User Mode Threads**: for supporting threads in user mode.
- **GDB Stub**: for supporting GDB stub to enable remote debugging.

Early Boot Sequence

The goal of the early boot sequence is to take the system from the state it is after reset to a state where is can run C code and thus the common kernel initialization sequence. Most of the time, very few steps are needed, while some architectures require a bit more work to be performed.

Common steps for all architectures:

- Setup an initial stack.
- If running an XIP (eXecute-In-Place) kernel, copy initialized data from ROM to RAM.
- If not using an ELF loader, zero the BSS section.
- Jump to `z_cstart()`, the early kernel initialization
  - `z_cstart()` is responsible for context switching out of the fake context running at startup into the main thread.

Some examples of architecture-specific steps that have to be taken:

- If given control in real mode on x86_32, switch to 32-bit protected mode.

- Setup the segment registers on x86_32 to handle boot loaders that leave them in an unknown or broken state.
- Initialize a board-specific watchdog on Cortex-M3/4.
- Switch stacks from MSP to PSP on Cortex-M.
- Use a different approach than calling into _Swap() on Cortex-M to prevent race conditions.
- Setup FIRQ and regular IRQ handling on ARCv2.

Interrupt and Exception Handling

Each architecture defines interrupt and exception handling differently.

When a device wants to signal the processor that there is some work to be done on its behalf, it raises an interrupt. When a thread does an operation that is not handled by the serial flow of the software itself, it raises an exception. Both, interrupts and exceptions, pass control to a handler. The handler is known as an ISR (Interrupt Service Routine) in the case of interrupts. The handler performs the work required by the exception or the interrupt. For interrupts, that work is device-specific. For exceptions, it depends on the exception, but most often the core kernel itself is responsible for providing the handler.

The kernel has to perform some work in addition to the work the handler itself performs. For example:

- Prior to handing control to the handler:
  - Save the currently executing context.
  - Possibly getting out of power saving mode, which includes waking up devices.
  - Updating the kernel uptime if getting out of tickless idle mode.
- After getting control back from the handler:
  - Decide whether to perform a context switch.
  - When performing a context switch, restore the context being context switched in.

This work is conceptually the same across architectures, but the details are completely different:

- The registers to save and restore.
- The processor instructions to perform the work.
- The numbering of the exceptions.
- etc.

It thus needs an architecture-specific implementation, called the interrupt/exception stub.

Another issue is that the kernel defines the signature of ISRs as:

```
void (*isr)(void *parameter)
```

Architectures do not have a consistent or native way of handling parameters to an ISR. As such there are two commonly used methods for handling the parameter.

- Using some architecture defined mechanism, the parameter value is forced in the stub. This is commonly found in X86-based architectures.
- The parameters to the ISR are inserted and tracked via a separate table requiring the architecture to discover at runtime which interrupt is executing. A common interrupt handler demuxer is installed for all entries of the real interrupt vector table, which then

fetches the device's ISR and parameter from the separate table. This approach is commonly used in the ARC and ARM architectures via the `CONFIG_GEN_ISR_TABLES` implementation. You can find examples of the stubs by looking at `_interrupt_enter()` in x86, `_IntExit()` in ARM, `_isr_wrapper()` in ARM, or the full implementation description for ARC in arch/arc/core/isr_wrapper.S.

Each architecture also has to implement primitives for interrupt control:

- locking interrupts: *irq_lock()*, *irq_unlock()*.
- registering interrupts: *IRQ_CONNECT()*.
- programming the priority if possible `irq_priority_set()`.
- enabling/disabling interrupts: *irq_enable()*, *irq_disable()*.

Note

IRQ_CONNECT is a macro that uses assembler and/or linker script tricks to connect interrupts at build time, saving boot time and text size.

The vector table should contain a handler for each interrupt and exception that can possibly occur. The handler can be as simple as a spinning loop. However, we strongly suggest that handlers at least print some debug information. The information helps figuring out what went wrong when hitting an exception that is a fault, like divide-by-zero or invalid memory access, or an interrupt that is not expected (*spurious interrupt*). See the ARM implementation in arch/arm/core/cortex_m/fault.c for an example.

Thread Context Switching

Multi-threading is the basic purpose to have a kernel at all. Zephyr supports two types of threads: preemptible and cooperative.

Two crucial concepts when writing an architecture port are the following:

- Cooperative threads run at a higher priority than preemptible ones, and always preempt them.
- After handling an interrupt, if a cooperative thread was interrupted, the kernel always goes back to running that thread, since it is not preemptible.

A context switch can happen in several circumstances:

- When a thread executes a blocking operation, such as taking a semaphore that is currently unavailable.
- When a preemptible thread unblocks a thread of higher priority by releasing the object on which it was blocked.
- When an interrupt unblocks a thread of higher priority than the one currently executing, if the currently executing thread is preemptible.
- When a thread runs to completion.
- When a thread causes a fatal exception and is removed from the running threads. For example, referencing invalid memory,

Therefore, the context switching must thus be able to handle all these cases.

The kernel keeps the next thread to run in a "cache", and thus the context switching code only has to fetch from that cache to select which thread to run.

There are two types of context switches: *cooperative* and *preemptive*.

- A *cooperative* context switch happens when a thread willfully gives the control to another thread. There are two cases where this happens
  - When a thread explicitly yields.
  - When a thread tries to take an object that is currently unavailable and is willing to wait until the object becomes available.
- A *preemptive* context switch happens either because an ISR or a thread causes an operation that schedules a thread of higher priority than the one currently running, if the currently running thread is preemptible. An example of such an operation is releasing an object on which the thread of higher priority was waiting.

Note

Control is never taken from cooperative thread when one of them is the running thread.

A cooperative context switch is always done by having a thread call the `_Swap()` kernel internal symbol. When `_Swap` is called, the kernel logic knows that a context switch has to happen: `_Swap` does not check to see if a context switch must happen. Rather, `_Swap` decides what thread to context switch in. `_Swap` is called by the kernel logic when an object being operated on is unavailable, and some thread yielding/sleeping primitives.

Note

On x86 and Nios2, `_Swap` is generic enough and the architecture flexible enough that `_Swap` can be called when exiting an interrupt to provoke the context switch. This should not be taken as a rule, since neither the ARM Cortex-M or ARCv2 port do this.

Since `_Swap` is cooperative, the caller-saved registers from the ABI are already on the stack. There is no need to save them in the k_thread structure.

A context switch can also be performed preemptively. This happens upon exiting an ISR, in the kernel interrupt exit stub:

- `_interrupt_enter` on x86 after the handler is called.
- `_IntExit` on ARM.
- `_firq_exit` and `_rirq_exit` on ARCv2.

In this case, the context switch must only be invoked when the interrupted thread was preemptible, not when it was a cooperative one, and only when the current interrupt is not nested.

The kernel also has the concept of "locking the scheduler". This is a concept similar to locking the interrupts, but lighter-weight since interrupts can still occur. If a thread has locked the scheduler, is it temporarily non-preemptible.

So, the decision logic to invoke the context switch when exiting an interrupt is simple:

- If the interrupted thread is not preemptible, do not invoke it.
- Else, fetch the cached thread from the ready queue, and:
  - If the cached thread is not the current thread, invoke the context switch.
  - Else, do not invoke it.

This is simple, but crucial: if this is not implemented correctly, the kernel will not function as intended and will experience bizarre crashes, mostly due to stack corruption.

Note

If running a coop-only system, i.e. if `CONFIG_NUM_PREEMPT_PRIORITIES` is 0, no preemptive context switch ever happens. The interrupt code can be optimized to not take any scheduling decision when this is the case.

Thread Creation and Termination

To start a new thread, a stack frame must be constructed so that the context switch can pop it the same way it would pop one from a thread that had been context switched out. This is to be implemented in an architecture-specific `_new_thread` internal routine.

The thread entry point is also not to be called directly, i.e. it should not be set as the PC (program counter) for the new thread. Rather it must be wrapped in `_thread_entry`. This means that the PC in the stack frame shall be set to `_thread_entry`, and the thread entry point shall be passed as the first parameter to `_thread_entry`. The specifics of this depend on the ABI.

The need for an architecture-specific thread termination implementation depends on the architecture. There is a generic implementation, but it might not work for a given architecture.

One reason that has been encountered for having an architecture-specific implementation of thread termination is that aborting a thread might be different if aborting because of a graceful exit or because of an exception. This is the case for ARM Cortex-M, where the CPU has to be taken out of handler mode if the thread triggered a fatal exception, but not if the thread gracefully exits its entry point function.

This means implementing an architecture-specific version of *k_thread_abort()*, and setting the Kconfig option `CONFIG_ARCH_HAS_THREAD_ABORT` as needed for the architecture (e.g. see arch/arm/core/cortex_m/Kconfig).

Thread Local Storage

To enable thread local storage on a new architecture:

1. Implement *arch_tls_stack_setup()* to setup the TLS storage area in stack. Refer to the toolchain documentation on how the storage area needs to be structured. Some helper functions can be used:
   - Function `z_tls_data_size()` returns the size needed for thread local variables (excluding any extra data required by toolchain and architecture).
   - Function `z_tls_copy()` prepares the TLS storage area for thread local variables. This only copies the variable themselves and does not do architecture and/or toolchain specific data.
2. In the context switching, grab the `tls` field inside the new thread's `struct k_thread` and put it into an appropriate register (or some other variable) for access to the TLS storage area. Refer to toolchain and architecture documentation on which registers to use.
3. In kconfig, add `select CONFIG_ARCH_HAS_THREAD_LOCAL_STORAGE` to kconfig related to the new architecture.
4. Run the `tests/kernel/threads/tls` to make sure the new code works.

Device Drivers

The kernel requires very few hardware devices to function. In theory, the only required device is the interrupt controller, since the kernel can run without a system clock. In practice, to get

access to most, if not all, of the sanity check test suite, a system clock is needed as well. Since these two are usually tied to the architecture, they are part of the architecture port.

Interrupt Controllers There can be significant differences between the interrupt controllers and the interrupt concepts across architectures.

For example, x86 has the concept of an IDT and different interrupt controllers. The position of an interrupt in the IDT determines its priority.

On the other hand, the ARM Cortex-M has the NVIC (Nested Vectored Interrupt Controller) as part of the architecture definition. There is no need for an IDT-like table that is separate from the NVIC vector table. The position in the table has nothing to do with priority of an IRQ: priorities are programmable per-entry.

The ARCv2 has its interrupt unit as part of the architecture definition, which is somewhat similar to the NVIC. However, where ARC defines interrupts as having a one-to-one mapping between exception and interrupt numbers (i.e. exception 1 is IRQ1, and device IRQs start at 16), ARM has IRQ0 being equivalent to exception 16 (and weirdly enough, exception 1 can be seen as IRQ-15).

All these differences mean that very little, if anything, can be shared between architectures with regards to interrupt controllers.

System Clock x86 has APIC timers and the HPET as part of its architecture definition. ARM Cortex-M has the SYSTICK exception. Finally, ARCv2 has the timer0/1 device.

Kernel timeouts are handled in the context of the system clock timer driver's interrupt handler.

Console Over Serial Line There is one other device that is almost a requirement for an architecture port, since it is so useful for debugging. It is a simple polling, output-only, serial port driver on which to send the console (`printk`, `printf`) output.

It is not required, and a RAM console (`CONFIG_RAM_CONSOLE`) can be used to send all output to a circular buffer that can be read by a debugger instead.

Utility Libraries

The kernel depends on a few functions that can be implemented with very few instructions or in a lock-less manner in modern processors. Those are thus expected to be implemented as part of an architecture port.

- Atomic operators.
  - If instructions do exist for a given architecture, the implementation is configured using the `CONFIG_ATOMIC_OPERATIONS_ARCH` Kconfig option.
  - If instructions do not exist for a given architecture, a generic version that wraps *`irq_lock()`* or *`irq_unlock()`* around non-atomic operations exists. It is configured using the `CONFIG_ATOMIC_OPERATIONS_C` Kconfig option.
- Find-least-significant-bit-set and find-most-significant-bit-set.
  - If instructions do not exist for a given architecture, it is always possible to implement these functions as generic C functions.

It is possible to use compiler built-ins to implement these, but be careful they use the required compiler barriers.

CPU Idling/Power Management

The kernel provides support for CPU power management with two functions: *arch_cpu_idle()* and *arch_cpu_atomic_idle()*.

arch_cpu_idle() can be as simple as calling the power saving instruction for the architecture with interrupts unlocked, for example `hlt` on x86, `wfi` or `wfe` on ARM, `sleep` on ARC. This function can be called in a loop within a context that does not care if it get interrupted or not by an interrupt before going to sleep. There are basically two scenarios when it is correct to use this function:

- In a single-threaded system, in the only thread when the thread is not used for doing real work after initialization, i.e. it is sitting in a loop doing nothing for the duration of the application.
- In the idle thread.

arch_cpu_atomic_idle(), on the other hand, must be able to atomically re-enable interrupts and invoke the power saving instruction. It can thus be used in real application code, again in single-threaded systems.

Normally, idling the CPU should be left to the idle thread, but in some very special scenarios, these APIs can be used by applications.

Both functions must exist for a given architecture. However, the implementation can be simply the following steps, if desired:

1. unlock interrupts
2. NOP

However, a real implementation is strongly recommended.

Fault Management

In the event of an unhandled CPU exception, the architecture code must call into `z_fatal_error()`. This function dumps out architecture-agnostic information and makes a policy decision on what to do next by invoking `k_sys_fatal_error()`. This function can be overridden to implement application-specific policies that could include locking interrupts and spinning forever (the default implementation) or even powering off the system (if supported).

Toolchain and Linking

Toolchain support has to be added to the build system.

Some architecture-specific definitions are needed in include/zephyr/toolchain/gcc.h. See what exists in that file for currently supported architectures.

Each architecture also needs its own linker script, even if most sections can be derived from the linker scripts of other architectures. Some sections might be specific to the new architecture, for example the SCB section on ARM and the IDT section on x86.

Memory Management and Memory Mapping

If the target platform enables paging and requires drivers to memory-map their I/O regions, `CONFIG_MMU` needs to be enabled and the following API implemented:

- *arch_mem_map()*
- *arch_mem_unmap()*
- *arch_page_phys_get()*

Stack Objects

The presence of memory protection hardware affects how stack objects are created. All architecture ports must specify the required alignment of the stack pointer, which is some combination of CPU and ABI requirements. This is defined in architecture headers with ARCH_STACK_PTR_ALIGN and is typically something small like 4, 8, or 16 bytes.

Two types of thread stacks exist:

- "kernel" stacks defined with *K_KERNEL_STACK_DEFINE()* and related APIs, which can host kernel threads running in supervisor mode or used as the stack for interrupt/exception handling. These have significantly relaxed alignment requirements and use less reserved data. No memory is reserved for privilege elevation stacks.
- "thread" stacks which typically use more memory, but are capable of hosting thread running in user mode, as well as any use-cases for kernel stacks.

If CONFIG_USERSPACE is not enabled, "thread" and "kernel" stacks are equivalent.

Additional macros may be defined in the architecture layer to specify the alignment of the base of stack objects, any reserved data inside the stack object not used for the thread's stack buffer, and how to round up stack sizes to support user mode threads. In the absence of definitions some defaults are assumed:

- ARCH_KERNEL_STACK_RESERVED: default no reserved space
- ARCH_THREAD_STACK_RESERVED: default no reserved space
- ARCH_KERNEL_STACK_OBJ_ALIGN: default align to ARCH_STACK_PTR_ALIGN
- ARCH_THREAD_STACK_OBJ_ALIGN: default align to ARCH_STACK_PTR_ALIGN
- ARCH_THREAD_STACK_SIZE_ALIGN: default round up to ARCH_STACK_PTR_ALIGN

All stack creation macros are defined in terms of these.

Stack objects all have the following layout, with some regions potentially zero-sized depending on configuration. There are always two main parts: reserved memory at the beginning, and then the stack buffer itself. The bounds of some areas can only be determined at runtime in the context of its associated thread object. Other areas are entirely computable at build time.

Some architectures may need to carve-out reserved memory at runtime from the stack buffer, instead of unconditionally reserving it at build time, or to supplement an existing reserved area (as is the case with the ARM FPU). Such carve-outs will always be tracked in `thread.stack_info.start`. The region specified by `thread.stack_info.start` and `thread.stack_info.size` is always fully accessible by a user mode thread. `thread.stack_info.delta` denotes an offset which can be used to compute the initial stack pointer from the very end of the stack object, taking into account storage for TLS and ASLR random offsets.

```
+---------------------+ <- thread.stack_obj
| Reserved Memory     | } K_(THREAD|KERNEL)_STACK_RESERVED
+---------------------+
| Carved-out memory   |
|.....................| <- thread.stack_info.start
| Unused stack buffer |
|                     |
|.....................| <- thread's current stack pointer
| Used stack buffer   |
|                     |
|.....................| <- Initial stack pointer. Computable
| ASLR Random offset  |      with thread.stack_info.delta
+---------------------| <- thread.userspace_local_data
| Thread-local data   |
+---------------------+ <- thread.stack_info.start + thread.stack_info.size
```

At present, Zephyr does not support stacks that grow upward.

No Memory Protection If no memory protection is in use, then the defaults are sufficient.

HW-based stack overflow detection This option uses hardware features to generate a fatal error if a thread in supervisor mode overflows its stack. This is useful for debugging, although for a couple reasons, you can't reliably make any assertions about the state of the system after this happens:

- The kernel could have been inside a critical section when the overflow occurs, leaving important global data structures in a corrupted state.
- For systems that implement stack protection using a guard memory region, it's possible to overshoot the guard and corrupt adjacent data structures before the hardware detects this situation.

To enable the CONFIG_HW_STACK_PROTECTION feature, the system must provide some kind of hardware-based stack overflow protection, and enable the CONFIG_ARCH_HAS_STACK_PROTECTION option.

Two forms of HW-based stack overflow detection are supported: dedicated CPU features for this purpose, or special read-only guard regions immediately preceding stack buffers.

CONFIG_HW_STACK_PROTECTION only catches stack overflows for supervisor threads. This is not required to catch stack overflow from user threads; CONFIG_USERSPACE is orthogonal.

This feature only detects supervisor mode stack overflows, including stack overflows when handling system calls. It doesn't guarantee that the kernel has not been corrupted. Any stack overflow in supervisor mode should be treated as a fatal error, with no assertions about the integrity of the overall system possible.

Stack overflows in user mode are recoverable (from the kernel's perspective) and require no special configuration; CONFIG_HW_STACK_PROTECTION only applies to catching overflows when the CPU is in supervisor mode.

CPU-based stack overflow detection If we are detecting stack overflows in supervisor mode via special CPU registers (like ARM's SPLIM), then the defaults are sufficient.

Guard-based stack overflow detection We are detecting supervisor mode stack overflows via special memory protection region located immediately before the stack buffer that generates an exception on write. Reserved memory will be used for the guard region.

ARCH_KERNEL_STACK_RESERVED should be defined to the minimum size of a memory protection region. On most ARM CPUs this is 32 bytes. ARCH_KERNEL_STACK_OBJ_ALIGN should also be set to the required alignment for this region.

MMU-based systems should not reserve RAM for the guard region and instead simply leave an non-present virtual page below every stack when it is mapped into the address space. The stack object will still need to be properly aligned and sized to page granularity.

```
+-----------------------------+ <- thread.stack_obj
| Guard reserved memory       | } K_KERNEL_STACK_RESERVED
+-----------------------------+
| Guard carve-out             |
|.............................| <- thread.stack_info.start
| Stack buffer                |
.                             .
```

Guard carve-outs for kernel stacks are uncommon and should be avoided if possible. They tend to be needed for two situations:

- The same stack may be re-purposed to host a user thread, in which case the guard is unnecessary and shouldn't be unconditionally reserved. This is the case when privilege elevation stacks are not inside the stack object.
- The required guard size is variable and depends on context. For example, some ARM CPUs have lazy floating point stacking during exceptions and may decrement the stack pointer by a large value without writing anything, completely overshooting a minimally-sized guard and corrupting adjacent memory. Rather than unconditionally reserving a larger guard, the extra memory is carved out if the thread uses floating point.

User mode enabled Enabling user mode activates two new requirements:

- A separate fixed-sized privilege mode stack, specified by `CONFIG_PRIVILEGED_STACK_SIZE`, must be allocated that the user thread cannot access. It is used as the stack by the kernel when handling system calls. If stack guards are implemented, a stack guard region must be able to be placed before it, with support for carve-outs if necessary.
- The memory protection hardware must be able to program a region that exactly covers the thread's stack buffer, tracked in `thread.stack_info`. This implies that `ARCH_THREAD_STACK_SIZE_ADJUST()` will need to round up the requested stack size so that a region may cover it, and that `ARCH_THREAD_STACK_OBJ_ALIGN()` is also specified per the granularity of the memory protection hardware.

This becomes more complicated if the memory protection hardware requires that all memory regions be sized to a power of two, and aligned to their own size. This is common on older MPUs and is known with `CONFIG_MPU_REQUIRES_POWER_OF_TWO_ALIGNMENT`.

`thread.stack_info` always tracks the user-accessible part of the stack object, it must always be correct to program a memory protection region with user access using the range stored within.

Non power-of-two memory region requirements On systems without power-of-two region requirements, the reserved memory area for threads stacks defined by `K_THREAD_STACK_RESERVED` may be used to contain the privilege mode stack. The layout could be something like:

```
+-----------------------------+ <- thread.stack_obj
| Other platform data         |
+-----------------------------+
| Guard region (if enabled)   |
+-----------------------------+
| Guard carve-out (if needed) |
|.............................|
| Privilege elevation stack   |
+-----------------------------| <- thread.stack_obj +
| Stack buffer                |      K_THREAD_STACK_RESERVED =
.                             .      thread.stack_info.start
```

The guard region, and any carve-out (if needed) would be configured as a read-only region when the thread is created.

- If the thread is a supervisor thread, the privilege elevation region is just extra stack memory. An overflow will eventually crash into the guard region.
- If the thread is running in user mode, a memory protection region will be configured to allow user threads access to the stack buffer, but nothing before or after it. An overflow in user mode will crash into the privilege elevation stack, which the user thread has no access to. An overflow when handling a system call will crash into the guard region.

On an MMU system there should be no physical guards; the privilege mode stack will be mapped into kernel memory, and the stack buffer in the user part of memory, each with non-present virtual guard pages below them to catch runtime stack overflows.

Other platform data may be stored before the guard region, but this is highly discouraged if such data could be stored in `thread.arch` somewhere.

`ARCH_THREAD_STACK_RESERVED` will need to be defined to capture the size of the reserved region containing platform data, privilege elevation stacks, and guards. It must be appropriately sized such that an MPU region to grant user mode access to the stack buffer can be placed immediately after it.

Power-of-two memory region requirements Thread stack objects must be sized and aligned to the same power of two, without any reserved memory to allow efficient packing in memory. Thus, any guards in the thread stack must be completely carved out, and the privilege elevation stack must be allocated elsewhere.

`ARCH_THREAD_STACK_SIZE_ADJUST()` and `ARCH_THREAD_STACK_OBJ_ALIGN()` should both be defined to `Z_POW2_CEIL()`. `K_THREAD_STACK_RESERVED` must be 0.

For the privilege stacks, the `CONFIG_GEN_PRIV_STACKS` must be, enabled. For every thread stack found in the system, a corresponding fixed- size kernel stack used for handling system calls is generated. The address of the privilege stacks can be looked up quickly at runtime based on the thread stack address using `z_priv_stack_find()`. These stacks are laid out the same way as other kernel-only stacks.

```
+-----------------------------+ <- z_priv_stack_find(thread.stack_obj)
| Reserved memory             | } K_KERNEL_STACK_RESERVED
+-----------------------------+
| Guard carve-out (if needed) |
|.............................|
| Privilege elevation stack   |
|                             |
+-----------------------------+ <- z_priv_stack_find(thread.stack_obj) +
                                     K_KERNEL_STACK_RESERVED +
                                     CONFIG_PRIVILEGED_STACK_SIZE

+-----------------------------+ <- thread.stack_obj
| MPU guard carve-out         |
| (supervisor mode only)      |
|.............................| <- thread.stack_info.start
| Stack buffer                |
.                             .
```

The guard carve-out in the thread stack object is only used if the thread is running in supervisor mode. If the thread drops to user mode, there is no guard and the entire object is used as the stack buffer, with full access to the associated user mode thread and `thread.stack_info` updated appropriately.

User Mode Threads

To support user mode threads, several kernel-to-arch APIs need to be implemented, and the system must enable the `CONFIG_ARCH_HAS_USERSPACE` option. Please see the documentation for each of these functions for more details:

- *`arch_buffer_validate()`* to test whether the current thread has access permissions to a particular memory region
- *`arch_user_mode_enter()`* which will irreversibly drop a supervisor thread to user mode privileges. The stack must be wiped.
- *`arch_syscall_oops()`* which generates a kernel oops when system call parameters can't be validated, in such a way that the oops appears to be generated from where the system call was invoked in the user thread

- *arch_syscall_invoke0()* through *arch_syscall_invoke6()* invoke a system call with the appropriate number of arguments which must all be passed in during the privilege elevation via registers.
- *arch_is_user_context()* return nonzero if the CPU is currently running in user mode
- *arch_mem_domain_max_partitions_get()* which indicates the max number of regions for a memory domain. MMU systems have an unlimited amount, MPU systems have constraints on this.

Some architectures may need to update software memory management structures or modify hardware registers on another CPU when memory domain APIs are invoked. If so, `CONFIG_ARCH_MEM_DOMAIN_SYNCHRONOUS_API` must be selected by the architecture and some additional APIs must be implemented. This is common on MMU systems and uncommon on MPU systems:

- `arch_mem_domain_thread_add()`
- `arch_mem_domain_thread_remove()`
- `arch_mem_domain_partition_add()`
- `arch_mem_domain_partition_remove()`

Please see the doxygen documentation of these APIs for details.

In addition to implementing these APIs, there are some other tasks as well:

- `_new_thread()` needs to spawn threads with *K_USER* in user mode
- On context switch, the outgoing thread's stack memory should be marked inaccessible to user mode by making the appropriate configuration changes in the memory management hardware.. The incoming thread's stack memory should likewise be marked as accessible. This ensures that threads can't mess with other thread stacks.
- On context switch, the system needs to switch between memory domains for the incoming and outgoing threads.
- Thread stack areas must include a kernel stack region. This should be inaccessible to user threads at all times. This stack will be used when system calls are made. This should be fixed size for all threads, and must be large enough to handle any system call.
- A software interrupt or some kind of privilege elevation mechanism needs to be established. This is closely tied to how the _arch_syscall_invoke macros are implemented. On system call, the appropriate handler function needs to be looked up in _k_syscall_table. Bad system call IDs should jump to the `K_SYSCALL_BAD` handler. Upon completion of the system call, care must be taken not to leak any register state back to user mode.

GDB Stub

To enable GDB stub for remote debugging on a new architecture:

1. Create a new `gdbstub.h` header file under appropriate architecture include directory (`include/arch/<arch>/gdbstub.h`).
   - Create a new struct `struct gdb_ctx` as the GDB context.
     - Must define a member named `exception` of type `unsigned int` to store the GDB exception reason. This value needs to be set before entering `z_gdb_main_loop()`.
     - Architecture can define as many members as needed for GDB stub to function.
     - Pointer to this struct needs to be passed to `z_gdb_main_loop()`, where this pointer will be passed to other GDB stub functions.
2. Functions for entering and exiting GDB stub main loop.

- If the architecture relies on interrupts to service breakpoints, interrupt service routines (ISR) need to be implemented, which will serve as the entry point to GDB stub main loop.
- These functions need to save and restore context so code execution can continue as if no breakpoints have been encountered.
- These functions need to call `z_gdb_main_loop()` after saving execution context to go into the GDB stub main loop to receive commands from GDB.
- Before calling `z_gdb_main_loop()`, `gdb_ctx.exception` must be set to specify the exception reason.

3. Implement necessary functions to support GDB stub functionality:

   - *arch_gdb_init()*
     - This needs to initialize necessary bits to support GDB stub functionality, for example, setting up the GDB context and connecting debug interrupts.
     - This must stop code execution via architecture specific method (e.g. raising debug interrupts). This allows GDB to connect during boot.
   - *arch_gdb_continue()*
     - This function is called when GDB sends a `c` or `continue` command to continue code execution.
   - *arch_gdb_step()*
     - This function is called when GDB sends a `si` or `stepi` command to execute one machine instruction, before returning to GDB prompt.
   - Hardware register read/write functions:
     - Since the GDB stub is running on the target, manipulation of hardware registers need to cached to avoid affecting the execution of GDB stub. Think of it as context switching, where the execution context is changed to the GDB stub. So that the register values of the running thread before context switch need to be stored. Manipulation of register values must only be done to this cached copy. The updated values will then be written to hardware registers before switching back to the previous running thread.
     - *arch_gdb_reg_readall()*
       - This collects all hardware register values that would appear in a `g`/`G` packets which will be sent back to GDB. The format of the G-packet is architecture specific. Consult GDB on what is expected.
       - Note that, for most architectures, a valid G-packet must be returned and sent to GDB. If a packet without incorrect length is sent to GDB, GDB will abort the debugging session.
     - *arch_gdb_reg_writeall()*
       - This takes a G-packet sent by GDB and populates the hardware registers with values from the G-packet.
     - *arch_gdb_reg_readone()*
       - This reads the value of one hardware register and sends the result to GDB.
     - *arch_gdb_reg_writeone()*
       - This writes the value of one hardware register received from GDB.
   - Breakpoints:
     - *arch_gdb_add_breakpoint()* and *arch_gdb_remove_breakpoint()*

  - GDB may decide to use software breakpoints which modifies the memory at the breakpoint locations to replace the instruction with software breakpoint or trap instructions. GDB will then restore the memory content once execution reaches the breakpoints. GDB supports this by default and there is usually no need to handle software breakpoints in the architecture code (where breakpoint type is `0`).
  - Hardware breakpoints (type `1`) are required if the code is in ROM or flash that cannot be modified at runtime. Consult the architecture datasheet on how to enable hardware breakpoints.
  - If hardware breakpoints are not supported by the architecture, there is no need to implement these in architecture code. GDB will then rely on software breakpoints.

4. For architecture where certain memory regions are not accessible, an array named `gdb_mem_region_array` of type `gdb_mem_region` needs to be defined to specify regions that are accessible. For each array item:
   - `gdb_mem_region.start` specifies the start of a memory region.
   - `gdb_mem_region.end` specifies the end of a memory region.
   - `gdb_mem_region.attributes` specifies the permission of a memory region.
     - `GDB_MEM_REGION_RO`: region is read-only.
     - `GDB_MEM_REGION_RW`: region is read-write.
   - `gdb_mem_region.alignment` specifies read/write alignment of a memory region. Use `0` if there is no alignment requirement and read/write can be done byte-by-byte.

API Reference

Timing

group `arch-timing`

Functions

void **arch_busy_wait**(uint32_t usec_to_wait)

Architecture-specific implementation of busy-waiting.

Parameters

- **usec_to_wait** – Wait period, in microseconds

static inline uint32_t **arch_k_cycle_get_32**(void)

Obtain the current cycle count, in units specified by CONFIG_SYS_CLOCK_HW_CYCLES_PER_SEC.

While this is historically specified as part of the architecture API, in practice virtually all platforms forward it to the sys_clock_cycle_get_32() API provided by the timer driver.

See also

k_cycle_get_32()

Returns

The current cycle time. This should count up monotonically through the

full 32 bit space, wrapping at 0xffffffff. Hardware with fewer bits of precision in the timer is expected to synthesize a 32 bit count.

static inline uint64_t **arch_k_cycle_get_64**(void)

As for *arch_k_cycle_get_32()*, but with a 64 bit return value.

Not all timer hardware has a 64 bit timer, this needs to be implemented only if CONFIG_TIMER_HAS_64BIT_CYCLE_COUNTER is set.

See also

arch_k_cycle_get_32()

Returns

The current cycle time. This should count up monotonically through the full 64 bit space, wrapping at 2^64-1. Hardware with fewer bits of precision in the timer is generally not expected to implement this API.

Threads

group **arch-threads**

Functions

void **arch_new_thread**(struct *k_thread* *thread, k_thread_stack_t *stack, char *stack_ptr, k_thread_entry_t entry, void *p1, void *p2, void *p3)

Handle arch-specific logic for setting up new threads.

The stack and arch-specific thread state variables must be set up such that a later attempt to switch to this thread will succeed and we will enter z_thread_entry with the requested thread and arguments as its parameters.

At some point in this function's implementation, z_setup_new_thread() must be called with the true bounds of the available stack buffer within the thread's stack object.

The provided stack pointer is guaranteed to be properly aligned with respect to the CPU and ABI requirements. There may be space reserved between the stack pointer and the bounds of the stack buffer for initial stack pointer randomization and thread-local storage.

Fields in thread->base will be initialized when this is called.

Parameters

- **thread** – Pointer to uninitialized struct *k_thread*
- **stack** – Pointer to the stack object
- **stack_ptr** – Aligned initial stack pointer
- **entry** – Thread entry function
- **p1** – 1st entry point parameter
- **p2** – 2nd entry point parameter
- **p3** – 3rd entry point parameter

static inline void **arch_switch**(void *switch_to, void **switched_from)

Cooperative context switch primitive.

The action of *arch_switch()* should be to switch to a new context passed in the first argument, and save a pointer to the current context into the address passed in the second argument.

The actual type and interpretation of the switch handle is specified by the architecture. It is the same data structure stored in the "switch_handle" field of a newly-created thread in *arch_new_thread()*, and passed to the kernel as the "interrupted" argument to z_get_next_switch_handle().

Note that on SMP systems, the kernel uses the store through the second pointer as a synchronization point to detect when a thread context is completely saved (so another CPU can know when it is safe to switch). This store must be done AFTER all relevant state is saved, and must include whatever memory barriers or cache management code is required to be sure another CPU will see the result correctly.

The simplest implementation of *arch_switch()* is generally to push state onto the thread stack and use the resulting stack pointer as the switch handle. Some architectures may instead decide to use a pointer into the thread struct as the "switch handle" type. These can legally assume that the second argument to *arch_switch()* is the address of the switch_handle field of struct thread_base and can use an offset on this value to find other parts of the thread struct. For example a (C pseudocode) implementation of *arch_switch()* might look like:

void *arch_switch(void *switch_to, void **switched_from)* { struct *k_thread* *new = switch_to; struct *k_thread* *old = *CONTAINER_OF(switched_from, struct k_thread,switch_handle)*;

// save old context... *switched_from = old; // restore new context... }

Note that the kernel manages the switch_handle field for synchronization as described above. So it is not legal for architecture code to assume that it has any particular value at any other time. In particular it is not legal to read the field from the address passed in the second argument.

Parameters

- **switch_to** – Incoming thread's switch handle
- **switched_from** – Pointer to outgoing thread's switch handle storage location, which must be updated.

void **arch_switch_to_main_thread**(struct *k_thread* *main_thread, char *stack_ptr, k_thread_entry_t _main)

Custom logic for entering main thread context at early boot.

Used by architectures where the typical trick of setting up a dummy thread in early boot context to "switch out" of isn't workable.

Parameters

- **main_thread** – main thread object
- **stack_ptr** – Initial stack pointer
- **_main** – Entry point for application main function.

int **arch_float_disable**(struct *k_thread* *thread)

Disable floating point context preservation.

The function is used to disable the preservation of floating point context information for a particular thread.

> **Note**
>
> For ARM architecture, disabling floating point preservation may only be requested for the current thread and cannot be requested in ISRs.

Return values

- `0` – On success.
- `-EINVAL` – If the floating point disabling could not be performed.
- `-ENOTSUP` – If the operation is not supported

int `arch_float_enable`(struct *k_thread* *thread, unsigned int options)

Enable floating point context preservation.

The function is used to enable the preservation of floating point context information for a particular thread. This API depends on each architecture implementation. If the architecture does not support enabling, this API will always be failed.

The *options* parameter indicates which floating point register sets will be used by the specified thread. Currently it is used by x86 only.

Parameters

- `thread` – ID of thread.
- `options` – architecture dependent options

Return values

- `0` – On success.
- `-EINVAL` – If the floating point enabling could not be performed.
- `-ENOTSUP` – If the operation is not supported

group `arch-tls`

Functions

size_t `arch_tls_stack_setup`(struct *k_thread* *new_thread, char *stack_ptr)

Setup Architecture-specific TLS area in stack.

This sets up the stack area for thread local storage. The structure inside TLS area is architecture specific.

Parameters

- `new_thread` – New thread object
- `stack_ptr` – Stack pointer

Returns

Number of bytes taken by the TLS area

Power Management

group `arch-pm`

Functions

FUNC_NORETURN void **arch_system_halt**(unsigned int reason)

Halt the system, optionally propagating a reason code.

void **arch_cpu_idle**(void)

Power save idle routine.

This function will be called by the kernel idle loop or possibly within an implementation of z_pm_save_idle in the kernel when the '_pm_save_flag' variable is non-zero.

Architectures that do not implement power management instructions may immediately return, otherwise a power-saving instruction should be issued to wait for an interrupt.

See also

k_cpu_idle()

Note

The function is expected to return after the interrupt that has caused the CPU to exit power-saving mode has been serviced, although this is not a firm requirement.

void **arch_cpu_atomic_idle**(unsigned int key)

Atomically re-enable interrupts and enter low power mode.

The requirements for *arch_cpu_atomic_idle()* are as follows:

a. Enabling interrupts and entering a low-power mode needs to be atomic, i.e. there should be no period of time where interrupts are enabled before the processor enters a low-power mode. See the comments in *k_lifo_get()*, for example, of the race condition that occurs if this requirement is not met.

b. After waking up from the low-power mode, the interrupt lockout state must be restored as indicated in the 'key' input parameter.

See also

k_cpu_atomic_idle()

Parameters

- **key** – Lockout key returned by previous invocation of *arch_irq_lock()*

Symmetric Multi-Processing

group **arch-smp**

Typedefs

typedef void (*`arch_cpustart_t`)(void *data)

Per-cpu entry function.

Param data
context parameter, implementation specific

Functions

void `arch_cpu_start`(int cpu_num, k_thread_stack_t *stack, int sz, *arch_cpustart_t* fn, void *arg)

Start a numbered CPU on a MP-capable system.

This starts and initializes a specific CPU. The main thread on startup is running on CPU zero, other processors are numbered sequentially. On return from this function, the CPU is known to have begun operating and will enter the provided function. Its interrupts will be initialized but disabled such that *irq_unlock()* with the provided key will work to enable them.

Normally, in SMP mode this function will be called by the kernel initialization and should not be used as a user API. But it is defined here for special-purpose apps which want Zephyr running on one core and to use others for design-specific processing.

Parameters

- `cpu_num` – Integer number of the CPU
- `stack` – Stack memory for the CPU
- `sz` – Stack buffer size, in bytes
- `fn` – Function to begin running on the CPU.
- `arg` – Untyped argument to be passed to “fn”

bool `arch_cpu_active`(int cpu_num)

Return CPU power status.

Parameters

- `cpu_num` – Integer number of the CPU

static inline struct _cpu *`arch_curr_cpu`(void)

Return the CPU struct for the currently executing CPU.

static inline uint32_t `arch_proc_id`(void)

Processor hardware ID.

Most multiprocessor architectures have a low-level unique ID value associated with the current CPU that can be retrieved rapidly and efficiently in kernel context. Note that while the numbering of the CPUs is guaranteed to be unique, the values are platform-defined. In particular, they are not guaranteed to match Zephyr’s own sequential CPU IDs (even though on some platforms they do).

Note

There is an inherent race with this API: the system may preempt the current thread and migrate it to another CPU before the value is used. Safe usage requires knowing the migration is impossible (e.g. because the code is in interrupt context, holds a spinlock, or cannot migrate due to k_cpu_mask state).

Returns
Unique ID for currently-executing CPU

void **arch_sched_broadcast_ipi**(void)
Broadcast an interrupt to all CPUs.

This will invoke z_sched_ipi() on all other CPUs in the system.

void **arch_sched_directed_ipi**(uint32_t cpu_bitmap)
Direct IPIs to the specified CPUs.

This will invoke z_sched_ipi() on the CPUs identified by *cpu_bitmap*.

Parameters

- **cpu_bitmap** – A bitmap indicating which CPUs need the IPI

int **arch_smp_init**(void)

static inline unsigned int **arch_num_cpus**(void)
Returns the number of CPUs.

For most systems this will be the same as CONFIG_MP_MAX_NUM_CPUS, however some systems may determine this at runtime instead.

Returns
the number of CPUs

Interrupts

group **arch-irq**

Functions

static inline bool **arch_is_in_isr**(void)
Test if the current context is in interrupt context.

XXX: This is inconsistently handled among arches wrt exception context See: #17656

Returns
true if we are in interrupt context

static inline unsigned int **arch_irq_lock**(void)
Lock interrupts on the current CPU.

See also

irq_lock()

static inline void **arch_irq_unlock**(unsigned int key)
Unlock interrupts on the current CPU.

See also

irq_unlock()

static inline bool **arch_irq_unlocked**(unsigned int key)

Test if calling *arch_irq_unlock()* with this key would unlock irqs.

Parameters

- **key** – value returned by *arch_irq_lock()*

Returns

true if interrupts were unlocked prior to the *arch_irq_lock()* call that produced the key argument.

void **arch_irq_disable**(unsigned int irq)

Disable the specified interrupt line.

See also

irq_disable()

Note

: The behavior of interrupts that arrive after this call returns and before the corresponding call to arch_irq_enable() is undefined. The hardware is not required to latch and deliver such an interrupt, though on some architectures that may work. Other architectures will simply lose such an interrupt and never deliver it. Many drivers and subsystems are not tolerant of such dropped interrupts and it is the job of the application layer to ensure that behavior remains correct.

void **arch_irq_enable**(unsigned int irq)

Enable the specified interrupt line.

See also

irq_enable()

int **arch_irq_is_enabled**(unsigned int irq)

Test if an interrupt line is enabled.

See also

irq_is_enabled()

int **arch_irq_connect_dynamic**(unsigned int irq, unsigned int priority, void (*routine)(const void *parameter), const void *parameter, uint32_t flags)

Arch-specific hook to install a dynamic interrupt.

Parameters

- **irq** – IRQ line number
- **priority** – Interrupt priority

- `routine` – Interrupt service routine
- `parameter` – ISR parameter
- `flags` – Arch-specific IRQ configuration flag

Returns
The vector assigned to this interrupt

int `arch_irq_disconnect_dynamic`(unsigned int irq, unsigned int priority, void (*routine)(const void *parameter), const void *parameter, uint32_t flags)

Arch-specific hook to dynamically uninstall a shared interrupt.

If the interrupt is not being shared, then the associated _sw_isr_table entry will be replaced by (NULL, z_irq_spurious) (default entry).

Parameters

- `irq` – IRQ line number
- `priority` – Interrupt priority
- `routine` – Interrupt service routine
- `parameter` – ISR parameter
- `flags` – Arch-specific IRQ configuration flag

Returns
0 in case of success, negative value otherwise

unsigned int `arch_irq_allocate`(void)

Arch-specific hook for allocating IRQs.

Note: disable/enable IRQ relevantly inside the implementation of such function to avoid concurrency issues. Also, an allocated IRQ is assumed to be used thus a following

See also

arch_irq_is_used() should return true.

Returns
The newly allocated IRQ or UINT_MAX on error.

void `arch_irq_set_used`(unsigned int irq)

Arch-specific hook for declaring an IRQ being used.

Note: disable/enable IRQ relevantly inside the implementation of such function to avoid concurrency issues.

Parameters

- `irq` – the IRQ to declare being used

bool `arch_irq_is_used`(unsigned int irq)

Arch-specific hook for checking if an IRQ is being used already.

Parameters

- `irq` – the IRQ to check

Returns
true if being, false otherwise

Userspace

group `arch-userspace`

Functions

static inline uintptr_t **arch_syscall_invoke0**(uintptr_t call_id)

Invoke a system call with 0 arguments.

No general-purpose register state other than return value may be preserved when transitioning from supervisor mode back down to user mode for security reasons.

It is required that all arguments be stored in registers when elevating privileges from user to supervisor mode.

Processing of the syscall takes place on a separate kernel stack. Interrupts should be enabled when invoking the system call marshallers from the dispatch table. Thread preemption may occur when handling system calls.

Call IDs are untrusted and must be bounds-checked, as the value is used to index the system call dispatch table, containing function pointers to the specific system call code.

Parameters

- **call_id** – System call ID

Returns
Return value of the system call. Void system calls return 0 here.

static inline uintptr_t **arch_syscall_invoke1**(uintptr_t arg1, uintptr_t call_id)

Invoke a system call with 1 argument.

See also

arch_syscall_invoke0()

Parameters

- **arg1** – First argument to the system call.
- **call_id** – System call ID, will be bounds-checked and used to reference kernel-side dispatch table

Returns
Return value of the system call. Void system calls return 0 here.

static inline uintptr_t **arch_syscall_invoke2**(uintptr_t arg1, uintptr_t arg2, uintptr_t call_id)

Invoke a system call with 2 arguments.

See also

arch_syscall_invoke0()

Parameters

- **arg1** – First argument to the system call.

- `arg2` – Second argument to the system call.
- `call_id` – System call ID, will be bounds-checked and used to reference kernel-side dispatch table

Returns
Return value of the system call. Void system calls return 0 here.

static inline uintptr_t `arch_syscall_invoke3`(uintptr_t arg1, uintptr_t arg2, uintptr_t arg3, uintptr_t call_id)

Invoke a system call with 3 arguments.

See also

arch_syscall_invoke0()

Parameters

- `arg1` – First argument to the system call.
- `arg2` – Second argument to the system call.
- `arg3` – Third argument to the system call.
- `call_id` – System call ID, will be bounds-checked and used to reference kernel-side dispatch table

Returns
Return value of the system call. Void system calls return 0 here.

static inline uintptr_t `arch_syscall_invoke4`(uintptr_t arg1, uintptr_t arg2, uintptr_t arg3, uintptr_t arg4, uintptr_t call_id)

Invoke a system call with 4 arguments.

See also

arch_syscall_invoke0()

Parameters

- `arg1` – First argument to the system call.
- `arg2` – Second argument to the system call.
- `arg3` – Third argument to the system call.
- `arg4` – Fourth argument to the system call.
- `call_id` – System call ID, will be bounds-checked and used to reference kernel-side dispatch table

Returns
Return value of the system call. Void system calls return 0 here.

static inline uintptr_t `arch_syscall_invoke5`(uintptr_t arg1, uintptr_t arg2, uintptr_t arg3, uintptr_t arg4, uintptr_t arg5, uintptr_t call_id)

Invoke a system call with 5 arguments.

See also

arch_syscall_invoke0()

Parameters

- `arg1` – First argument to the system call.
- `arg2` – Second argument to the system call.
- `arg3` – Third argument to the system call.
- `arg4` – Fourth argument to the system call.
- `arg5` – Fifth argument to the system call.
- `call_id` – System call ID, will be bounds-checked and used to reference kernel-side dispatch table

Returns

Return value of the system call. Void system calls return 0 here.

static inline uintptr_t `arch_syscall_invoke6`(uintptr_t arg1, uintptr_t arg2, uintptr_t arg3, uintptr_t arg4, uintptr_t arg5, uintptr_t arg6, uintptr_t call_id)

Invoke a system call with 6 arguments.

See also

arch_syscall_invoke0()

Parameters

- `arg1` – First argument to the system call.
- `arg2` – Second argument to the system call.
- `arg3` – Third argument to the system call.
- `arg4` – Fourth argument to the system call.
- `arg5` – Fifth argument to the system call.
- `arg6` – Sixth argument to the system call.
- `call_id` – System call ID, will be bounds-checked and used to reference kernel-side dispatch table

Returns

Return value of the system call. Void system calls return 0 here.

static inline bool `arch_is_user_context`(void)

Indicate whether we are currently running in user mode.

Returns

True if the CPU is currently running with user permissions

int **arch_mem_domain_max_partitions_get**(void)

Get the maximum number of partitions for a memory domain.

Returns
Max number of partitions, or -1 if there is no limit

int **arch_buffer_validate**(const void *addr, size_t size, int write)

Check memory region permissions.

Given a memory region, return whether the current memory management hardware configuration would allow a user thread to read/write that region. Used by system calls to validate buffers coming in from userspace.

Notes: The function is guaranteed to never return validation success, if the entire buffer area is not user accessible.

The function is guaranteed to correctly validate the permissions of the supplied buffer, if the user access permissions of the entire buffer are enforced by a single, enabled memory management region.

In some architectures the validation will always return failure if the supplied memory buffer spans multiple enabled memory management regions (even if all such regions permit user access).

Warning

Buffer of size zero (0) has undefined behavior.

Parameters

- **addr** – start address of the buffer
- **size** – the size of the buffer
- **write** – If non-zero, additionally check if the area is writable. Otherwise, just check if the memory can be read.

Returns
nonzero if the permissions don't match.

size_t **arch_virt_region_align**(uintptr_t phys, size_t size)

Get the optimal virtual region alignment to optimize the MMU table layout.

Some MMU HW requires some region to be aligned to some of the intermediate block alignment in order to reduce table usage. This call returns the optimal virtual address alignment in order to permit such optimization in the following MMU mapping call.

Parameters

- **phys** – **[in]** Physical address of region to be mapped, aligned to CONFIG_MMU_PAGE_SIZE
- **size** – **[in]** Size of region to be mapped, aligned to CONFIG_MMU_PAGE_SIZE

Returns
Alignment to apply on the virtual address of this region

FUNC_NORETURN void **arch_user_mode_enter**(k_thread_entry_t user_entry, void *p1, void *p2, void *p3)

Perform a one-way transition from supervisor to user mode.

Implementations of this function must do the following:

- Reset the thread's stack pointer to a suitable initial value. We do not need any prior context since this is a one-way operation.
- Set up any kernel stack region for the CPU to use during privilege elevation
- Put the CPU in whatever its equivalent of user mode is
- Transfer execution to *arch_new_thread()* passing along all the supplied arguments, in user mode.

Parameters

- `user_entry` – Entry point to start executing as a user thread
- `p1` – 1st parameter to user thread
- `p2` – 2nd parameter to user thread
- `p3` – 3rd parameter to user thread

FUNC_NORETURN void **arch_syscall_oops**(void *ssf)

Induce a kernel oops that appears to come from a specific location.

Normally, k_oops() generates an exception that appears to come from the call site of the k_oops() itself.

However, when validating arguments to a system call, if there are problems we want the oops to appear to come from where the system call was invoked and not inside the validation function.

Parameters

- `ssf` – System call stack frame pointer. This gets passed as an argument to _k_syscall_handler_t functions and its contents are completely architecture specific.

size_t **arch_user_string_nlen**(const char *s, size_t maxsize, int *err)

Safely take the length of a potentially bad string.

This must not fault, instead the `err` parameter must have -1 written to it. This function otherwise should work exactly like libc strnlen(). On success `err` should be set to 0.

Parameters

- `s` – String to measure
- `maxsize` – Max length of the string
- `err` – Error value to write

Returns

Length of the string, not counting NULL byte, up to maxsize

static inline bool **arch_mem_coherent**(void *ptr)

Detect memory coherence type.

Required when ARCH_HAS_COHERENCE is true. This function returns true if the byte pointed to lies within an architecture-defined "coherence region" (typically implemented with uncached memory) and can safely be used in multiprocessor code without explicit flush or invalidate operations.

Note

The result is for only the single byte at the specified address, this API is not required to check region boundaries or to expect aligned pointers. The expectation is that the code above will have queried the appropriate address(es).

static inline void **arch_cohere_stacks**(struct *k_thread* *old_thread, void *old_switch_handle, struct *k_thread* *new_thread)

Ensure cache coherence prior to context switch.

Required when ARCH_HAS_COHERENCE is true. On cache-incoherent multiprocessor architectures, thread stacks are cached by default for performance reasons. They must therefore be flushed appropriately on context switch. The rules are:

a. The region containing live data in the old stack (generally the bytes between the current stack pointer and the top of the stack memory) must be flushed to underlying storage so a new CPU that runs the same thread sees the correct data. This must happen before the assignment of the switch_handle field in the thread struct which signals the completion of context switch.

b. Any data areas to be read from the new stack (generally the same as the live region when it was saved) should be invalidated (and NOT flushed!) in the data cache. This is because another CPU may have run or re-initialized the thread since this CPU suspended it, and any data present in cache will be stale.

Note

The kernel will call this function during interrupt exit when a new thread has been chosen to run, and also immediately before entering *arch_switch()* to effect a code-driven context switch. In the latter case, it is very likely that more data will be written to the old_thread stack region after this function returns but before the completion of the switch. Simply flushing naively here is not sufficient on many architectures and coordination with the *arch_switch()* implementation is likely required.

Parameters

- **old_thread** – The old thread to be flushed before being allowed to run on other CPUs.
- **old_switch_handle** – The switch handle to be stored into old_thread (it will not be valid until the cache is flushed so is not present yet). This will be NULL if inside z_swap() (because the *arch_switch()* has not saved it yet).
- **new_thread** – The new thread to be invalidated before it runs locally.

Memory Management

group arch-mmu

Defines

ARCH_DATA_PAGE_ACCESSED

Bit indicating the data page was accessed since the value was last cleared.

Used by marking eviction algorithms. Safe to set this if uncertain.

This bit is undefined if ARCH_DATA_PAGE_LOADED is not set.

ARCH_DATA_PAGE_DIRTY

Bit indicating the data page, if evicted, will need to be paged out.

Set if the data page was modified since it was last paged out, or if it has never been paged out before. Safe to set this if uncertain.

This bit is undefined if ARCH_DATA_PAGE_LOADED is not set.

ARCH_DATA_PAGE_LOADED

Bit indicating that the data page is loaded into a physical page frame.

If un-set, the data page is paged out or not mapped.

ARCH_DATA_PAGE_NOT_MAPPED

If ARCH_DATA_PAGE_LOADED is un-set, this will indicate that the page is not mapped at all.

This bit is undefined if ARCH_DATA_PAGE_LOADED is set.

Enums

enum arch_page_location

Status of a particular page location.

Values:

enumerator ARCH_PAGE_LOCATION_PAGED_OUT

The page has been evicted to the backing store.

enumerator ARCH_PAGE_LOCATION_PAGED_IN

The page is resident in memory.

enumerator ARCH_PAGE_LOCATION_BAD

The page is not mapped.

Functions

void arch_mem_map(void *virt, uintptr_t phys, size_t size, uint32_t flags)

Map physical memory into the virtual address space.

This is a low-level interface to mapping pages into the address space. Behavior when providing unaligned addresses/sizes is undefined, these are assumed to be aligned to CONFIG_MMU_PAGE_SIZE.

The core kernel handles all management of the virtual address space; by the time we invoke this function, we know exactly where this mapping will be established. If the page tables already had mappings installed for the virtual memory region, these will be overwritten.

If the target architecture supports multiple page sizes, currently only the smallest page size will be used.

The memory range itself is never accessed by this operation.

This API must be safe to call in ISRs or exception handlers. Calls to this API are assumed to be serialized, and indeed all usage will originate from kernel/mm.c which handles virtual memory management.

Architectures are expected to pre-allocate page tables for the entire address space, as defined by CONFIG_KERNEL_VM_BASE and CONFIG_KERNEL_VM_SIZE. This operation should never require any kind of allocation for paging structures.

Validation of arguments should be done via assertions.

This API is part of infrastructure still under development and may change.

Parameters

- `virt` – Page-aligned Destination virtual address to map
- `phys` – Page-aligned Source physical address to map
- `size` – Page-aligned size of the mapped memory region in bytes
- `flags` – Caching, access and control flags, see K_MAP_* macros

void `arch_mem_unmap`(void *addr, size_t size)

Remove mappings for a provided virtual address range.

This is a low-level interface for un-mapping pages from the address space. When this completes, the relevant page table entries will be updated as if no mapping was ever made for that memory range. No previous context needs to be preserved. This function must update mappings in all active page tables.

Behavior when providing unaligned addresses/sizes is undefined, these are assumed to be aligned to CONFIG_MMU_PAGE_SIZE.

Behavior when providing an address range that is not already mapped is undefined.

This function should never require memory allocations for paging structures, and it is not necessary to free any paging structures. Empty page tables due to all contained entries being un-mapped may remain in place.

Implementations must invalidate TLBs as necessary.

This API is part of infrastructure still under development and may change.

Parameters

- `addr` – Page-aligned base virtual address to un-map
- `size` – Page-aligned region size

int `arch_page_phys_get`(void *virt, uintptr_t *phys)

Get the mapped physical memory address from virtual address.

The function only needs to query the current set of page tables as the information it reports must be common to all of them if multiple page tables are in use. If multiple page tables are active it is unnecessary to iterate over all of them.

Unless otherwise specified, virtual pages have the same mappings across all page tables. Calling this function on data pages that are exceptions to this rule (such as the scratch page) is undefined behavior. Just check the currently installed page tables and return the information in that.

Parameters

- `virt` – Page-aligned virtual address
- `phys` – **[out]** Mapped physical address (can be NULL if only checking if virtual address is mapped)

Return values

- `0` – if mapping is found and valid

- `-EFAULT` – if virtual address is not mapped

void `arch_reserved_pages_update`(void)

Update page frame database with reserved pages.

Some page frames within system RAM may not be available for use. A good example of this is reserved regions in the first megabyte on PC-like systems.

Implementations of this function should mark all relevant entries in k_mem_page_frames with K_PAGE_FRAME_RESERVED. This function is called at early system initialization with mm_lock held.

void `arch_mem_page_out`(void *addr, uintptr_t location)

Update all page tables for a paged-out data page.

This function:

- Sets the data page virtual address to trigger a fault if accessed that can be distinguished from access violations or un-mapped pages.
- Saves the provided location value so that it can retrieved for that data page in the page fault handler.
- The location value semantics are undefined here but the value will be always be page-aligned. It could be 0.

If multiple page tables are in use, this must update all page tables. This function is called with interrupts locked.

Calling this function on data pages which are already paged out is undefined behavior.

This API is part of infrastructure still under development and may change.

void `arch_mem_page_in`(void *addr, uintptr_t phys)

Update all page tables for a paged-in data page.

This function:

- Maps the specified virtual data page address to the provided physical page frame address, such that future memory accesses will function as expected. Access and caching attributes are undisturbed.
- Clears any accounting for “accessed” and “dirty” states.

If multiple page tables are in use, this must update all page tables. This function is called with interrupts locked.

Calling this function on data pages which are already paged in is undefined behavior.

This API is part of infrastructure still under development and may change.

void `arch_mem_scratch`(uintptr_t phys)

Update current page tables for a temporary mapping.

Map a physical page frame address to a special virtual address K_MEM_SCRATCH_PAGE, with read/write access to supervisor mode, such that when this function returns, the calling context can read/write the page frame’s contents from the K_MEM_SCRATCH_PAGE address.

This mapping only needs to be done on the current set of page tables, as it is only used for a short period of time exclusively by the caller. This function is called with interrupts locked.

This API is part of infrastructure still under development and may change.

enum *arch_page_location* `arch_page_location_get`(void *addr, uintptr_t *location)

Fetch location information about a page at a particular address.

The function only needs to query the current set of page tables as the information it reports must be common to all of them if multiple page tables are in use. If multiple page tables are active it is unnecessary to iterate over all of them. This may allow certain types of optimizations (such as reverse page table mapping on x86).

This function is called with interrupts locked, so that the reported information can't become stale while decisions are being made based on it.

Unless otherwise specified, virtual data pages have the same mappings across all page tables. Calling this function on data pages that are exceptions to this rule (such as the scratch page) is undefined behavior. Just check the currently installed page tables and return the information in that.

Parameters

- `addr` – Virtual data page address that took the page fault
- `location` – **[out]** In the case of ARCH_PAGE_LOCATION_PAGED_OUT, the backing store location value used to retrieve the data page. In the case of ARCH_PAGE_LOCATION_PAGED_IN, the physical address the page is mapped to.

Return values

- `ARCH_PAGE_LOCATION_PAGED_OUT` – The page was evicted to the backing store.
- `ARCH_PAGE_LOCATION_PAGED_IN` – The data page is resident in memory.
- `ARCH_PAGE_LOCATION_BAD` – The page is un-mapped or otherwise has had invalid access

uintptr_t `arch_page_info_get`(void *addr, uintptr_t *location, bool clear_accessed)

Retrieve page characteristics from the page table(s)

The architecture is responsible for maintaining "accessed" and "dirty" states of data pages to support marking eviction algorithms. This can either be directly supported by hardware or emulated by modifying protection policy to generate faults on reads or writes. In all cases the architecture must maintain this information in some way.

For the provided virtual address, report the logical OR of the accessed and dirty states for the relevant entries in all active page tables in the system if the page is mapped and not paged out.

If clear_accessed is true, the ARCH_DATA_PAGE_ACCESSED flag will be reset. This function will report its prior state. If multiple page tables are in use, this function clears accessed state in all of them.

This function is called with interrupts locked, so that the reported information can't become stale while decisions are being made based on it.

The return value may have other bits set which the caller must ignore.

Clearing accessed state for data pages that are not ARCH_DATA_PAGE_LOADED is undefined behavior.

ARCH_DATA_PAGE_DIRTY and ARCH_DATA_PAGE_ACCESSED bits in the return value are only significant if ARCH_DATA_PAGE_LOADED is set, otherwise ignore them.

ARCH_DATA_PAGE_NOT_MAPPED bit in the return value is only significant if ARCH_DATA_PAGE_LOADED is un-set, otherwise ignore it.

Unless otherwise specified, virtual data pages have the same mappings across all page tables. Calling this function on data pages that are exceptions to this rule (such as the scratch page) is undefined behavior.

This API is part of infrastructure still under development and may change.

Parameters

- `addr` – Virtual address to look up in page tables
- `location` – **[out]** If non-NULL, updated with either physical page frame address or backing store location depending on ARCH_DATA_PAGE_LOADED state. This is not touched if ARCH_DATA_PAGE_NOT_MAPPED.
- `clear_accessed` – Whether to clear ARCH_DATA_PAGE_ACCESSED state

Return values

`Value` – with ARCH_DATA_PAGE_* bits set reflecting the data page configuration

Miscellaneous Architecture APIs

group `arch-misc`

Functions

int `arch_printk_char_out`(int c)

Early boot console output hook.

Definition of this function is optional. If implemented, any invocation of printk() (or logging calls with CONFIG_LOG_MODE_MINIMAL which are backed by printk) will default to sending characters to this function. It is useful for early boot debugging before main serial or console drivers come up.

This can be overridden at runtime with __printk_hook_install().

The default __weak implementation of this does nothing.

Parameters

- `c` – Character to print

Returns

The character printed

static inline void `arch_kernel_init`(void)

Architecture-specific kernel initialization hook.

This function is invoked near the top of z_cstart, for additional architecture-specific setup before the rest of the kernel is brought up.

static inline void `arch_nop`(void)

Do nothing and return.

Yawn.

GDB Stub APIs

group `arch-gdbstub`

Functions

void **arch_gdb_init**(void)

Architecture layer debug start.

This function is called by `gdb_init()`

void **arch_gdb_continue**(void)

Continue running program.

Continue software execution.

void **arch_gdb_step**(void)

Continue with one step.

Continue software execution until reaches the next statement.

size_t **arch_gdb_reg_readall**(struct gdb_ctx *ctx, uint8_t *buf, size_t buflen)

Read all registers, and outputs as hexadecimal string.

This reads all CPU registers and outputs as hexadecimal string. The output string must be parsable by GDB.

Parameters

- **ctx** – GDB context
- **buf** – Buffer to output hexadecimal string.
- **buflen** – Length of buffer.

Returns

Length of hexadecimal string written. Return 0 if error or not supported.

size_t **arch_gdb_reg_writeall**(struct gdb_ctx *ctx, uint8_t *hex, size_t hexlen)

Take a hexadecimal string and update all registers.

This takes in a hexadecimal string as presented from GDB, and updates all CPU registers with new values.

Parameters

- **ctx** – GDB context
- **hex** – Input hexadecimal string.
- **hexlen** – Length of hexadecimal string.

Returns

Length of hexadecimal string parsed. Return 0 if error or not supported.

size_t **arch_gdb_reg_readone**(struct gdb_ctx *ctx, uint8_t *buf, size_t buflen, uint32_t regno)

Read one register, and outputs as hexadecimal string.

This reads one CPU register and outputs as hexadecimal string. The output string must be parsable by GDB.

Parameters

- **ctx** – GDB context
- **buf** – Buffer to output hexadecimal string.
- **buflen** – Length of buffer.
- **regno** – Register number

Returns

Length of hexadecimal string written. Return 0 if error or not supported.

size_t `arch_gdb_reg_writeone`(struct gdb_ctx *ctx, uint8_t *hex, size_t hexlen, uint32_t regno)

Take a hexadecimal string and update one register.

This takes in a hexadecimal string as presented from GDB, and updates one CPU registers with new value.

Parameters

- `ctx` – GDB context
- `hex` – Input hexadecimal string.
- `hexlen` – Length of hexadecimal string.
- `regno` – Register number

Returns

Length of hexadecimal string parsed. Return 0 if error or not supported.

int `arch_gdb_add_breakpoint`(struct gdb_ctx *ctx, uint8_t type, uintptr_t addr, uint32_t kind)

Add breakpoint or watchpoint.

Parameters

- `ctx` – GDB context
- `type` – Breakpoint or watchpoint type
- `addr` – Address of breakpoint or watchpoint
- `kind` – Size of breakpoint/watchpoint in bytes

Return values

- `0` – Operation successful
- `-1` – Error encountered
- `-2` – Not supported

int `arch_gdb_remove_breakpoint`(struct gdb_ctx *ctx, uint8_t type, uintptr_t addr, uint32_t kind)

Remove breakpoint or watchpoint.

Parameters

- `ctx` – GDB context
- `type` – Breakpoint or watchpoint type
- `addr` – Address of breakpoint or watchpoint
- `kind` – Size of breakpoint/watchpoint in bytes

Return values

- `0` – Operation successful
- `-1` – Error encountered
- `-2` – Not supported

7.8.2 SoC Porting Guide

This page describes how to add support for a new *SoC* in Zephyr, be it in the upstream Zephyr project or locally in your own repository.

SoC Definitions

It is expected that you are familiar with the board concept in Zephyr. A high level overview of the hardware support hierarchy and terms used in the Zephyr documentation can be seen in *Hardware support hierarchy*.

For SoC porting, the most important terms are:

- SoC: the exact system on a chip the board's CPU is part of.
- SoC series: a group of tightly related SoCs.
- SoC family: a wider group of SoCs with similar characteristics.
- CPU Cluster: a cluster of one or more CPU cores.
- CPU core: a particular CPU instance of a given architecture.
- Architecture: an instruction set architecture.

Architecture See *Architecture Porting Guide*.

Create your SoC directory

Each SoC must have a unique name. Use the official name given by the SoC vendor and check that it's not already in use. In some cases someone else may have contributed a SoC with identical name. If the SoC name is already in use, then you should probably improve the existing SoC instead of creating a new one. The script `list_hardware` can be used to retrieve a list of all SoCs known in Zephyr, for example `./scripts/list_hardware.py --soc-root=. --socs` from the Zephyr base directory for a list of names that are already in use.

Start by creating the directory `zephyr/soc/<VENDOR>/soc1`, where `<VENDOR>` is your vendor sub-directory.

Note

A `<VENDOR>` subdirectory is mandatory if contributing your SoC to Zephyr, but if your SoC is placed in a local repo, then any folder structure under `<your-repo>/soc` is permitted. The `<VENDOR>` subdirectory must match a vendor defined in the list in dts/bindings/vendor-prefixes.txt. If the SoC vendor does not have a prefix in that list, then one must be created.

Note

The SoC directory name does not need to match the name of the SoC. Multiple SoCs can even be defined in one directory. In Zephyr, SoCs are often organized in sub-folders in a common SoC Family or SoC Series tree.

Your SoC directory should look like this:

```
soc/<VENDOR>/<soc-name>
├── soc.yml
├── soc.h
├── CMakeLists.txt
├── Kconfig
├── Kconfig.soc
└── Kconfig.defconfig
```

Replace <soc-name> with your SoC's name.

The mandatory files are:

1. soc.yml: a YAML file describing the high-level meta data of the SoC such as: - SoC name: the name of the SoC - CPU clusters: CPU clusters if the SoC contains one or more clusters - SoC series: the SoC series to which the SoC belong - SoC family: the SoC family to which the series belong
2. soc.h: a header file which can be used to describe or provide configuration macros for the SoC. The soc.h will often be included in drivers, sub-systems, boards, and other source code found in Zephyr.
3. Kconfig.soc: the base SoC configuration which defines a Kconfig SoC symbol in the form of config SOC_<soc-name> and provides the SoC name to the Kconfig SOC setting. If the soc.yml describes a SoC family and series, then those must also be defined in this file. Kconfig settings outside of the SoC tree must not be selected. To select general Zephyr Kconfig settings the Kconfig file must be used.
4. CMakeLists.txt: CMake file loaded by the Zephyr build system. This CMake file can define additional include paths and/or source files to be used when a build targets the SoC. Also the base line linker script to use must be defined.

The optional files are:

- Kconfig, Kconfig.defconfig software configuration in *Configuration System (Kconfig)* format. These select the architecture and peripherals available.

Write your SoC YAML

The SoC YAML file describes the SoC family, SoC series, and SoC at a high level.

Detailed configurations, such as hardware description and configuration are done in devicetree and Kconfig.

The skeleton of a simple SoC YAML file containing just one SoC is:

```
socs:
- name: <soc1>
```

It is possible to have multiple SoC located in the SoC folder. For example if they belong to a common family or series it is recommended to locate such SoC in a common tree. Multiple SoCs and SoC series in a common folder can be described in the soc.yml file as:

```
family:
  name: <family-name>
  series:
    - name: <series-1-name>
      socs:
        - name: <soc1>
          cpucluster:
            - name: <coreA>
            - name: <coreB>
              ...
        - name: <soc2>
    - name: <series-2-name>
      ...
```

Write your SoC devicetree

SoC devicetree include files are located in the <zephyr-repo>/dts folder under a corresponding <ARCH>/<VENDOR>.

The SoC dts/<ARCH>/<VENDOR>/<soc>.dtsi describes your SoC hardware in the Devicetree Source (DTS) format and must be included by any boards which use the SoC.

If a highlevel <arch>.dtsi file exists, then a good starting point is to include this file in your <soc>.dtsi.

In general, <soc>.dtsi should look like this:

```
#include <arch>/<arch>.dtsi

/ {
        chosen {
                /* common chosen settings for your SoC */
        };

        cpus {
                #address-cells = <m>;
                #size-cells = <n>;

                cpu@0 {
                device_type = "cpu";
                compatible = "<compatibles>";
                /* ... your CPU definitions ... */
        };

        soc {
                /* Your SoC definitions and peripherals */
                /* such as ram, clock, buses, peripherals. */
        };
};
```

Hint

It is possible to structure multiple <VENDOR>/<soc>.dtsi files in sub-directories for a cleaner file system structure. For example organized pre SoC series, like this: <VENDOR>/<SERIES>/<soc>.dtsi.

Multiple CPU clusters Devicetree reflects the hardware. The memory space and peripherals available to one CPU cluster can be very different from another CPU cluster, therefore each CPU cluster will often have its own .dtsi file.

CPU cluster .dtsi files should follow the naming scheme <soc>_<cluster>.dtsi. A <soc>_<cluster>.dtsi file will look similar to a SoC .dtsi without CPU clusters.

Write Kconfig files

Zephyr uses the Kconfig language to configure software features. Your SoC needs to provide some Kconfig settings before you can compile a Zephyr application for it.

Setting Kconfig configuration values is documented in detail in *Setting Kconfig configuration values*.

There is one mandatory Kconfig file in the SoC directory, and two optional files for a SoC:

```
soc/<vendor>/<your soc>
├── Kconfig.soc
├── Kconfig
└── Kconfig.defconfig
```

`Kconfig.soc`
: A shared Kconfig file which can be sourced both in Zephyr Kconfig and sysbuild Kconfig trees.

  This file selects the SoC family and series in the Kconfig tree and potential other SoC related Kconfig settings. In some cases a SOC_PART_NUMBER. This file must not select anything outside the re-usable Kconfig SoC tree.

  A `Kconfig.soc` may look like this:

```
config SOC_<series name>
        bool

config SOC_<SOC_NAME>
        bool
        select SOC_SERIES_<series name>

config SOC
        default "SoC name" if SOC_<SOC_NAME>
```

  Notice that `SOC_NAME` is a pure upper case version of the SoC name.

  The Kconfig `SOC` setting is globally defined as a string and therefore the `Kconfig.soc` file shall only define the default string value and not the type. Notice that the string value must match the SoC name used in the `soc.yml` file.

`Kconfig`
: Included by soc/Kconfig.

  This file can add Kconfig settings which are specific to the current SoC.

  The `Kconfig` will often indicate given hardware support using a setting of the form `HAS_<support>`.

```
config SOC_<SOC_NAME>
        select ARM
        select CPU_HAS_FPU
```

  If the setting name is identical to an existing Kconfig setting in Zephyr and only modifies the default value of said setting, then `Kconfig.defconfig` should be used instead.

`Kconfig.defconfig`
: SoC specific default values for Kconfig options.

  Not all SoCs have a `Kconfig.defconfig` file.

  The entire file should be inside a pair of `if SOC_<SOC_NAME>` / `endif` or `if SOC_SERIES_<SERIES_NAME>` / `endif`, like this:

```
if SOC_<SOC_NAME>

config NUM_IRQS
        default 32

endif # SOC_<SOC_NAME>
```

Multiple CPU clusters CPU clusters must provide additional Kconfig settings in the `Kconfig.soc` file. This will usually be in the form of `SOC_<SOC_NAME>_<CLUSTER>` so for a given `soc1` with two clusters `clusterA` and `clusterB`, then this will look like:

SoC's When a SoC defines CPU cluster

```
config SOC_SOC1_CLUSTERA
        bool
        select SOC_SOC1

config SOC_SOC1_CLUSTERB
        bool
        select SOC_SOC1
```

7.8.3 Board Porting Guide

To add Zephyr support for a new *board*, you at least need a *board directory* with various files in it. Files in the board directory inherit support for at least one SoC and all of its features. Therefore, Zephyr must support your *SoC* as well.

Transition to the current hardware model

Shortly after Zephyr 3.6.0 was released, a new hardware model was introduced to Zephyr. This new model overhauls the way both SoCs and boards are named and defined, and adds support for features that had been identified as important over the years. Among them:

- Support for multi-core, multi-arch AMP (Asymmetrical Multi Processing) SoCs
- Support for multi-SoC boards
- Support for reusing the SoC and board Kconfig trees outside of the Zephyr build system
- Support for advanced use cases with *Sysbuild (System build)*
- Removal of all existing arbitrary and inconsistent uses of Kconfig and folder names

All the documentation in this page refers to the current hardware model. Please refer to the documentation in Zephyr v3.6.0 (or earlier) for information on the previous, now obsolete, hardware model.

More information about the rationale, development and concepts behind the new model can be found in the original issue, the original Pull Request and, for a complete set of changes introduced, the hardware model v2 commit.

Some non-critical features, enhancements and improvements of the new hardware model are still in development. Check the hardware model v2 enhancements issue for a complete list.

The transition from the previous hardware model to the current one (commonly referred to as “hardware model v2”) requires modifications to all existing board and SoC definitions. A decision was made not to provide direct backwards compatibility for the previous model, which leaves users transitioning from a previous version of Zephyr to one including the new model (v3.7.0 and onwards) with two options if they have an out-of-tree board (or SoC):

1. Convert the out-of-tree board to the current hardware model (recommended)
2. Take the SoC definition from Zephyr v3.6.0 and copy it to your downstream repository (ensuring that the build system can find it via a *zephyr module* or `SOC_ROOT`). This will allow your board, defined in the previous hardware model, to continue to work

When converting your board from the previous to the current hardware model, we recommend first reading through this page to understand the model in detail. You can then use the example-application conversion Pull Request as an example on how to port a simple board. Additionally, a conversion script is available and works reliably in many cases (though multi-core SoCs may not be handled entirely). Finally, the hardware model v2 commit contains the full conversion of all existing boards from the old to the current model, so you can use it as a complete conversion reference.

Hardware support hierarchy

Zephyr's hardware support is based on a series of hierarchical abstractions. Primarily, each *board* has one or more *SoC*. Each SoC can be optionally classed into an *SoC series*, which in turn may optionally belong to an *SoC family*. Each SoC has one or more *CPU cluster*, each containing one or more *CPU core* of a particular *architecture*.

You can visualize the hierarchy in the diagram below:

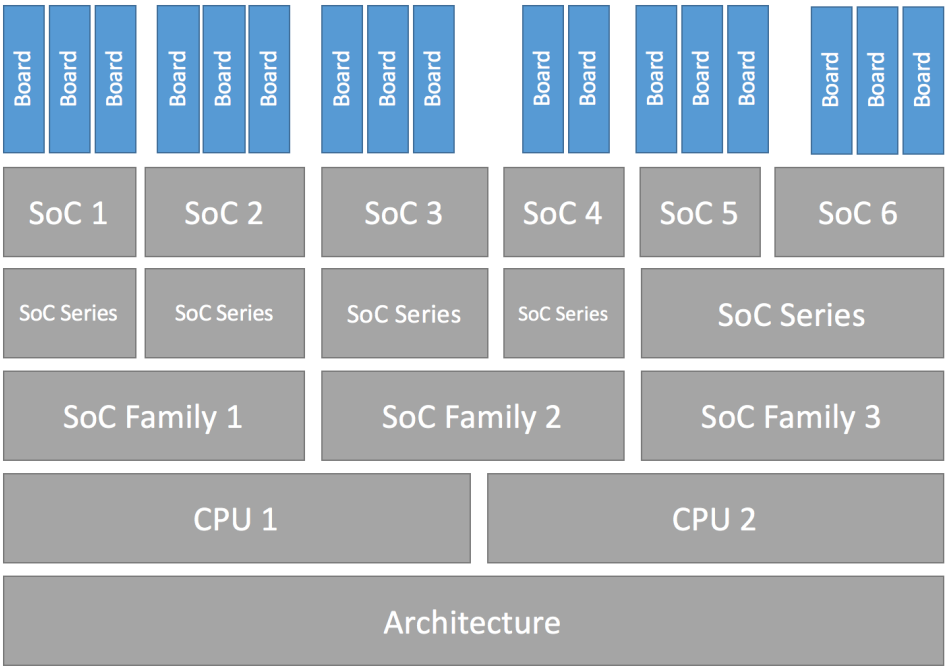


Fig. 5: Hardware support Hierarchy

Below are some examples of the hierarchy described in this section, in the form of a *board* per row with its corresponding hierarchy entries. Notice how the *SoC series* and *SoC family* levels are not always used.

board name	*board qualifiers*	*SoC*	*SoC Series*	*SoC family*	CPU core	*architecture*
nrf52dk	nrf52832	nRF52832	nRF52	Nordic nRF	Arm Cortex-M4	ARMv7-M
frdm_k64f	mk64f12	MK64F12	Kinetis K6x	NXP Kinetis	Arm Cortex-M4	ARMv7-M
rv32m1_vega	openisa_rv32m1/r	RV32M1	(Not used)	(Not used)	RI5CY	RISC-V RV32
nrf5340dk	nrf5340/cpuapp	nRF5340	nRF53	Nordic nRF	Arm Cortex-M33	ARMv8-M
	nrf5340/cpunet	nRF5340	nRF53	Nordic nRF	Arm Cortex-M33	ARMv8-M
mimx8mp_e	mimx8ml8/a53	i.MX8M Plus	i.MX8M	NXP i.MX	Arm Cortex-A53	ARMv8-A
	mimx8ml8/m7	i.MX8M Plus	i.MX8M	NXP i.MX	Arm Cortex-M7	ARMv7-M
	mimx8ml8/adsp	i.MX8M Plus	i.MX8M	NXP i.MX	Cadence HIFI4	Xtensa LX6

Additional details about terminology can be found in the next section.

Board terminology

The previous section introduced the hierarchical manner in which Zephyr classifies and implements hardware support. This section focuses on the terminology used around hardware support, and in particular when defining and working with boards and SoCs.

The overall set of terms used around the concept of board in Zephyr is depicted in the image below, which uses the bl5340_dvk board as reference.

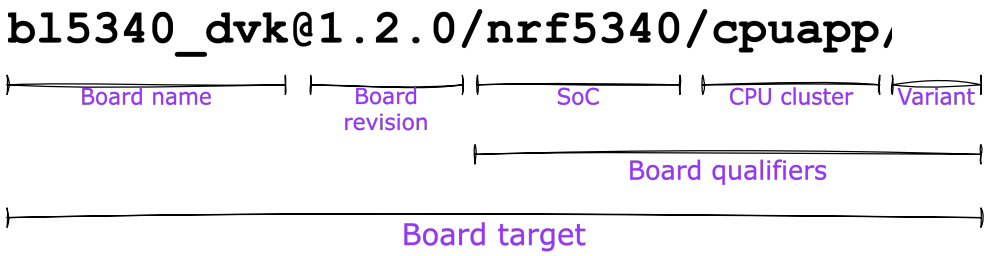


Fig. 6: Board terminology diagram

The diagram shows the different terms that are used to describe boards:

- The *board name*: `bl5340_dvk`
- The optional *board revision*: `1.2.0`
- The *board qualifiers*, that optionally describe the *SoC*, *CPU cluster* and *variant*: `nrf5340/cpuapp/ns`
- The *board target*, which uniquely identifies a combination of the above and can be used to specify the hardware to build for when using the tooling provided by Zephyr: `bl5340_dvk@1.2.0/nrf5340/cpuapp/ns`

Formally this can also be seen as *board name*`[@`*revision*`][/`*board qualifiers*`]`, which can be extended to *board name*`[@`*revision*`][/`*SoC*`[/`*CPU cluster*`][/`*variant*`]]`.

If a board contains only one single-core SoC, then the SoC can be omitted from the board target. This implies that if the board does not define any board qualifiers, the board name can be used as a board target. Conversely, if board qualifiers are part of the board definition, then the SoC can be omitted by leaving it out but including the corresponding forward-slashes: `//`.

Continuing with the example above, The board bl5340_dvk is a single SoC board where the SoC defines two CPU clusters: `cpuapp` and `cpunet`. One of the CPU clusters, `cpuapp`, additionally defines a non-secure board variant, `ns`.

The board qualifiers `nrf5340/cpuapp/ns` can be read as:

- `nrf5340`: The SoC, which is a Nordic nRF5340 dual-core SoC
- `cpuapp`: The CPU cluster `cpuapp`, which consists of a single Cortex-M33 CPU core. The number of cores in a CPU cluster cannot be determined from the board qualifiers.
- `ns`: a variant, in this case `ns` is a common variant name is Zephyr denoting a non-secure build for boards supporting *Trusted Firmware-M*.

Not all SoCs define CPU clusters or variants. For example a simple board like the thingy52_nrf52832 contains a single SoC with no CPU clusters and no variants. For `thingy52` the board target `thingy52/nrf52832` can be read as:

- `thingy52`: board name.

- `nrf52832`: The board qualifiers, in this case identical to the SoC, which is a Nordic nRF52832.

Make sure your SoC is supported

Start by making sure your SoC is supported by Zephyr. If it is, it's time to *Create your board directory*. If you don't know, try:

- checking boards for names that look relevant, and reading individual board documentation to find out for sure.
- asking your SoC vendor

If you need to add a SoC, CPU cluster, or even architecture support, this is the wrong page, but here is some general advice.

Architecture See *Architecture Porting Guide*.

CPU Core CPU core support files go in `core` subdirectories under arch, e.g. arch/x86/core.

See *Install a Toolchain* for information about toolchains (compiler, linker, etc.) supported by Zephyr. If you need to support a new toolchain, *Build and Configuration Systems* is a good place to start learning about the build system. Please reach out to the community if you are looking for advice or want to collaborate on toolchain support.

SoC Zephyr SoC support files are in architecture-specific subdirectories of soc. They are generally grouped by SoC family.

When adding a new SoC family or series for a vendor that already has SoC support within Zephyr, please try to extract common functionality into shared files to avoid duplication. If there is no support for your vendor yet, you can add it in a new directory `zephyr/soc/<VENDOR>/<YOUR-SOC>`; please use self-explanatory directory names.

Create your board directory

Once you've found an existing board that uses your SoC, you can usually start by copy/pasting its board directory and changing its contents for your hardware.

You need to give your board a unique name. Run `west boards` for a list of names that are already taken, and pick something new. Let's say your board is called `plank` (please don't actually use that name).

Start by creating the board directory `zephyr/boards/<VENDOR>/plank`, where `<VENDOR>` is your vendor subdirectory. (You don't have to put your board directory in the zephyr repository, but it's the easiest way to get started. See *Custom Board, Devicetree and SOC Definitions* for documentation on moving your board directory to a separate repository once it's working.)

Note

A `<VENDOR>` subdirectory is mandatory if contributing your board to Zephyr, but if your board is placed in a local repo, then any folder structure under `<your-repo>/boards` is permitted. If the vendor is defined in the list in dts/bindings/vendor-prefixes.txt then you must use that vendor prefix as `<VENDOR>`. `others` may be used as vendor prefix if the vendor is not defined.

Note

The board directory name does not need to match the name of the board. Multiple boards can even defined be in one directory.

Your board directory should look like this:

```
boards/<VENDOR>/plank
├── board.yml
├── board.cmake
├── CMakeLists.txt
├── doc
│   ├── plank.png
│   └── index.rst
├── Kconfig.plank
├── Kconfig.defconfig
├── plank_defconfig
├── plank_<qualifiers>_defconfig
├── plank.dts
├── plank_<qualifiers>.dts
└── plank.yaml
```

Replace `plank` with your board's name, of course.

The mandatory files are:

1. `board.yml`: a YAML file describing the high-level meta data of the boards such as the boards names, their SoCs, and variants. CPU clusters for multi-core SoCs are not described in this file as they are inherited from the SoC's YAML description.
2. `plank.dts` or `plank_<qualifiers>.dts`: a hardware description in *devicetree* format. This declares your SoC, connectors, and any other hardware components such as LEDs, buttons, sensors, or communication peripherals (USB, BLE controller, etc).
3. `Kconfig.plank`: the base software configuration for selecting SoC and other board and SoC related settings. Kconfig settings outside of the board and SoC tree must not be selected. To select general Zephyr Kconfig settings the `Kconfig` file must be used.

The optional files are:

- `Kconfig`, `Kconfig.defconfig` software configuration in *Configuration System (Kconfig)* formats. This provides default settings for software features and peripheral drivers.
- `plank_defconfig` and `plank_<qualifiers>_defconfig`: software configuration in Kconfig `.conf` format.
- `board.cmake`: used for *Flash and debug support*
- `CMakeLists.txt`: if you need to add additional source files to your build.
- `doc/index.rst`, `doc/plank.png`: documentation for and a picture of your board. You only need this if you're *Contributing your board* to Zephyr.
- `plank.yaml`: a YAML file with miscellaneous metadata used by the *Test Runner (Twister)*.

Board qualifiers of the form `<soc>/<cpucluster>/<variant>` are normalized so that / is replaced with _ when used for filenames, for example: `soc1/foo` becomes `soc1_foo` when used in filenames.

Write your board YAML

The board YAML file describes the board at a high level. This includes the SoC, board variants, and board revisions.

Detailed configurations, such as hardware description and configuration are done in devicetree and Kconfig.

The skeleton of the board YAML file is:

```
board:
  name: <board-name>
  vendor: <board-vendor>
  revision:
    format: <major.minor.patch|letter|number|custom>
    default: <default-revision-value>
    exact: <true|false>
    revisions:
    - name: <revA>
    - name: <revB>
      ...
  socs:
  - name: <soc-1>
    variants:
    - name: <variant-1>
    - name: <variant-2>
      variants:
      - name: <sub-variant-2-1>
        ...
  - name: <soc-2>
    ...
```

It is possible to have multiple boards located in the board folder. If multiple boards are placed in the same board folder, then the file `board.yml` must describe those in a list as:

```
boards:
- name: <board-name-1>
  vendor: <board-vendor>
  ...
- name: <board-name-2>
  vendor: <board-vendor>
  ...
...
```

Write your devicetree

The devicetree file `boards/<vendor>/plank/plank.dts` or `boards/<vendor>/plank/plank_<qualifiers>.dts` describes your board hardware in the Devicetree Source (DTS) format (as usual, change `plank` to your board's name). If you're new to devicetree, see *Introduction to devicetree*.

In general, `plank.dts` should look like this:

```
/dts-v1/;
#include <your_soc_vendor/your_soc.dtsi>

/ {
        model = "A human readable name";
        compatible = "yourcompany,plank";

        chosen {
                zephyr,console = &your_uart_console;
                zephyr,sram = &your_memory_node;
                /* other chosen settings  for your hardware */
        };
```

(continues on next page)

(continued from previous page)

```
        /*
         * Your board-specific hardware: buttons, LEDs, sensors, etc.
         */

        leds {
                compatible = "gpio-leds";
                led0: led_0 {
                        gpios = < /* GPIO your LED is hooked up to */ >;
                        label = "LED 0";
                };
                /* ... other LEDs ... */
        };

        buttons {
                compatible = "gpio-keys";
                /* ... your button definitions ... */
        };

        /* These aliases are provided for compatibility with samples */
        aliases {
                led0 = &led0; /* now you support the blinky sample! */
                /* other aliases go here */
        };
};

&some_peripheral_you_want_to_enable { /* like a GPIO or SPI controller */
        status = "okay";
};

&another_peripheral_you_want {
        status = "okay";
};
```

Only one `.dts` file will be used, and the most specific file which exists will be used.

This means that if both `plank.dts` and `plank_soc1_foo.dts` exist, then when building for `plank` / `plank/soc1`, then `plank.dts` is used. When building for `plank//foo` / `plank/soc1/foo` the `plank_soc1_foo.dts` is used.

This allows board maintainers to write a base devicetree file for the board or write specific devicetree files for a given board's SoC or variant.

If you're in a hurry, simple hardware can usually be supported by copy/paste followed by trial and error. If you want to understand details, you will need to read the rest of the devicetree documentation and the devicetree specification.

Example: FRDM-K64F and Hexiwear K64 This section contains concrete examples related to writing your board's devicetree.

The FRDM-K64F and Hexiwear K64 board devicetrees are defined in frdm_k64fs.dts and hexiwear_k64.dts respectively. Both boards have NXP SoCs from the same Kinetis SoC family, the K6X.

Common devicetree definitions for K6X are stored in nxp_k6x.dtsi, which is included by both board `.dts` files. nxp_k6x.dtsi in turn includes armv7-m.dtsi, which has common definitions for Arm v7-M cores.

Since nxp_k6x.dtsi is meant to be generic across K6X-based boards, it leaves many devices disabled by default using `status` properties. For example, there is a CAN controller defined as follows (with unimportant parts skipped):

```
can0: can@40024000 {
     ...
     status = "disabled";
     ...
};
```

It is up to the board `.dts` or application overlay files to enable these devices as desired, by setting `status = "okay"`. The board `.dts` files are also responsible for any board-specific configuration of the device, such as adding nodes for on-board sensors, LEDs, buttons, etc.

For example, FRDM-K64 (but not Hexiwear K64) `.dts` enables the CAN controller and sets the bus speed:

```
&can0 {
     status = "okay";
};
```

The `&can0 { ... };` syntax adds/overrides properties on the node with label `can0`, i.e. the `can@4002400` node defined in the `.dtsi` file.

Other examples of board-specific customization is pointing properties in `aliases` and `chosen` to the right nodes (see *Aliases and chosen nodes*), and making GPIO/pinmux assignments.

Write Kconfig files

Zephyr uses the Kconfig language to configure software features. Your board needs to provide some Kconfig settings before you can compile a Zephyr application for it.

Setting Kconfig configuration values is documented in detail in *Setting Kconfig configuration values*.

There is one mandatory Kconfig file in the board directory, and several optional files for a board named `plank`:

```
boards/<vendor>/plank
├── Kconfig
├── Kconfig.plank
├── Kconfig.defconfig
├── plank_defconfig
└── plank_<qualifiers>_defconfig
```

`Kconfig.plank`
: A shared Kconfig file which can be sourced both in Zephyr Kconfig and sysbuild Kconfig trees.

  This file selects the SoC in the Kconfig tree and potential other SoC related Kconfig settings. This file must not select anything outside the re-usable Kconfig board and SoC trees.

  A `Kconfig.plank` may look like this:

  ```
  config BOARD_PLANK
          select SOC_SOC1
  ```

  The Kconfig symbols `BOARD_`*`board`* and `BOARD_`*`normalized_board_target`* are constructed by the build system, therefore no type shall be defined in above code snippet.

`Kconfig`
: Included by boards/Kconfig.

  This file can add Kconfig settings which are specific to the current board.

  Not all boards have a `Kconfig` file.

A board specific setting should be defining a custom setting and usually with a prompt, like this:

```
config BOARD_FEATURE
        bool "Board specific feature"
```

If the setting name is identical to an existing Kconfig setting in Zephyr and only modifies the default value of said setting, then `Kconfig.defconfig` should be used instead.

`Kconfig.defconfig`
: Board-specific default values for Kconfig options.

  Not all boards have a `Kconfig.defconfig` file.

  The entire file should be inside an `if BOARD_PLANK` / `endif` pair of lines, like this:

```
if BOARD_PLANK

config FOO
        default y

if NETWORKING
config SOC_ETHERNET_DRIVER
        default y
endif # NETWORKING

endif # BOARD_PLANK
```

`plank_defconfig` / `plank_<qualifiers>_defconfig`
: A Kconfig fragment that is merged as-is into the final build directory `.config` whenever an application is compiled for your board.

  If both the common `plank_defconfig` file and one or more board qualifiers specific `plank_<qualifiers>_defconfig` files exist, then all matching files will be used. This allows you to place configuration which is common for all board SoCs, CPU clusters, and board variants in the base `plank_defconfig` and only place the adjustments specific for a given SoC or board variant in the `plank_<qualifiers>_defconfig`.

  The `_defconfig` should contain mandatory settings for your system clock, console, etc. The results are architecture-specific, but typically look something like this:

```
CONFIG_SYS_CLOCK_HW_CYCLES_PER_SEC=120000000  # set up your clock, etc
CONFIG_SERIAL=y
```

`plank_x_y_z_defconfig` / `plank_<qualifiers>_x_y_z_defconfig`
: A Kconfig fragment that is merged as-is into the final build directory `.config` whenever an application is compiled for your board revision `x.y.z`.

Build, test, and fix

Now it's time to build and test the application(s) you want to run on your board until you're satisfied.

For example:

```
west build -b plank samples/hello_world
west flash
```

For `west flash` to work, see *Flash and debug support* below. You can also just flash `build/zephyr/zephyr.elf`, `zephyr.hex`, or `zephyr.bin` with any other tools you prefer.

General recommendations

For consistency and to make it easier for users to build generic applications that are not board specific for your board, please follow these guidelines while porting.

- Unless explicitly recommended otherwise by this section, leave peripherals and their drivers disabled by default.
- Configure and enable a system clock, along with a tick source.
- Provide pin and driver configuration that matches the board's valuable components such as sensors, buttons or LEDs, and communication interfaces such as USB, Ethernet connector, or Bluetooth/Wi-Fi chip.
- If your board uses a well-known connector standard (like Arduino, Mikrobus, Grove, or 96Boards connectors), add connector nodes to your DTS and configure pin muxes accordingly.
- Configure components that enable the use of these pins, such as configuring an SPI instance to use the usual Arduino SPI pins.
- If available, configure and enable a serial output for the console using the `zephyr,console` chosen node in the devicetree.
- If your board supports networking, configure a default interface.
- Enable all GPIO ports connected to peripherals or expansion connectors.
- If available, enable pinmux and interrupt controller drivers.
- It is recommended to enable the MPU by default, if there is support for it in hardware. For boards with limited memory resources it is acceptable to disable it. When the MPU is enabled, it is recommended to also enable hardware stack protection (CONFIG_HW_STACK_PROTECTION=y) and, thus, allow the kernel to detect stack overflows when the system is running in privileged mode.

Flash and debug support

Zephyr supports *Building, Flashing and Debugging* via west extension commands.

To add `west flash` and `west debug` support for your board, you need to create a `board.cmake` file in your board directory. This file's job is to configure a "runner" for your board. (There's nothing special you need to do to get `west build` support for your board.)

"Runners" are Zephyr-specific Python classes that wrap *flash and debug host tools* and integrate with west and the zephyr build system to support `west flash` and related commands. Each runner supports flashing, debugging, or both. You need to configure the arguments to these Python scripts in your `board.cmake` to support those commands like this example `board.cmake`:

```
board_runner_args(jlink "--device=nrf52" "--speed=4000")
board_runner_args(pyocd "--target=nrf52" "--frequency=4000000")

include(${ZEPHYR_BASE}/boards/common/nrfjprog.board.cmake)
include(${ZEPHYR_BASE}/boards/common/jlink.board.cmake)
include(${ZEPHYR_BASE}/boards/common/pyocd.board.cmake)
```

This example configures the `nrfjprog`, `jlink`, and `pyocd` runners.

Warning

Runners usually have names which match the tools they wrap, so the `jlink` runner wraps Segger's J-Link tools, and so on. But the runner command line options like `--speed` etc. are specific to the Python scripts.

Note

Runners and board configuration should be created without being targeted to a single operating system if the tool supports multiple operating systems, nor should it rely upon special system setup/configuration. For example; do not assume that a user will have prior knowledge/configuration or (if using Linux) special udev rules installed, do not assume one specific `/dev/X` device for all platforms as this will not be compatible with Windows or macOS, and allow for overriding of the selected device so that multiple boards can be connected to a single system and flashed/debugged at the choice of the user.

For more details:

- Run `west flash --context` to see a list of available runners which support flashing, and `west flash --context -r <RUNNER>` to view the specific options available for an individual runner.
- Run `west debug --context` and `west debug --context <RUNNER>` to get the same output for runners which support debugging.
- Run `west flash --help` and `west debug --help` for top-level options for flashing and debugging.
- See *Flash and debug runners* for Python APIs.
- Look for `board.cmake` files for other boards similar to your own for more examples.

To see what a `west flash` or `west debug` command is doing exactly, run it in verbose mode:

```
west --verbose flash
west --verbose debug
```

Verbose mode prints any host tool commands the runner uses.

The order of the `include()` calls in your `board.cmake` matters. The first `include` sets the default runner if it's not already set. For example, including `nrfjprog.board.cmake` first means that `nrfjprog` is the default flash runner for this board. Since `nrfjprog` does not support debugging, `jlink` is the default debug runner.

Multiple board revisions

See *Building for a board revision* for basics on this feature from the user perspective.

Board revisions are described in the `revision` entry of the `board.yml`.

```
board:
  revision:
    format: <major.minor.patch|letter|number|custom>
    default: <default-revision-value>
    exact: <true|false>
    revisions:
    - name: <revA>
    - name: <revB>
```

Zephyr natively supports the following revision formats:

- `major.minor.patch`: match a three digit revision, such as `1.2.3`.

- `number`: matches integer revisions
- `letter`: matches single letter revisions from `A` to `Z` only

Fuzzy revision matching Fuzzy revision matching is enabled per default.

If the user selects a revision between those available, the closest revision number that is not larger than the user's choice is used. For example, if the board `plank` defines revisions `0.5.0`, and `1.5.0` and the user builds for `plank@0.7.0`, the build system will target revision `0.5.0`.

The build system will print this at CMake configuration time:

```
-- Board: plank, Revision: 0.7.0 (Active: 0.5.0)
```

This allows you to only create revision configuration files for board revision numbers that introduce incompatible changes.

Similar for `letter` where revision `A`, `D`, and `F` could be defined and the user builds for `plank@E`, the build system will target revision `D` .

Exact revision matching Exact revision matching is enabled when `exact: true` is specified in the revision section in `board.yml`.

When exact is defined then building for `plank@0.7.0` in the above example will result in the following error message:

```
Board revision `0.7.0` not found.  Please specify a valid board revision.
```

Board revision configuration adjustment When the user builds for board `plank@<revision>` it is possible to make adjustments to the board's normal configuration.

As described in the *Write your devicetree* and *Write Kconfig files* sections the board default configuration is created from the files `<board>.dts` / `<board>_<qualifiers>.dts` and `<board>_defconfig` / `<board>_<qualifiers>_defconfig`. When building for a specific board revision, the above files are used as a starting point and the following board files will be used in addition:

- `<board>_<qualifiers>_<revision>_defconfig`: a specific revision defconfig which is only used for the board and SOC / variants identified by `<board>_<qualifiers>`.
- `<board>_<revision>_defconfig`: a specific revision defconfig which is used for the board regardless of the SOC / variants.
- `<board>_<qualifiers>_<revision>.overlay`: a specific revision dts overlay which is only used for the board and SOC / variants identified by `<board>_<qualifiers>`.
- `<board>_<revision>.overlay`: a specific revision dts overlay which is used for the board regardless of the SOC / variants.

This split allows boards with multiple SoCs, multi-core SoCs, or variants to place common revision adjustments which apply to all SoCs and variants in a single file, while still providing the ability to place SoC or variant specific adjustments in a dedicated revision file.

Using the `plank` board from previous sections, then we could have the following revision adjustments:

```
boards/zephyr/plank
├── plank_0_5_0_defconfig          # Kconfig adjustment for all plank board qualifiers on␣
↪revision 0.5.0
├── plank_0_5_0.overlay            # DTS overlay for all plank board qualifiers on revision␣
↪0.5.0
```

(continues on next page)

(continued from previous page)

```
└── plank_soc1_foo_1_5_0_defconfig # Kconfig adjustment for plank board when building for␣
↪soc1 variant foo on revision 1.5.0
```

Custom revision.cmake files

Some boards may not use board revisions supported natively by Zephyr. For example string revisions.

One reason why Zephyr doesn't support string revisions is that strings can take many forms and it's not always clear if the given strings are just strings, such as `blue`, `green`, `red`, etc. or if they provide an order which can be matched against higher or lower revisions, such as `alpha`, `beta`', `gamma`.

Due to the sheer number of possibilities with strings, including the possibility of doing regex matches internally, then string revisions must be done using `custom` revision type.

To indicate to the build system that `custom` revisions are used, the format field in the `revision` section of the `board.yml` must be written as:

```
board:
  revision:
    format: custom
```

When using custom revisions then a `revision.cmake` must be created in the board directory.

The `revision.cmake` will be included by the build system when building for the board and it is the responsibility of the file to validate the revision specified by the user.

The `BOARD_REVISION` variable holds the revision value specified by the user.

To signal to the build system that it should use a different revision than the one specified by the user, `revision.cmake` can set the variable `ACTIVE_BOARD_REVISION` to the revision to use instead. The corresponding Kconfig files and devicetree overlays must be named `<board>_<ACTIVE_BOARD_REVISION>_defconfig` and `<board>_<ACTIVE_BOARD_REVISION>.overlay`.

Contributing your board

If you want to contribute your board to Zephyr, first – thanks!

There are some extra things you'll need to do:

1. Make sure you've followed all the *General recommendations*. They are requirements for boards included with Zephyr.
2. Add documentation for your board using the template file doc/templates/board.tmpl. See *Documentation Generation* for information on how to build your documentation before submitting your pull request.
3. Prepare a pull request adding your board which follows the *Contribution Guidelines*.

Board extensions

Boards already supported by Zephyr can be extended by downstream users, such as `example-application` or vendor SDKs. In some situations, certain hardware description or *choices* can not be added in the upstream Zephyr repository, but they can be in a downstream project, where custom bindings or driver classes can also be created. This feature may also be useful in development phases, when the board skeleton lives upstream, but other features are developed in a downstream module.

Board extensions are board fragments that can be present in an out-of-tree board root folder, under `${BOARD_ROOT}/boards/extensions`. Here is an example structure of an extension for the `plank` board and its revisions:

```
boards/extensions/plank
├── plank.conf                # optional
├── plank_<revision>.conf     # optional
├── plank.overlay             # optional
└── plank_<revision>.overlay  # optional
```

A board extension directory must follow the naming structure of the original board it extends. It may contain Kconfig fragments and/or devicetree overlays. Extensions are, by default, automatically loaded and applied on top of board files, before anything else. There is no guarantee on which order extensions are applied, in case multiple exist. This feature can be disabled by passing `-DBOARD_EXTENSIONS=OFF` when building.

Note that board extensions need to follow the *same guidelines* as regular boards. For example, it is wrong to enable extra peripherals or subsystems in a board extension.

Warning

Board extensions are not allowed in any module referenced in Zephyr's `west.yml` manifest file. Any board changes are required to be submitted to the main Zephyr repository.

7.8.4 Shields

Shields, also known as "add-on" or "daughter boards", attach to a board to extend its features and services for easier and modularized prototyping. In Zephyr, the shield feature provides Zephyr-formatted shield descriptions for easier compatibility with applications.

Shield porting and configuration

Shield configuration files are available in the board directory under /boards/shields:

```
boards/shields/<shield>
├── <shield>.overlay
├── Kconfig.shield
└── Kconfig.defconfig
```

These files provides shield configuration as follows:

- **<shield>.overlay**: This file provides a shield description in devicetree format that is merged with the board's *devicetree* before compilation.
- **Kconfig.shield**: This file defines shield Kconfig symbols that will be used for default shield configuration. To ease use with applications, the default shield configuration here should be consistent with those in the *Write your devicetree*.
- **Kconfig.defconfig**: This file defines the default shield configuration. It is made to be consistent with the *Write your devicetree*. Hence, shield configuration should be done by keeping in mind that features activation is application responsibility.

Besides, in order to avoid name conflicts with devices that may be defined at board level, it is advised, specifically for shields devicetree descriptions, to provide a device nodelabel is the form <device>_<shield>, for instance:

```
sdhc_myshield: sdhc@1 {
        reg = <1>;
        ...
};
```

Board compatibility

Hardware shield-to-board compatibility depends on the use of well-known connectors used on popular boards (such as Arduino and 96boards). For software compatibility, boards must also provide a configuration matching their supported connectors.

This should be done at two different level:

- Pinmux: Connector pins should be correctly configured to match shield pins
- Devicetree: A board *devicetree* file, `BOARD.dts` should define an alternate nodelabel for each connector interface. For example, for Arduino I2C:

```
arduino_i2c: &i2c1 {};
```

Board specific shield configuration If modifications are needed to fit a shield to a particular board or board revision, you can override a shield description for a specific board by adding board or board revision overriding files to a shield, as follows:

```
boards/shields/<shield>
└── boards
    ├── <board>_<revision>.overlay
    ├── <board>.overlay
    ├── <board>.defconfig
    ├── <board>_<revision>.conf
    └── <board>.conf
```

Shield activation

Activate support for one or more shields by adding the matching `--shield` arguments to the west command:

```
# From the root of the zephyr repository
west build -b None --shield x_nucleo_idb05a1 --shield x_nucleo_iks01a1 your_app
```

Alternatively, it could be set by default in a project's CMakeLists.txt:

```
set(SHIELD x_nucleo_iks01a1)
```

Shield variants

Some shields may support several variants or revisions. In that case, it is possible to provide multiple version of the shields description:

```
boards/shields/<shield>
├── <shield_v1>.overlay
├── <shield_v1>.defconfig
├── <shield_v2>.overlay
└── <shield_v2>.defconfig
```

In this case, a shield-particular revision name can be used:

```
# From the root of the zephyr repository
west build -b None --shield shield_v2 your_app
```

You can also provide a board-specific configuration to a specific shield revision:

```
boards/shields/<shield>
├── <shield_v1>.overlay
├── <shield_v1>.defconfig
├── <shield_v2>.overlay
├── <shield_v2>.defconfig
└── boards
    └── <shield_v2>
        ├── <board>.overlay
        └── <board>.defconfig
```

GPIO nexus nodes

GPIOs accessed by the shield peripherals must be identified using the shield GPIO abstraction, for example from the `arduino-header-r3` compatible. Boards that provide the header must map the header pins to SOC-specific pins. This is accomplished by including a nexus node that looks like the following into the board devicetree file:

```
arduino_header: connector {
        compatible = "arduino-header-r3";
        #gpio-cells = <2>;
        gpio-map-mask = <0xffffffff 0xffffffc0>;
        gpio-map-pass-thru = <0 0x3f>;
        gpio-map = <0 0 &gpioa 0 0>,    /* A0 */
                   <1 0 &gpioa 1 0>,    /* A1 */
                   <2 0 &gpioa 4 0>,    /* A2 */
                   <3 0 &gpiob 0 0>,    /* A3 */
                   <4 0 &gpioc 1 0>,    /* A4 */
                   <5 0 &gpioc 0 0>,    /* A5 */
                   <6 0 &gpioa 3 0>,    /* D0 */
                   <7 0 &gpioa 2 0>,    /* D1 */
                   <8 0 &gpioa 10 0>,   /* D2 */
                   <9 0 &gpiob 3 0>,    /* D3 */
                   <10 0 &gpiob 5 0>,   /* D4 */
                   <11 0 &gpiob 4 0>,   /* D5 */
                   <12 0 &gpiob 10 0>,  /* D6 */
                   <13 0 &gpioa 8 0>,   /* D7 */
                   <14 0 &gpioa 9 0>,   /* D8 */
                   <15 0 &gpioc 7 0>,   /* D9 */
                   <16 0 &gpiob 6 0>,   /* D10 */
                   <17 0 &gpioa 7 0>,   /* D11 */
                   <18 0 &gpioa 6 0>,   /* D12 */
                   <19 0 &gpioa 5 0>,   /* D13 */
                   <20 0 &gpiob 9 0>,   /* D14 */
                   <21 0 &gpiob 8 0>;   /* D15 */
};
```

This specifies how Arduino pin references like `<&arduino_header 11 0>` are converted to SOC gpio pin references like `<&gpiob 4 0>`.

In Zephyr GPIO specifiers generally have two parameters (indicated by `#gpio-cells = <2>`): the pin number and a set of flags. The low 6 bits of the flags correspond to features that can be configured in devicetree. In some cases it's necessary to use a non-zero flag value to tell the driver how a particular pin behaves, as with:

```
drdy-gpios = <&arduino_header 11 GPIO_ACTIVE_LOW>;
```

After preprocessing this becomes `<&arduino_header 11 1>`. Normally the presence of such a flag would cause the map lookup to fail, because there is no map entry with a non-zero flags value. The `gpio-map-mask` property specifies that, for lookup, all bits of the pin and all but the low 6 bits of the flags are used to identify the specifier. Then the `gpio-map-pass-thru` specifies that the low 6 bits of the flags are copied over, so the SOC GPIO reference becomes `<&gpiob 4 1>` as intended.

See nexus node for more information about this capability.

Chapter 8

Contributing to Zephyr

Contributions from the community are the backbone of the project. Whether it is by submitting code, improving documentation, or proposing new features, your efforts are highly appreciated. This page lists useful resources and guidelines to help you in your contribution journey.

8.1 General Guidelines

8.1.1 Contribution Guidelines

As an open-source project, we welcome and encourage the community to submit patches directly to the project. In our collaborative open source environment, standards and methods for submitting changes help reduce the chaos that can result from an active development community.

This document explains how to participate in project conversations, log bugs and enhancement requests, and submit patches to the project so your patch will be accepted quickly in the codebase.

Licensing

Licensing is very important to open source projects. It helps ensure the software continues to be available under the terms that the author desired.

Zephyr uses the Apache 2.0 license (as found in the LICENSE file in the project's GitHub repo) to strike a balance between open contribution and allowing you to use the software however you would like to. The Apache 2.0 license is a permissive open source license that allows you to freely use, modify, distribute and sell your own products that include Apache 2.0 licensed software. (For more information about this, check out articles such as Why choose Apache 2.0 licensing and Top 10 Apache License Questions Answered).

A license tells you what rights you have as a developer, as provided by the copyright holder. It is important that the contributor fully understands the licensing rights and agrees to them. Sometimes the copyright holder isn't the contributor, such as when the contributor is doing work on behalf of a company.

Components using other Licenses There are some imported or reused components of the Zephyr project that use other licensing, as described in *Licensing of Zephyr Project components*.

Importing code into the Zephyr OS from other projects that use a license other than the Apache 2.0 license needs to be fully understood in context and approved by the Zephyr governing board.

By carefully reviewing potential contributions and also enforcing a *Developer Certification of Origin (DCO)* for contributed code, we can ensure that the Zephyr community can develop products with the Zephyr Project without concerns over patent or copyright issues.

See *Contributing External Components* for more information about this contributing and review process for imported components.

Licensing of Zephyr Project components The Zephyr kernel tree imports or reuses packages, scripts and other files that are not covered by the Apache 2.0 License. In some places there is no LICENSE file or way to put a LICENSE file there, so we describe the licensing in this document.

scripts/{checkpatch.pl,checkstack.pl,spelling.txt}
Origin: Linux Kernel

Licensing: GPLv2 License

scripts/{coccicheck,coccinelle/array_size.cocci,coccinelle/deref_null.cocci,coccinelle/deref_null.cocci,coc
Origin: Coccinelle

Licensing: GPLv2 License

subsys/testsuite/coverage/coverage.h
Origin: GCC, the GNU Compiler Collection

Licensing: GPLv2 License with Runtime Library Exception

boards/ene/kb1200_evb/support/openocd.cfg
Licensing: GPLv2 License

Copyrights Notices

Please follow this Community Best Practice for Copyright Notices from the Linux Foundation.

Developer Certification of Origin (DCO)

To make a good faith effort to ensure licensing criteria are met, the Zephyr project requires the Developer Certificate of Origin (DCO) process to be followed.

The DCO is an attestation attached to every contribution made by every developer. In the commit message of the contribution, (described more fully later in this document), the developer simply adds a `Signed-off-by` statement and thereby agrees to the DCO.

When a developer submits a patch, it is a commitment that the contributor has the right to submit the patch per the license. The DCO agreement is shown below and at http://developercertificate.org/.

```
Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I
    have the right to submit it under the open source license
    indicated in the file; or

(b) The contribution is based upon previous work that, to the
    best of my knowledge, is covered under an appropriate open
    source license and I have the right under that license to
    submit that work with modifications, whether created in whole
    or in part by me, under the same open source license (unless
    I am permitted to submit under a different license), as
    Indicated in the file; or
```

(continues on next page)

(continued from previous page)

```
(c) The contribution was provided directly to me by some other
    person who certified (a), (b) or (c) and I have not modified
    it.

(d) I understand and agree that this project and the contribution
    are public and that a record of the contribution (including
    all personal information I submit with it, including my
    sign-off) is maintained indefinitely and may be redistributed
    consistent with this project or the open source license(s)
    involved.
```

DCO Sign-Off The "sign-off" in the DCO is a "Signed-off-by:" line in each commit's log message. The Signed-off-by: line must be in the following format:

```
Signed-off-by: Your Name <your.email@example.com>
```

For your commits, replace:

- `Your Name` with your legal name (pseudonyms, hacker handles, and the names of groups are not allowed)
- `your.email@example.com` with the same email address you are using to author the commit (CI will fail if there is no match)

You can automatically add the Signed-off-by: line to your commit body using `git commit -s`. Use other commits in the zephyr git history as examples. See *Git Setup* for instructions on configuring user and email settings in Git.

Additional requirements:

- If you are altering an existing commit created by someone else, you must add your Signed-off-by: line without removing the existing one.
- If you forget to add the Signed-off-by: line, you can add it to your previous commit by running `git commit --amend -s`.
- If you've pushed your changes to GitHub already you'll need to force push your branch after this with `git push -f`.

Notes Any contributions made as part of submitted pull requests are considered free for the Project to use. Developers are permitted to cherry-pick patches that are included in pull requests submitted by other contributors. It is expected that

- the content of the patches will not be substantially modified,
- the cherry-picked commits or portions of a commit shall preserve the original sign-off messages and the author identity.

Modifying Contributions made by other developers describes additional recommended policies around working with contributions submitted by other developers.

Prerequisites

As a contributor, you'll want to be familiar with the Zephyr project, how to configure, install, and use it as explained in the Zephyr Project website and how to set up your development environment as introduced in the Zephyr *Getting Started Guide*.

You should be familiar with common developer tools such as Git and CMake, and platforms such as GitHub.

If you haven't already done so, you'll need to create a (free) GitHub account on https://github.com and have Git tools available on your development system.

Note

The Zephyr development workflow supports all 3 major operating systems (Linux, macOS, and Windows) but some of the tools used in the sections below are only available on Linux and macOS. On Windows, instead of running these tools yourself, you will need to rely on the Continuous Integration (CI) service using Github Actions, which runs automatically on GitHub when you submit your Pull Request (PR). You can see any failure results in the workflow details link near the end of the PR conversation list. See *Continuous Integration* for more information

Source Tree Structure

To clone the main Zephyr Project repository use the instructions in *Get Zephyr and install Python dependencies*.

This section describes the main repository's source tree. In addition to the Zephyr kernel itself, you'll also find the sources for technical documentation, sample code, supported board configurations, and a collection of subsystem tests. All of these are available for developers to contribute to and enhance.

Understanding the Zephyr source tree can help locate the code associated with a particular Zephyr feature.

At the top of the tree, several files are of importance:

`CMakeLists.txt`
: The top-level file for the CMake build system, containing a lot of the logic required to build Zephyr.

`Kconfig`
: The top-level Kconfig file, which refers to the file `Kconfig.zephyr` also found in the top-level directory.

  See *the Kconfig section of the manual* for detailed Kconfig documentation.

`west.yml`
: The *West (Zephyr's meta-tool)* manifest, listing the external repositories managed by the west command-line tool.

The Zephyr source tree also contains the following top-level directories, each of which may have one or more additional levels of subdirectories not described here.

`arch`
: Architecture-specific kernel and system-on-chip (SoC) code. Each supported architecture (for example, x86 and ARM) has its own subdirectory, which contains additional subdirectories for the following areas:

  - architecture-specific kernel source files
  - architecture-specific kernel include files for private APIs

`soc`
: SoC related code and configuration files.

`boards`
: Board related code and configuration files.

`doc`
: Zephyr technical documentation source files and tools used to generate the https://docs.zephyrproject.org web content.

`drivers`
: Device driver code.

`dts`
: *devicetree* source files used to describe non-discoverable board-specific hardware details.

`include`
: Include files for all public APIs, except those defined under `lib`.

`kernel`
: Architecture-independent kernel code.

`lib`
: Library code, including the minimal standard C library.

`misc`
: Miscellaneous code that doesn't belong to any of the other top-level directories.

`samples`
: Sample applications that demonstrate the use of Zephyr features.

`scripts`
: Various programs and other files used to build and test Zephyr applications.

`cmake`
: Additional build scripts needed to build Zephyr.

`subsys`
: Subsystems of Zephyr, including:

  - USB device stack code
  - Networking code, including the Bluetooth stack and networking stacks
  - File system code
  - Bluetooth host and controller

`tests`
: Test code and benchmarks for Zephyr features.

`share`
: Additional architecture independent data. It currently contains Zephyr's CMake package.

Pull Requests and Issues

Before starting on a patch, first check in our issues Zephyr Project Issues system to see what's been reported on the issue you'd like to address. Have a conversation on the Zephyr devel mailing list (or the Zephyr Discord Server) to see what others think of your issue (and proposed solution). You may find others that have encountered the issue you're finding, or that have similar ideas for changes or additions. Send a message to the Zephyr devel mailing list to introduce and discuss your idea with the development community.

It's always a good practice to search for existing or related issues before submitting your own. When you submit an issue (bug or feature request), the triage team will review and comment on the submission, typically within a few business days.

You can find all open pull requests on GitHub and open Zephyr Project Issues in Github issues.

Git Setup

We need to know who you are, and how to contact you. To add this information to your Git installation, set the Git configuration variables `user.name` to your full name, and `user.email` to your email address.

For example, if your name is `Zephyr Developer` and your email address is `z.developer@example.com`:

```
git config --global user.name "Zephyr Developer"
git config --global user.email "z.developer@example.com"
```

Pull Request Guidelines

When opening a new Pull Request, adhere to the following guidelines to ensure compliance with Zephyr standards and facilitate the review process.

If in doubt, it's advisible to explore existing Pull Requests within the Zephyr repository. Use the search filters and labels to locate PRs related to changes similar to the ones you are proposing.

Commit Message Guidelines Changes are submitted as Git commits. Each commit has a *commit message* describing the change. Acceptable commit messages look like this:

```
[area]: [summary of change]

[Commit message body (must be non-empty)]

Signed-off-by: [Your Full Name] <[your.email@address]>
```

You need to change text in square brackets (`[like this]`) above to fit your commit.

Examples and more details follow.

Example Here is an example of a good commit message.

```
drivers: sensor: abcd1234: fix bus I/O error handling

The abcd1234 sensor driver is failing to check the flags field in
the response packet from the device which signals that an error
occurred. This can lead to reading invalid data from the response
buffer. Fix it by checking the flag and adding an error path.

Signed-off-by: Zephyr Developer <z.developer@example.com>
```

[area]: [summary of change] This line is called the commit's *title*. Titles must be:

- one line
- less than 72 characters long
- followed by a completely blank line

[area]
: The `[area]` prefix usually identifies the area of code being changed. It can also identify the change's wider context if multiple areas are affected.

  Here are some examples:

  - `doc: ...` for documentation changes
  - `drivers: foo:` for `foo` driver changes
  - `Bluetooth: Shell:` for changes to the Bluetooth shell
  - `net: ethernet:` for Ethernet-related networking changes
  - `dts:` for treewide devicetree changes

- `style:` for code style changes

If you're not sure what to use, try running `git log FILE`, where `FILE` is a file you are changing, and using previous commits that changed the same file as inspiration.

[summary of change]
The `[summary of change]` part should be a quick description of what you've done. Here are some examples:

- `doc: update wiki references to new site`
- `drivers: sensor: sensor_shell: fix channel name collision`

Warning

An empty commit message body is not permitted. Even for trivial changes, please include a descriptive commit message body. Your pull request will fail CI checks if you do not.

Commit Message Body This part of the commit should explain what your change does, and why it's needed. Be specific. A body that says `"Fixes stuff"` will be rejected. Be sure to include the following as relevant:

- **what** the change does,
- **why** you chose that approach,
- **what** assumptions were made, and
- **how** you know it works – for example, which tests you ran.

Each line in your commit message should usually be 75 characters or less. Use newlines to wrap longer lines. Exceptions include lines with long URLs, email addresses, etc.

For examples of accepted commit messages, you can refer to the Zephyr GitHub changelog.

Tip

You should have set your *Git Setup* already. Create your commit with `git commit -s` to add the Signed-off-by: line automatically using this information.

Signed-off-by: ... For open source licensing reasons, your commit must include a Signed-off-by: line that looks like this:

```
Signed-off-by: [Your Full Name] <[your.email@address]>
```

For example, if your full name is `Zephyr Developer` and your email address is `z.developer@example.com`:

```
Signed-off-by: Zephyr Developer <z.developer@example.com>
```

This means that you have personally made sure your change complies with the *Developer Certification of Origin (DCO)*. For this reason, you must use your legal name. Pseudonyms or "hacker aliases" are not permitted.

Your name and the email address you use must match the name and email in the Git commit's `Author:` field.

See the *Contributor Expectations* for a more complete discussion of contributor and reviewer expectations.

Adding links Do not include GitHub references in the commit message directly, as it can lose meaning in case the repository is forked, for example. Instead, if the change addresses a specific GitHub issue, include in the Pull Request message a line of the form:

```
Fixes #[issue number]
```

Where `[issue number]` is the relevant GitHub issue's number. For example:

```
Fixes: #1234
```

You can point to other relevant information that can be found on the web using `Link:` tags. This includes, for example: GitHub issues, datasheets, reference manuals, etc.

```
Link: https://github.com/zephyrproject-rtos/zephyr/issues/<issue number>
```

Coding Style In general, follow the Linux kernel coding style, with the following exceptions:

- The line length is 100 columns or fewer. In the documentation, longer lines for URL references are an allowed exception.
- Add braces to every `if`, `else`, `do`, `while`, `for` and `switch` body, even for single-line code blocks. Use the `--ignore BRACES` flag to make *checkpatch* stop complaining.
- Use spaces instead of tabs to align comments after declarations, as needed.
- Use C89-style single line comments, `/* */`. The C99-style single line comment, `//`, is not allowed.
- Use `/** */` for doxygen comments that need to appear in the documentation.
- Avoid using binary literals (constants starting with `0b`).
- Avoid using non-ASCII symbols in code, unless it significantly improves clarity, avoid emojis in any case.

Use these coding guidelines to ensure that your development complies with the project's style and naming conventions.

The Linux kernel GPL-licensed tool `checkpatch` is used to check coding style conformity.

> **Note**
>
> checkpatch does not currently run on Windows.

Checkpatch is available in the scripts directory. To invoke it when committing code, make the file *$ZEPHYR_BASE/.git/hooks/pre-commit* executable and edit it to contain:

```
#!/bin/sh
set -e exec
exec git diff --cached | ${ZEPHYR_BASE}/scripts/checkpatch.pl -
```

Instead of running checkpatch at each commit, you may prefer to run it only before pushing on zephyr repo. To do this, make the file *$ZEPHYR_BASE/.git/hooks/pre-push* executable and edit it to contain:

```
#!/bin/sh
remote="$1"
url="$2"

z40=0000000000000000000000000000000000000000

echo "Run push hook"
```

(continues on next page)

(continued from previous page)

```
while read local_ref local_sha remote_ref remote_sha
do
    args="$remote $url $local_ref $local_sha $remote_ref $remote_sha"
    exec ${ZEPHYR_BASE}/scripts/series-push-hook.sh $args
done

exit 0
```

If you want to override checkpatch verdict and push you branch despite reported issues, you can add option –no-verify to the git push command.

A more complete alternative to this is using *check_compliance.py* script.

clang-format The clang-format tool can be helpful to quickly reformat large amounts of new source code to our *Coding Style* standards together with the `.clang-format` configuration file provided in the repository. `clang-format` is well integrated into most editors, but you can also run it manually like this:

```
clang-format -i my_source_file.c
```

`clang-format` is part of LLVM, which can be downloaded from the project releases page. Note that if you are a Linux user, `clang-format` will likely be available as a package in your distribution repositories.

When there are differences between the *Coding Style* guidelines and the formatting generated by code formatting tools, the *Coding Style* guidelines take precedence. If there is ambiguity between formatting tools and the guidelines, maintainers may decide which style should be adopted.

Continuous Integration (CI) The Zephyr Project operates a Continuous Integration (CI) system that runs on every Pull Request (PR) in order to verify several aspects of the PR:

- Git commit formatting
- Coding Style
- Twister builds for multiple architectures and boards
- Documentation build to verify any doc changes

CI is run on Github Actions and it uses the same tools described in the *CI Tests* section. The CI results must be green indicating “All checks have passed” before the Pull Request can be merged. CI is run when the PR is created, and again every time the PR is modified with a commit.

The current status of the CI run can always be found at the bottom of the GitHub PR page, below the review status. Depending on the success or failure of the run you will see:

- “All checks have passed”
- “All checks have failed”

In case of failure you can click on the “Details” link presented below the failure message in order to navigate to `Github Actions` and inspect the results. Once you click on the link you will be taken to the `Github actions` summary results page where a table with all the different builds will be shown. To see what build or test failed click on the row that contains the failed (i.e. non-green) build.

Running CI Tests Locally

check_compliance.py The `check_compliance.py` script serves as a valuable tool for assessing code compliance with Zephyr's established guidelines and best practices. The script acts as wrapper for a suite of tools that performs various checks, including linters and formatters.

Developers are encouraged to run the script locally to validate their changes before opening a new Pull Request:

```
./scripts/ci/check_compliance.py -c upstream/main..
```

> **Note**
>
> twister is only fully supported on Linux; on Windows and MacOS the execution of tests is not supported, only building.

twister If you think your change may break some test, you can submit your PR as a draft and let the project CI automatically run the *Test Runner (Twister)* for you.

If a test fails, you can check from the CI run logs how to rerun it locally, for example:

```
west twister -p native_sim -s tests/drivers/build_all/sensor/sensors.generic_test
```

Static Code Analysis

Coverity Scan is a free service for static code analysis of Open Source projects. It is based on Coverity's commercial product and is able to analyze C, C++ and Java code.

Coverity's static code analysis doesn't run the code. Instead of that it uses abstract interpretation to gain information about the code's control flow and data flow. It's able to follow all possible code paths that a program may take. For example the analyzer understands that malloc() returns a memory that must be freed with free() later. It follows all branches and function calls to see if all possible combinations free the memory. The analyzer is able to detect all sorts of issues like resource leaks (memory, file descriptors), NULL dereferencing, use after free, unchecked return values, dead code, buffer overflows, integer overflows, uninitialized variables, and many more.

The results are available on the Coverity Scan website. In order to access the results you have to create an account yourself. From the Zephyr project page, you may select "Add me to project" to be added to the project. New members must be approved by an admin.

Static analysis of the Zephyr codebase is conducted on a bi-weekly basis. GitHub issues are automatically created for any issues detected by static analysis tools. These issues will have the same (or equivalent) priority initially defined by the tool.

To ensure accountability and efficient issue resolution, they are assigned to the respective maintainer who is responsible for the affected code.

A dedicated team comprising members with expertise in static analysis, code quality, and software security ensures the effectiveness of the static analysis process and verifies that identified issues are properly triaged and resolved in a timely manner.

Workflow If after analyzing the Coverity report it is concluded that it is a false positive please set the classification to either "False positive" or "Intentional", the action to "Ignore", owner to your own account and add a comment why the issue is considered false positive or intentional.

Update the related Github issue in the zephyr project with the details, and only close it after completing the steps above on scan service website. Any issues closed without a fix or without ignoring the entry in the scan service will be automatically reopened if the issue continues to be present in the code.

Contribution Workflow

One general practice we encourage, is to make small, controlled changes. This practice simplifies review, makes merging and rebasing easier, and keeps the change history clear and clean.

When contributing to the Zephyr Project, it is also important you provide as much information as you can about your change, update appropriate documentation, and test your changes thoroughly before submitting.

The general GitHub workflow used by Zephyr developers uses a combination of command line Git commands and browser interaction with GitHub. As it is with Git, there are multiple ways of getting a task done. We'll describe a typical workflow here:

1. Create a Fork of Zephyr to your personal account on GitHub. (Click on the fork button in the top right corner of the Zephyr project repo page in GitHub.)
2. On your development computer, change into the `zephyr` folder that was created when you *obtained the code*:

   ```
   cd zephyrproject/zephyr
   ```

   Rename the default remote pointing to the upstream repository from `origin` to `upstream`:

   ```
   git remote rename origin upstream
   ```

   Let Git know about the fork you just created, naming it `origin`:

   ```
   git remote add origin https://github.com/<your github id>/zephyr
   ```

   and verify the remote repos:

   ```
   git remote -v
   ```

   The output should look similar to:

   ```
   origin   https://github.com/<your github id>/zephyr (fetch)
   origin   https://github.com/<your github id>/zephyr (push)
   upstream https://github.com/zephyrproject-rtos/zephyr (fetch)
   upstream https://github.com/zephyrproject-rtos/zephyr (push)
   ```

3. Create a topic branch (off of `main`) for your work (if you're addressing an issue, we suggest including the issue number in the branch name):

   ```
   git checkout main
   git checkout -b fix_comment_typo
   ```

   Some Zephyr subsystems do development work on a separate branch from `main` so you may need to indicate this in your checkout:

   ```
   git checkout -b fix_out_of_date_patch origin/net
   ```

4. Make changes, test locally, change, test, test again, ... (Check out the prior chapter on *twister* as well).
5. When things look good, start the pull request process by adding your changed files:

   ```
   git add [file(s) that changed, add -p if you want to be more specific]
   ```

   You can see files that are not yet staged using:

   ```
   git status
   ```

6. Verify changes to be committed look as you expected:

```
git diff --cached
```

7. Commit your changes to your local repo:

   ```
   git commit -s
   ```

   The `-s` option automatically adds your `Signed-off-by:` to your commit message. Your commit will be rejected without this line that indicates your agreement with the *Developer Certification of Origin (DCO)*. See the *Commit Message Guidelines* section for specific guidelines for writing your commit messages.

8. Push your topic branch with your changes to your fork in your personal GitHub account:

   ```
   git push origin fix_comment_typo
   ```

9. In your web browser, go to your forked repo and click on the `Compare & pull request` button for the branch you just worked on and you want to open a pull request with.

10. Review the pull request changes, and verify that you are opening a pull request for the `main` branch. The title and message from your commit message should appear as well.

11. A bot will assign one or more suggested reviewers (based on the MAINTAINERS file in the repo). If you are a project member, you can select additional reviewers now too.

12. Click on the submit button and your pull request is sent and awaits review. Email will be sent as review comments are made, or you can check on your pull request at https://github.com/zephyrproject-rtos/zephyr/pulls.

    Note

    As more commits are merged upstream, the GitHub PR page will show a `This branch is out-of-date with the base branch` message and a `Update branch` button on the PR page. That message should be ignored, as the commits will be rebased as part of merging anyway, and triggering a branch update from the GitHub UI will cause the PR approvals to be dropped.

13. While you're waiting for your pull request to be accepted and merged, you can create another branch to work on another issue. (Be sure to make your new branch off of `main` and not the previous branch.):

    ```
    git checkout main
    git checkout -b fix_another_issue
    ```

    and use the same process described above to work on this new topic branch.

14. If reviewers do request changes to your patch, you can interactively rebase commit(s) to fix review issues. In your development repo:

    ```
    git rebase -i <offending-commit-id>^
    ```

    In the interactive rebase editor, replace `pick` with `edit` to select a specific commit (if there's more than one in your pull request), or remove the line to delete a commit entirely. Then edit files to fix the issues in the review.

    As before, inspect and test your changes. When ready, continue the patch submission:

    ```
    git add [file(s)]
    git rebase --continue
    ```

    Update commit comment if needed, and continue:

```
git push --force origin fix_comment_typo
```

By force pushing your update, your original pull request will be updated with your changes so you won't need to resubmit the pull request.

15. After pushing the requested change, check on the PR page if there is a merge conflict. If so, rebase your local branch:

```
git fetch --all
git rebase --ignore-whitespace upstream/main
```

The `--ignore-whitespace` option stops `git apply` (called by rebase) from changing any whitespace. Resolve the conflicts and push again:

```
git push --force origin fix_comment_typo
```

> **Note**
>
> While amending commits and force pushing is a common review model outside GitHub, and the one recommended by Zephyr, it's not the main model supported by GitHub. Forced pushes can cause unexpected behavior, such as not being able to use "View Changes" buttons except for the last one - GitHub complains it can't find older commits. You're also not always able to compare the latest reviewed version with the latest submitted version. When rewriting history GitHub only guarantees access to the latest version.

16. If the CI run fails, you will need to make changes to your code in order to fix the issues and amend your commits by rebasing as described above. Additional information about the CI system can be found in *Continuous Integration*.

Contribution Tips The following is a list of tips to improve and accelerate the review process of Pull Requests. If you follow them, chances are your pull request will get the attention needed and it will be ready for merge sooner than later:

1. When pushing follow-up changes, use the `--keep-base` option of git-rebase
2. On the PR page, check if the change can still be merged with no merge conflicts
3. Make sure title of PR explains what is being fixed or added
4. Make sure your PR has a body with more details about the content of your submission
5. Make sure you reference the issue you are fixing in the body of the PR
6. Watch early CI results immediately after submissions and fix issues as they are discovered
7. Revisit PR after 1-2 hours to see the status of all CI checks, make sure all is green
8. If you get request for changes and submit a change to address them, make sure you click the "Re-request review" button on the GitHub UI to notify those who asked for the changes

Submitting Proposals You can request a new feature or submit a proposal by submitting an issue to our GitHub Repository. If you would like to implement a new feature, please submit an issue with a proposal (RFC) for your work first, to be sure that we can use it. Please consider what kind of change it is:

- For a Major Feature, first open an issue and outline your proposal so that it can be discussed. This will also allow us to better coordinate our efforts, prevent duplication of work, and help you to craft the change so that it is successfully accepted into the project. Providing the following information will increase the chances of your issue being dealt with quickly:

  - Overview of the Proposal
  - Motivation for or Use Case
  - Design Details
  - Alternatives
  - Test Strategy
- Small Features can be crafted and directly submitted as a Pull Request.

Identifying Contribution Origin When adding a new file to the tree, it is important to detail the source of origin on the file, provide attributions, and detail the intended usage. In cases where the file is an original to Zephyr, the commit message should include the following ("Original" is the assumption if no Origin tag is present):

```
Origin: Original
```

In cases where the file is *imported from an external project*, the commit message shall contain details regarding the original project, the location of the project, the SHA-id of the origin commit for the file and the intended purpose.

For example, a copy of a locally maintained import:

```
Origin: Contiki OS
License: BSD 3-Clause
URL: http://www.contiki-os.org/
commit: 853207acfdc6549b10eb3e44504b1a75ae1ad63a
Purpose: Introduction of networking stack.
```

For example, a copy of an externally maintained import in a module repository:

```
Origin: Tiny Crypt
License: BSD 3-Clause
URL: https://github.com/01org/tinycrypt
commit: 08ded7f21529c39e5133688ffb93a9d0c94e5c6e
Purpose: Introduction of TinyCrypt
```

Contributions to External Modules

Follow the guidelines in the *Modules (External projects)* section for contributing *new modules* and submitting changes to *existing modules*.

Treewide Changes

This section describes contributions that are treewide changes and some additional associated requirements that apply to them. These requirements exist to try to give such changes increased review and user visibility due to their large impact.

Definition and Decision Making A *treewide change* is defined as any change to Zephyr APIs, coding practices, or other development requirements that either implies required changes throughout the zephyr source code repository or can reasonably be expected to do so for a wide class of external Zephyr-based source code.

This definition is informal by necessity. This is because the decision on whether any particular change is treewide can be subjective and may depend on additional context.

Project maintainers should use good judgement and prioritize the Zephyr developer experience when deciding when a proposed change is treewide. Protracted disagreements can be resolved by the Zephyr Project's Technical Steering Committee (TSC), but please avoid premature escalation to the TSC.

Requirements for Treewide Changes

- The zephyr repository must apply the 'treewide' GitHub label to any issues or pull requests that are treewide changes
- The person proposing a treewide change must create an RFC issue describing the change, its rationale and impact, etc. before any pull requests related to the change can be merged
- The project's Architecture Working Group (WG) must include the issue on the agenda and discuss whether the project will accept or reject the change before any pull requests related to the change can be merged (with escalation to the TSC if consensus is not reached at the WG)
- The Architecture WG must specify the procedure for merging any PRs associated with each individual treewide change, including any required approvals for pull requests affecting specific subsystems or extra review time requirements
- The person proposing a treewide change must email devel@lists.zephyrproject.org about the RFC if it is accepted by the Architecture WG before any pull requests related to the change can be merged

Examples Some example past treewide changes are:

- the deprecation of version 1 of the *Logging API* in favor of version 2 (see commit 262cc55609)
- the removal of support for a legacy *Devicetree bindings* syntax (6bf761fc0a)

Note that adding a new version of a widely used API while maintaining support for the old one is not a treewide change. Deprecation and removal of such APIs, however, are treewide changes.

Specialized driver requirements

Drivers for standalone devices should use the Zephyr bus APIs (SPI, I2C...) whenever possible so that the device can be used with any SoC from any vendor implementing a compatible bus.

If it is not technically possible to achieve full performance using the Zephyr APIs due to specialized accelerators in a particular SoC family, one could extend the support for an external device by providing a specialized path for that SoC family. However, the driver must still provide a regular path (via Zephyr APIs) for all other SoCs. Every exception must be approved by the Architecture WG in order to be validated and potentially to be learned/improved from.

8.1.2 Coding Guidelines

The project TSC and the Safety Committee of the project agreed to implement a staged and incremental approach for complying with a set of coding rules (AKA Coding Guidelines) to improve quality and consistency of the code base. Below are the agreed upon stages:

Stage I (COMPLETED)
: Coding guideline rules are available to be followed and referenced, but not enforced. Rules are not yet enforced in CI and pull-requests cannot be blocked by reviewers/approvers due to violations.

Stage II
: Begin enforcement on a limited scope of the code base. Initially, this would be the safety certification scope. For rules easily applied across codebase, we should not limit compliance to initial scope. This step requires tooling, CI setup and an enforcement strategy.

Stage III
: Revisit the coding guideline rules and based on experience from previous stages, refine/iterate on selected rules.

Stage IV
: Expand enforcement to the wider codebase. Exceptions may be granted on some areas of the codebase with a proper justification. Exception would require TSC approval.

Note

Coding guideline rules may be removed/changed at any time by filing a GH issue/RFC.

Important

Current stage: The prerequisites for entering **Stage II** are currently being looked at: The tooling is in evaluation, CI setup and enforcement strategy is being worked on.

Main rules

The coding guideline rules are based on MISRA-C 2012 and are a subset of MISRA-C. The subset is listed in the table below with a summary of the rules, its severity and the equivalent rules from other standards for reference.

Note

For existing Zephyr maintainers and collaborators, if you are unable to obtain a copy through your employer, a limited number of copies will be made available through the project. If you need a copy of MISRA-C 2012, please send email to safety@lists.zephyrproject.org and provide details on reason why you can't obtain one through other options and expected contributions once you have one. The safety committee will review all requests.

Table 1: Main rules

MISRA C 2012	Severity	Description	CERT C	Example
Dir 1.1	Required	Any implementation-defined behaviour on which the output of the program depends shall be documented and understood	MSC09-C	Dir 1.1
Dir 2.1	Required	All source files shall compile without any compilation errors	N/A	Dir 2.1
Dir 3.1	Required	All code shall be traceable to documented requirements	N/A	Dir 3.1
Dir 4.1	Required	Run-time failures shall be minimized	N/A	Dir 4.1
Dir 4.2	Advisory	All usage of assembly language should be documented	N/A	Dir 4.2

continues on next page

Table 1 – continued from previous page

MISRA C 2012	Severity	Description	CERT C	Example
Dir 4.4	Advisory	Sections of code should not be “commented out”	MSC04-C	Dir 4.4
Dir 4.5	Advisory	Identifiers in the same name space with overlapping visibility should be typographically unambiguous	DCL02-C	Dir 4.5
Dir 4.6	Advisory	typedefs that indicate size and signedness should be used in place of the basic numerical types	N/A	Dir 4.6
Dir 4.7	Required	If a function returns error information, then that error information shall be tested	N/A	Dir 4.7
Dir 4.8	Advisory	If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden	DCL12-C	Dir 4.8 example 1 Dir 4.8 example 2
Dir 4.9	Advisory	A function should be used in preference to a function-like macro where they are interchangeable	PRE00-C	Dir 4.9
Dir 4.10	Required	Precautions shall be taken in order to prevent the contents of a header file being included more than once	PRE06-C	Dir 4.10
Dir 4.11	Required	The validity of values passed to library functions shall be checked	N/A	Dir 4.11
Dir 4.12	Required	Dynamic memory allocation shall not be used	STR01-C	Dir 4.12
Dir 4.13	Advisory	Functions which are designed to provide operations on a resource should be called in an appropriate sequence	N/A	Dir 4.13
Dir 4.14	Required	The validity of values received from external sources shall be checked	N/A	Dir 4.14
Rule 1.2	Advisory	Language extensions should not be used	MSC04-C	Rule 1.2
Rule 1.3	Required	There shall be no occurrence of undefined or critical unspecified behaviour	N/A	Rule 1.3
Rule 2.1	Required	A project shall not contain unreachable code	MSC07-C	Rule 2.1 example 1 Rule 2.1 example 2
Rule 2.2	Required	There shall be no dead code	MSC12-C	Rule 2.2
Rule 2.3	Advisory	A project should not contain unused type declarations	N/A	Rule 2.3
Rule 2.6	Advisory	A function should not contain unused label declarations	N/A	Rule 2.6
Rule 2.7	Advisory	There should be no unused parameters in functions	N/A	Rule 2.7

continues on next page

Table 1 – continued from previous page

MISRA C 2012	Severity	Description	CERT C	Example
Rule 3.1	Required	The character sequences /* and // shall not be used within a comment	MSC04-C	Rule 3.1
Rule 3.2	Required	Line-splicing shall not be used in // comments	N/A	Rule 3.2
Rule 4.1	Required	Octal and hexadecimal escape sequences shall be terminated	MSC09-C	Rule 4.1
Rule 4.2	Advisory	Trigraphs should not be used	PRE07-C	Rule 4.2
Rule 5.1	Required	External identifiers shall be distinct	DCL23-C	Rule 5.1 example 1 Rule 5.1 example 2
Rule 5.2	Required	Identifiers declared in the same scope and name space shall be distinct	DCL23-C	Rule 5.2
Rule 5.3	Required	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope	DCL23-C	Rule 5.3
Rule 5.4	Required	Macro identifiers shall be distinct	DCL23-C	Rule 5.4
Rule 5.5	Required	Identifiers shall be distinct from macro names	DCL23-C	Rule 5.5
Rule 5.6	Required	A typedef name shall be a unique identifier	N/A	Rule 5.6
Rule 5.7	Required	A tag name shall be a unique identifier	N/A	Rule 5.7
Rule 5.8	Required	Identifiers that define objects or functions with external linkage shall be unique	N/A	Rule 5.8 example 1 Rule 5.8 example 2
Rule 5.9	Advisory	Identifiers that define objects or functions with internal linkage should be unique	N/A	Rule 5.9 example 1 Rule 5.9 example 2
Rule 6.1	Required	Bit-fields shall only be declared with an appropriate type	INT14-C	Rule 6.1
Rule 6.2	Required	Single-bit named bit fields shall not be of a signed type	INT14-C	Rule 6.2
Rule 7.1	Required	Octal constants shall not be used	DCL18-C	Rule 7.1
Rule 7.2	Required	A u or U suffix shall be applied to all integer constants that are represented in an unsigned type	N/A	Rule 7.2

continues on next page

Table 1 – continued from previous page

MISRA C 2012	Severity	Description	CERT C	Example
Rule 7.3	Required	The lowercase character l shall not be used in a literal suffix	DCL16-C	Rule 7.3
Rule 7.4	Required	A string literal shall not be assigned to an object unless the objects type is pointer to const-qualified char	N/A	Rule 7.4
Rule 8.1	Required	Types shall be explicitly specified	N/A	Rule 8.1
Rule 8.2	Required	Function types shall be in prototype form with named parameters	DCL20-C	Rule 8.2
Rule 8.3	Required	All declarations of an object or function shall use the same names and type qualifiers	N/A	Rule 8.3
Rule 8.4	Required	A compatible declaration shall be visible when an object or function with external linkage is defined	N/A	Rule 8.4
Rule 8.5	Required	An external object or function shall be declared once in one and only one file	N/A	Rule 8.5 example 1 Rule 8.5 example 2
Rule 8.6	Required	An identifier with external linkage shall have exactly one external definition	N/A	Rule 8.6 example 1 Rule 8.6 example 2
Rule 8.8	Required	The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage	DCL15-C	Rule 8.8
Rule 8.9	Advisory	An object should be defined at block scope if its identifier only appears in a single function	DCL19-C	Rule 8.9
Rule 8.10	Required	An inline function shall be declared with the static storage class	N/A	Rule 8.10
Rule 8.12	Required	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique	INT09-C	Rule 8.12
Rule 8.14	Required	The restrict type qualifier shall not be used	N/A	Rule 8.14
Rule 9.1	Mandatory	The value of an object with automatic storage duration shall not be read before it has been set	N/A	Rule 9.1
Rule 9.2	Required	The initializer for an aggregate or union shall be enclosed in braces	N/A	Rule 9.2
Rule 9.3	Required	Arrays shall not be partially initialized	N/A	Rule 9.3
Rule 9.4	Required	An element of an object shall not be initialized more than once	N/A	Rule 9.4

continues on next page

Table 1 – continued from previous page

MISRA C 2012	Severity	Description	CERT C	Example
Rule 9.5	Required	Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly	N/A	Rule 9.5
Rule 10.1	Required	Operands shall not be of an inappropriate essential type	STR04-C	Rule 10.1
Rule 10.2	Required	Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations	STR04-C	Rule 10.2
Rule 10.3	Required	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category	STR04-C	Rule 10.3
Rule 10.4	Required	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category	STR04-C	Rule 10.4
Rule 10.5	Advisory	The value of an expression should not be cast to an inappropriate essential type	N/A	Rule 10.5
Rule 10.6	Required	The value of a composite expression shall not be assigned to an object with wider essential type	INT02-C	Rule 10.6
Rule 10.7	Required	If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type	INT02-C	Rule 10.7
Rule 10.8	Required	The value of a composite expression shall not be cast to a different essential type category or a wider essential type	INT02-C	Rule 10.8
Rule 11.2	Required	Conversions shall not be performed between a pointer to an incomplete type and any other type	N/A	Rule 11.2
Rule 11.6	Required	A cast shall not be performed between pointer to void and an arithmetic type	N/A	Rule 11.6
Rule 11.7	Required	A cast shall not be performed between pointer to object and a noninteger arithmetic type	N/A	Rule 11.7
Rule 11.8	Required	A cast shall not remove any const or volatile qualification from the type pointed to by a pointer	EXP05-C	Rule 11.8
Rule 11.9	Required	The macro NULL shall be the only permitted form of integer null pointer constant	N/A	Rule 11.9
Rule 12.1	Advisory	The precedence of operators within expressions should be made explicit	EXP00-C	Rule 12.1
Rule 12.2	Required	The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand	N/A	Rule 12.2
Rule 12.4	Advisory	Evaluation of constant expressions should not lead to unsigned integer wrap-around	N/A	Rule 12.4

continues on next page

Table 1 – continued from previous page

MISRA C 2012	Severity	Description	CERT C	Example
Rule 12.5	Mandatory	The sizeof operator shall not have an operand which is a function parameter declared as “array of type”	N/A	Rule 12.5
Rule 13.1	Required	Initializer lists shall not contain persistent side effects	N/A	Rule 13.1 example 1 Rule 13.1 example 2
Rule 13.2	Required	The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders	N/A	Rule 13.2
Rule 13.3	Advisory	A full expression containing an increment (++) or decrement (–) operator should have no other potential side effects other than that caused by the increment or decrement operator	N/A	Rule 13.3
Rule 13.4	Advisory	The result of an assignment operator should not be used	N/A	Rule 13.4
Rule 13.5	Required	The right hand operand of a logical && or || operator shall not contain persistent side effects	EXP10-C	Rule 13.5 example 1 Rule 13.5 example 2
Rule 13.6	Mandatory	The operand of the sizeof operator shall not contain any expression which has potential side effects	N/A	Rule 13.6
Rule 14.1	Required	A loop counter shall not have essentially floating type	N/A	Rule 14.1
Rule 14.2	Required	A for loop shall be well-formed	N/A	Rule 14.2
Rule 14.3	Required	Controlling expressions shall not be invariant	N/A	Rule 14.3
Rule 14.4	Required	The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type	N/A	Rule 14.4
Rule 15.2	Required	The goto statement shall jump to a label declared later in the same function	N/A	Rule 15.2
Rule 15.3	Required	Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement	N/A	Rule 15.3
Rule 15.6	Required	The body of an iteration-statement or a selection-statement shall be a compound-statement	EXP19-C	Rule 15.6
Rule 15.7	Required	All if else if constructs shall be terminated with an else statement	N/A	Rule 15.7

continues on next page

Table 1 – continued from previous page

MISRA C 2012	Severity	Description	CERT C	Example
Rule 16.1	Required	All switch statements shall be well-formed	N/A	Rule 16.1
Rule 16.2	Required	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement	MSC20-C	Rule 16.2
Rule 16.3	Required	An unconditional break statement shall terminate every switch-clause	N/A	Rule 16.3
Rule 16.4	Required	Every switch statement shall have a default label	N/A	Rule 16.4
Rule 16.5	Required	A default label shall appear as either the first or the last switch label of a switch statement	N/A	Rule 16.5
Rule 16.6	Required	Every switch statement shall have at least two switch-clauses	N/A	Rule 16.6
Rule 16.7	Required	A switch-expression shall not have essentially Boolean type	N/A	Rule 16.7
Rule 17.1	Required	The features of <stdarg.h> shall not be used	ERR00-C	Rule 17.1
Rule 17.2	Required	Functions shall not call themselves, either directly or indirectly	MEM05-C	Rule 17.2
Rule 17.3	Mandatory	A function shall not be declared implicitly	N/A	Rule 17.3
Rule 17.4	Mandatory	All exit paths from a function with non-void return type shall have an explicit return statement with an expression	N/A	Rule 17.4
Rule 17.5	Advisory	The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements	N/A	Rule 17.5
Rule 17.6	Mandatory	The declaration of an array parameter shall not contain the static keyword between the [ ]	N/A	Rule 17.6
Rule 17.7	Required	The value returned by a function having non-void return type shall be used	N/A	Rule 17.7
Rule 18.1	Required	A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand	EXP08-C	Rule 18.1
Rule 18.2	Required	Subtraction between pointers shall only be applied to pointers that address elements of the same array	EXP08-C	Rule 18.2
Rule 18.3	Required	The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object	EXP08-C	Rule 18.3
Rule 18.5	Advisory	Declarations should contain no more than two levels of pointer nesting	N/A	Rule 18.5

continues on next page

Table 1 – continued from previous page

MISRA C 2012	Severity	Description	CERT C	Example
Rule 18.6	Required	The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist	N/A	Rule 18.6 example 1 Rule 18.6 example 2
Rule 18.8	Required	Variable-length array types shall not be used	N/A	Rule 18.8
Rule 19.1	Mandatory	An object shall not be assigned or copied to an overlapping object	N/A	Rule 19.1
Rule 20.2	Required	The ‘, or characters and the /* or // character sequences shall not occur in a header file name”	N/A	Rule 20.2
Rule 20.3	Required	The #include directive shall be followed by either a <filename> or “filename” sequence	N/A	Rule 20.3
Rule 20.4	Required	A macro shall not be defined with the same name as a keyword	N/A	Rule 20.4
Rule 20.7	Required	Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses	PRE01-C	Rule 20.7
Rule 20.8	Required	The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1	N/A	Rule 20.8
Rule 20.9	Required	All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #defined before evaluation	N/A	Rule 20.9
Rule 20.11	Required	A macro parameter immediately following a # operator shall not immediately be followed by a ## operator	N/A	Rule 20.11
Rule 20.12	Required	A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators	N/A	Rule 20.12
Rule 20.13	Required	A line whose first token is # shall be a valid preprocessing directive	N/A	Rule 20.13
Rule 20.14	Required	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related	N/A	Rule 20.14
Rule 21.1	Required	#define and #undef shall not be used on a reserved identifier or reserved macro name	N/A	Rule 21.1
Rule 21.2	Required	A reserved identifier or macro name shall not be declared	N/A	Rule 21.2
Rule 21.3	Required	The memory allocation and deallocation functions of <stdlib.h> shall not be used	MSC24-C	Rule 21.3
Rule 21.4	Required	The standard header file <setjmp.h> shall not be used	N/A	Rule 21.4

continues on next page

Table 1 – continued from previous page

MISRA C 2012	Severity	Description	CERT C	Example
Rule 21.6	Required	The Standard Library input/output functions shall not be used	N/A	Rule 21.6
Rule 21.7	Required	The atof, atoi, atol and atoll functions of <stdlib.h> shall not be used	N/A	Rule 21.7
Rule 21.9	Required	The library functions bsearch and qsort of <stdlib.h> shall not be used	N/A	Rule 21.9
Rule 21.11	Required	The standard header file <tgmath.h> shall not be used	N/A	Rule 21.11
Rule 21.12	Advisory	The exception handling features of <fenv.h> should not be used	N/A	Rule 21.12
Rule 21.13	Mandatory	Any value passed to a function in <ctype.h> shall be representable as an unsigned char or be the value EO	N/A	Rule 21.13
Rule 21.14	Required	The Standard Library function memcmp shall not be used to compare null terminated strings	N/A	Rule 21.14
Rule 21.15	Required	The pointer arguments to the Standard Library functions memcpy, memmove and memcmp shall be pointers to qualified or unqualified versions of compatible types	N/A	Rule 21.15
Rule 21.16	Required	The pointer arguments to the Standard Library function memcmp shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an essentially enum type	N/A	Rule 21.16
Rule 21.17	Mandatory	Use of the string handling functions from <string.h> shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters	N/A	Rule 21.17
Rule 21.18	Mandatory	The size_t argument passed to any function in <string.h> shall have an appropriate value	N/A	Rule 21.18
Rule 21.19	Mandatory	The pointers returned by the Standard Library functions localeconv, getenv, setlocale or, strerror shall only be used as if they have pointer to const-qualified type	N/A	Rule 21.19
Rule 21.20	Mandatory	The pointer returned by the Standard Library functions asctime, ctime, gmtime, localtime, localeconv, getenv, setlocale or strerror shall not be used following a subsequent call to the same function	N/A	Rule 21.20
Rule 22.1	Required	All resources obtained dynamically by means of Standard Library functions shall be explicitly released	N/A	Rule 22.1
Rule 22.2	Mandatory	A block of memory shall only be freed if it was allocated by means of a Standard Library function	N/A	Rule 22.2

continues on next page

Table 1 – continued from previous page

MISRA C 2012	Severity	Description	CERT C	Example
Rule 22.3	Required	The same file shall not be open for read and write access at the same time on different streams	N/A	Rule 22.3
Rule 22.4	Mandatory	There shall be no attempt to write to a stream which has been opened as read-only	N/A	Rule 22.4
Rule 22.5	Mandatory	A pointer to a FILE object shall not be dereferenced	N/A	Rule 22.5
Rule 22.6	Mandatory	The value of a pointer to a FILE shall not be used after the associated stream has been closed	N/A	Rule 22.6
Rule 22.7	Required	The macro EOF shall only be compared with the unmodified return value from any Standard Library function capable of returning EOF	N/A	Rule 22.7
Rule 22.8	Required	The value of errno shall be set to zero prior to a call to an errno-setting-function	N/A	Rule 22.8
Rule 22.9	Required	The value of errno shall be tested against zero after calling an errno-setting-function	N/A	Rule 22.9
Rule 22.10	Required	The value of errno shall only be tested when the last function to be called was an errno-setting-function	N/A	Rule 22.10

Additional rules

Rule A.1: Conditional Compilation

Severity Required

Description Do not conditionally compile function declarations in header files. Do not conditionally compile structure declarations in header files. You may conditionally exclude fields within structure definitions to avoid wasting memory when the feature they support is not enabled.

Rationale Excluding declarations from the header based on compile-time options may prevent their documentation from being generated. Their absence also prevents use of `if (IS_ENABLED(CONFIG_FOO)) {}` as an alternative to preprocessor conditionals when the code path should change based on the selected options.

Rule A.2: Inclusive Language

Severity Required

Description Do not introduce new usage of offensive terms listed below. This rule applies but is not limited to source code, comments, documentation, and branch names. Replacement terms may vary by area or subsystem, but should aim to follow updated industry standards when possible.

Exceptions are allowed for maintaining existing implementations or adding new implementations of industry standard specifications governed externally to the Zephyr Project.

Existing usage is recommended to change as soon as updated industry standard specifications become available or new terms are publicly announced by the governing body, or immediately if no specifications apply.

Offensive Terms	Recommended Replacements
`{master,leader} / slave`	• `{primary,main} / {secondary, replica}` • `{initiator,requester} / {target, responder}` • `{controller,host} / {device,worker, proxy,target}` • `director / performer` • `central / peripheral`
`blacklist / whitelist`	• `denylist / allowlist` • `blocklist / allowlist` • `rejectlist / acceptlist`
`grandfather policy`	• `legacy`
`sanity`	• `coherence` • `confidence`

Rationale Offensive terms do not create an inclusive community environment and therefore violate the Zephyr Project Code of Conduct. This coding rule was inspired by a similar rule in Linux.

Status Related GitHub Issues and Pull Requests are tagged with the Inclusive Language Label.

Area	Selected Replacements	Status
API	See Bluetooth Appropriate Language Mapping Tables	
CAN	This CAN in Automation Inclusive Language news post has a list of general recommendations. See CAN in Automation Inclusive Language for terms to be used in specification document updates.	
eSPI	• `master / slave` => `controller / target`	Refer to eSPI Specification for new terminology
gPTP	• `master / slave` => TBD	
Inter-Integrated Circuit (I2C) Bus	• `master / slave` => TBD	NXP publishes the I2C Specification and has selected `controller / target` as replacement terms, but the timing to publish an announcement or new specification is TBD. Zephyr will update I2C when replacement terminology is confirmed by a public announcement or updated specification. See Zephyr issue 27033.
Inter-IC Sound (I2S) Bus	• `master / slave` => TBD	
SMP/AMP	• `master / slave` => TBD	
Serial Peripheral Interface (SPI) Bus	• `master / slave` => `controller / peripheral` • `MOSI / MISO / SS` => `SDO / SDI / CS`	The Open Source Hardware Association has selected these replacement terms. See OSHWA Resolution to Redefine SPI Signal Names
Test Runner (Twister)	• `platform_whitelist` => `platform_allow` • `sanitycheck` => `twister`	

Rule A.3: Macro name collisions

Severity Required

Description Macros with commonly used names such as `MIN`, `MAX`, `ARRAY_SIZE`, must not be modified or protected to avoid name collisions with other implementations. In particular, they must not be prefixed to place them in a Zephyr-specific namespace, re-defined using `#undef`, or conditionally excluded from compilation using `#ifndef`. Instead, if a conflict arises with an

existing definition originating from a *module*, the module's code itself needs to be modified (ideally upstream, alternatively via a change in Zephyr's own fork). This rule applies to Zephyr as a project in general, regardless of the time of introduction of the macro or its current name in the tree. If a macro name is commonly used in several other well-known open source projects then the implementation in Zephyr should use that name. While there is a subjective and non-measurable component to what "commonly used" means, the ultimate goal is to offer users familiar macros. Finally, this rule applies to inter-module name collisions as well: in that case both modules, prior to their inclusion, should be modified to use module-specific versions of the macro name that collides.

Rationale Zephyr is an RTOS that comes with additional functionality and dependencies in the form of modules. Those modules are typically independent projects that may use macro names that can conflict with other modules or with Zephyr itself. Since, in the context of this documentation, Zephyr is considered the central or main project, it should implement the non-namespaced versions of the macros. Given that Zephyr uses a fork of the corresponding upstream for each module, it is always possible to patch the macro implementation in each module to avoid collisions.

Rule A.4: C Standard Library Usage Restrictions in Zephyr Kernel

Severity Required

Description The use of the C standard library functions and macros in the Zephyr kernel shall be limited to the following functions and macros from the ISO/IEC 9899:2011 standard, also known as C11, and their extensions:

Table 2: List of allowed libc functions and macros in the Zephyr kernel

Function	Source
abort()	ISO/IEC 9899:2011
abs()	ISO/IEC 9899:2011
aligned_alloc()	ISO/IEC 9899:2011
assert()	ISO/IEC 9899:2011
atoi()	ISO/IEC 9899:2011
bsearch()	ISO/IEC 9899:2011
calloc()	ISO/IEC 9899:2011
exit()	ISO/IEC 9899:2011
fprintf()	ISO/IEC 9899:2011
fputc()	ISO/IEC 9899:2011
fputs()	ISO/IEC 9899:2011
free()	ISO/IEC 9899:2011
fwrite()	ISO/IEC 9899:2011
gmtime()	ISO/IEC 9899:2011
isalnum()	ISO/IEC 9899:2011
isalpha()	ISO/IEC 9899:2011
iscntrl()	ISO/IEC 9899:2011
isdigit()	ISO/IEC 9899:2011
isgraph()	ISO/IEC 9899:2011
isprint()	ISO/IEC 9899:2011
isspace()	ISO/IEC 9899:2011
isupper()	ISO/IEC 9899:2011
isxdigit()	ISO/IEC 9899:2011

continues on next page

Table 2 – continued from previous page

Function	Source
labs()	ISO/IEC 9899:2011
llabs()	ISO/IEC 9899:2011
malloc()	ISO/IEC 9899:2011
memchr()	ISO/IEC 9899:2011
memcmp()	ISO/IEC 9899:2011
memcpy()	ISO/IEC 9899:2011
memmove()	ISO/IEC 9899:2011
memset()	ISO/IEC 9899:2011
perror()	ISO/IEC 9899:2011
printf()	ISO/IEC 9899:2011
putc()	ISO/IEC 9899:2011
putchar()	ISO/IEC 9899:2011
puts()	ISO/IEC 9899:2011
qsort()	ISO/IEC 9899:2011
rand()	ISO/IEC 9899:2011
realloc()	ISO/IEC 9899:2011
snprintf()	ISO/IEC 9899:2011
sprintf()	ISO/IEC 9899:2011
sqrt()	ISO/IEC 9899:2011
sqrtf()	ISO/IEC 9899:2011
srand()	ISO/IEC 9899:2011
strcat()	ISO/IEC 9899:2011
strchr()	ISO/IEC 9899:2011
strcmp()	ISO/IEC 9899:2011
strcpy()	ISO/IEC 9899:2011
strcspn()	ISO/IEC 9899:2011
strerror()	ISO/IEC 9899:2011
strlen()	ISO/IEC 9899:2011
strncat()	ISO/IEC 9899:2011
strncmp()	ISO/IEC 9899:2011
strncpy()	ISO/IEC 9899:2011
strnlen()	POSIX.1-2008
strrchr()	ISO/IEC 9899:2011
strspn()	ISO/IEC 9899:2011
strstr()	ISO/IEC 9899:2011
strtol()	ISO/IEC 9899:2011
strtoll()	ISO/IEC 9899:2011
strtoul()	ISO/IEC 9899:2011
strtoull()	ISO/IEC 9899:2011
time()	ISO/IEC 9899:2011
tolower()	ISO/IEC 9899:2011
toupper()	ISO/IEC 9899:2011
vfprintf()	ISO/IEC 9899:2011
vprintf()	ISO/IEC 9899:2011
vsnprintf()	ISO/IEC 9899:2011
vsprintf()	ISO/IEC 9899:2011

All of the functions listed above must be implemented by the *minimal libc* to ensure that the Zephyr kernel can build with the minimal libc.

In addition, any functions from the above list that are not part of the ISO/IEC 9899:2011 standard must be implemented by the *common libc* to ensure their availability across multiple C standard libraries.

Introducing new C standard library functions to the Zephyr kernel is allowed with justification given that the above requirements are satisfied.

Note that the use of the functions listed above are subject to secure and safe coding practices and it should not be assumed that their use in the Zephyr kernel is unconditionally permitted by being listed in this rule.

The "Zephyr kernel" in this context consists of the following components:

- Kernel (`kernel`)
- OS Library (`lib/os`)
- Architecture Port (`arch`)
- Logging Subsystem (`subsys/logging`)

Rationale Zephyr kernel must be able to build with the *minimal libc*, a limited C standard library implementation that is part of the Zephyr RTOS and maintained by the Zephyr Project, to allow self-contained testing and verification of the kernel and core OS services.

In order to ensure that the Zephyr kernel can build with the minimal libc, it is necessary to restrict the use of the C standard library functions and macros in the Zephyr kernel to the functions and macros that are available as part of the minimal libc.

Rule A.5: C Standard Library Usage Restrictions in Zephyr Codebase

Severity Required

Description The use of the C standard library functions and macros in the Zephyr codebase shall be limited to the functions, excluding the Annex K "Bounds-checking interfaces", from the ISO/IEC 9899:2011 standard, also known as C11, unless exempted by this rule.

The "Zephyr codebase" in this context refers to all embedded source code files committed to the main Zephyr repository, except the Zephyr kernel as defined by the *Rule A.4: C Standard Library Usage Restrictions in Zephyr Kernel*. With embedded source code we refer to code which is meant to be executed in embedded targets, and therefore excludes host tooling, and code specific for the native test targets.

The following non-ISO 9899:2011, hereinafter referred to as non-standard, functions and macros are exempt from this rule and allowed to be used in the Zephyr codebase:

Table 3: List of allowed non-standard libc functions

Function	Source
gmtime_r()	POSIX.1-2001
strnlen()	POSIX.1-2008
strtok_r()	POSIX.1-2001

All non-standard functions and macros listed above must be implemented by the *common libc* in order to make sure that these functions can be made available when using a C standard library that does not implement these functions.

Adding a new non-standard function from common C standard libraries to the above list is allowed with justification, given that the above requirement is satisfied. However, when there exists a standard function that is functionally equivalent, the standard function shall be used.

Rationale Some C standard libraries, such as Newlib and Picolibc, include additional functions and macros that are defined by the standards and de-facto standards that extend the ISO C standard (e.g. POSIX, Linux).

The ISO/IEC 9899:2011 standard does not require C compiler toolchains to include the support for these non-standard functions, and therefore using these functions can lead to compatibility issues with the third-party toolchains that come with their own C standard libraries.

8.1.3 Proposals and RFCs

Many changes, including bug fixes and documentation improvements can be implemented and reviewed via the normal GitHub pull request workflow.

Many changes however are "substantial" and need to go through a design process and produce a consensus among the project stakeholders.

The "RFC" (request for comments) process is intended to provide a consistent and controlled path for new features to enter the project.

Contributors and project stakeholders should consider using this process if they intend to make "substantial" changes to Zephyr or its documentation. Some examples that would benefit from an RFC are:

- A new feature that creates new API surface area, and would require a feature flag if introduced.
- The modification of an existing stable API.
- The removal of features that already shipped as part of Zephyr.
- The introduction of new idiomatic usage or conventions, even if they do not include code changes to Zephyr itself.

The RFC process is a great opportunity to get more eyeballs on proposals coming from contributors before it becomes a part of Zephyr. Quite often, even proposals that seem "obvious" can be significantly improved once a wider group of interested people have a chance to weigh in.

The RFC process can also be helpful to encourage discussions about a proposed feature as it is being designed, and incorporate important constraints into the design while it's easier to change, before the design has been fully implemented.

Some changes do not require an RFC:

- Rephrasing, reorganizing or refactoring
- Addition or removal of warnings
- Addition of new boards, SoCs or drivers to existing subsystems
- ...

The process in itself consists in creating a GitHub issue with the *RFC label* that documents the proposal thoroughly. There is an RFC template included in the main Zephyr GitHub repository that serves as a guideline to write a new RFC.

As with Pull Requests, RFCs might require discussion in the context of one of the Zephyr meetings in order to move it forward in cases where there is either disagreement or not enough voiced opinions in order to proceed. Make sure to either label it appropriately or include it in the corresponding GitHub project in order for it to be examined during the next meeting.

8.1.4 Contributor Expectations

The Zephyr project encourages *contributors* to submit changes as smaller pull requests. Smaller pull requests (PRs) have the following benefits:

- Reviewed more quickly and reviewed more thoroughly. It's easier for reviewers to set aside a few minutes to review smaller changes several times than it is to allocate large blocks of time to review a large PR.
- Less wasted work if reviewers or maintainers reject the direction of the change.
- Easier to rebase and merge. Smaller PRs are less likely to conflict with other changes in the tree.
- Easier to revert if the PR breaks functionality.

Note

This page does not apply to draft PRs which can have any size, any number of commits and any combination of smaller PRs for testing and preview purposes. Draft PRs have no review expectation and PRs created as drafts from the start do not notify anyone by default.

Defining Smaller PRs

- Smaller PRs should encompass one self-contained logical change.
- When adding a new large feature or API, the PR should address only one part of the feature. In this case create an *RFC proposal* to describe the additional parts of the feature for reviewers.
- PRs should include tests or samples under the following conditions:
  - Adding new features or functionality.
  - Modifying a feature, especially for API behavior contract changes.
  - Fixing a hardware agnostic bug. The test should fail without the bug fixed and pass with the fix applied.
- PRs must update any documentation affected by a functional code change.
- If introducing a new API, the PR must include an example usage of the API. This provides context to the reviewer and prevents submitting PRs with unused APIs.

Multiple Commits on a Single PR

Contributors are further encouraged to break up PRs into multiple commits. Keep in mind each commit in the PR must still build cleanly and pass all the CI tests.

For example, when introducing an extension to an API, contributors can break up the PR into multiple commits targeting these specific changes:

1. Introduce the new APIs, including shared devicetree bindings
2. Update driver implementation X, with driver specific devicetree bindings
3. Update driver implementation Y
4. Add tests for the new API
5. Add a sample using the API
6. Update the documentation

Large Changes

Large changes to the Zephyr project must submit an *RFC proposal* describing the full scope of change and future work. The RFC proposal provides the required context to reviewers, but allows for smaller, incremental, PRs to get reviewed and merged into the project. The RFC should also define the minimum viable implementation.

Changes which require an RFC proposal include:

- Submitting a new feature.
- Submitting a new API.
- *Treewide Changes*.
- Other large changes that can benefit from the RFC proposal process.

Maintainers have the discretion to request that contributors create an RFC for PRs that are too large or complicated.

PR Requirements

- Each commit in the PR must provide a commit message following the *Commit Message Guidelines*.
- The PR description must include a summary of the changes and their rationales.
- All files in the PR must comply with *Licensing Requirements*.
- Follow the Zephyr *Coding Style* and *Coding Guidelines*.
- PRs must pass all CI checks. This is a requirement to merge the PR. Contributors may mark a PR as draft and explicitly request reviewers to provide early feedback, even with failing CI checks.
- When breaking a PR into multiple commits, each commit must build cleanly. The CI system does not enforce this policy, so it is the PR author's responsibility to verify.
- When major new functionality is added, tests for the new functionality shall be added to the automated test suite. All API functions should have test cases and there should be tests for the behavior contracts of the API. Maintainers and reviewers have the discretion to determine if the provided tests are sufficient. The examples below demonstrate best practices on how to test APIs effectively.
  - Kernel timer tests provide around 85% test coverage for the kernel timer , measured by lines of code.
  - Emulators for off-chip peripherals are an effective way to test driver APIs. The fuel gauge tests use the smart battery emulator , providing test coverage for the fuel gauge API and the smart battery driver .
  - Code coverage reports for the Zephyr project are available on Codecov.
- Incompatible changes to APIs must also update the release notes for the next release detailing the change. APIs marked as experimental are excluded from this requirement.
- Changes to APIs must increment the API version number according to the API version rules.
- PRs must also satisfy all *Merge Criteria* before a member of the release engineering team merges the PR into the zephyr tree.

Maintainers may request that contributors break up a PR into smaller PRs and may request that they create an *RFC proposal*.

Workflow Suggestions That Help Reviewers

- Unless they applied the reviewer's recommendation exactly, authors must not resolve and hide comments, they must let the initial reviewer do it. The Zephyr project does not require all comments to be resolved before merge. Leaving some completed discussions open can sometimes be useful to understand the greater picture.
- Respond to comments using the "Start Review" and "Add Review" green buttons in the "Files changed" view. This allows responding to multiple comments and publishing the responses in bulk. This reduces the number of emails sent to reviewers.
- As GitHub does not implement `git range-diff`, try to minimize rebases in the middle of a review. If a rebase is required, push this as a separate update with no other changes since the last push of the PR. When pushing a rebase only, add a comment to the PR indicating which commit is the rebase.

PR Review Escalation The Zephyr community is a diverse group of individuals, with different levels of commitment and priorities. As such, reviewers and maintainers may not get to a PR right away.

The Zephyr Dev Meeting performs a triage of PRs missing reviewer approval, following this process:

1. Identify and update PRs missing an Assignee.
2. Identify PRs without any comments or reviews, ping the PR Assignee to start a review or assign to a different maintainer.
3. For PRs that have otherwise stalled, the Zephyr Dev Meeting pings the Assignee and any reviewers that have left comments on the PR.

Contributors may escalate PRs outside of the Zephyr Dev Meeting triage process as follows:

- After 1 week of inactivity, ping the Assignee or reviewers on the PR by adding a comment to the PR.
- After 2 weeks of inactivity, post a message on the #pr-help channel on Discord linking to the PR.
- After 2 weeks of inactivity, add the dev-review label to the PR. This explicitly adds the PR to the agenda for the next Zephyr Dev Meeting independent of the triage process. Not all contributors have the required privileges to add labels to PRs, in this case the contributor should request help on Discord or send an email to the Zephyr devel mailing list.

Note that for new PRs, contributors should generally wait for at least one Zephyr Dev Meeting before escalating the PR themselves.

PR Technical Escalation In cases where a contributor objects to change requests from reviewers, Zephyr defines the following escalation process for resolving technical disagreements.

Before escalation of technical disagreements, follow the steps below:

- Resolve in the PR among assignee, maintainers and reviewer.
  - Assignee to act as moderator if applicable.
- Optionally resolve in the next Zephyr Dev Meeting meeting with more Maintainers and project stakeholders.
  - The involved parties and the Assignee to be present when the issue is discussed.
- If no progress is made, the assignee (maintainer) has the right to dismiss stale, unrelated or irrelevant change requests by reviewers giving the reviewers a minimum of 1 business day to respond and revisit their initial change requests or start the escalation process.

The assignee has the responsibility to document the reasoning for dismissing any reviews in the PR and should notify the reviewer about their review being dismissed.

To give the reviewers time to respond and escalate, the assignee is expected to block the PR from being merged either by not approving the PR or by setting the *DNM* label.

Escalation can be triggered by any party participating in the review process (assignee, reviewers or the original author of the change) following the steps below:

- Escalate to the Architecture Working Group by adding the *Architecture Review* label on the PR. Beside the weekly meeting where such escalations are processed, the Architecture Working Group shall facilitate an offline review of the escalation if requested, especially if any of the parties can't attend the meeting.
- If all avenues of resolution and escalation have failed, assignees can escalate to the TSC and get a binding resolution in the TSC by adding the *TSC* label on the PR.
- The Assignee is expected to ensure the resolution of the escalation and the outcome is documented in the related pull request or issues on Github.

Reviewer Expectations

- Be respectful when commenting on PRs. Refer to the Zephyr Code of Conduct for more details.
- The Zephyr Project recognizes that reviewers and maintainers have limited bandwidth. As a reviewer, prioritize review requests in the following order:
  1. PRs related to items in the Zephyr Release Plan or those targeting the next release during the stabilization period (after RC1).
  2. PRs where the reviewer has requested blocking changes.
  3. PRs assigned to the reviewer as the area maintainer.
  4. All other PRs.
- Reviewers shall strive to advance the PR to a mergeable state with their feedback and engagement with the PR author.
- Try to provide feedback on the entire PR in one shot. This provides the contributor an opportunity to address all comments in the next PR update.
- Partial reviews are permitted, but the reviewer must add a comment indicating what portion of the PR they reviewed. Examples of useful partial reviews include:
  - Domain specific reviews (e.g. Devicetree).
  - Code style changes that impact the readability of the PR.
  - Reviewing commits separately when the requested changes cascade into the later commits.
- Avoid increasing scope of the PR by requesting new features, especially when there is a corresponding *RFC* associated with the PR. Instead, reviewers should add suggestions as a comment to the *RFC*. This also encourages more collaboration as it is easier for multiple contributors to work on a feature once the minimum implementation has merged.
- When using the "Request Changes" option, mark trivial, non-functional, requests as "Non-blocking" in the comment. Reviewers should approve PRs once only non-blocking changes remain. The PR author has discretion as to whether they address all non-blocking comments. PR authors should acknowledge every review comment in some way, even if it's just with an emoticon.

- Reviewers shall be *clear* and *concise* what changes they are requesting when the "Request Changes" option is used. Requested changes shall be in the scope of the PR in question and following the contribution and style guidelines of the project.
- Reviewers shall not close a PR due to technical or structural disagreement. If requested changes cannot be resolved within the review process, the *PR Technical Escalation* path shall be used for any potential resolution path, which may include closing the PR.

Contribution Guidelines
Learn about the overall process and guidelines for contributing to the Zephyr project.

This page is a mandatory read for first-time contributors as it contains important information on how to ensure your contribution can be considered for inclusion in the project and potentially merged.

Contributor Expectations
This document is another mandatory read that describes the expected behavior of *all* contributors to the project.

Coding Guidelines
Code contributions are expected to follow a set of coding guidelines to ensure consistency and readability across the code base.

This page describes these guidelines and introduces important considerations regarding the use of *inclusive language*.

Proposals and RFCs
Learn when and how to submit RFCs (Request for Comments) for new features and changes to the project.

8.2 Documentation

The Zephyr project thrives on good documentation. Whether it is as part of a code contribution or as a standalone effort, contributing documentation is particularly valuable to the project.

8.2.1 Documentation Guidelines

Note

For instructions on building the documentation, see *Documentation Generation*.

Zephyr Project content is written using the reStructuredText markup language (.rst file extension) with Sphinx extensions, and processed using Sphinx to create a formatted standalone website. Developers can view this content either in its raw form as .rst markup files, or (with Sphinx installed) they can *build the documentation* locally to generate the documentation in HTML or PDF format. The HTML content can then be viewed using a web browser. This same .rst content is served by the Zephyr documentation website.

You can read details about reStructuredText and about Sphinx extensions from their respective websites.

This document provides a quick reference for commonly used reST and Sphinx-defined directives and roles used to create the documentation you're reading.

Headings

While reST allows use of both and overline and matching underline to indicate a heading, we only use an underline indicator for headings.

- Document title (h1) use "#" for the underline character
- First section heading level (h2) use "*"
- Second section heading level (h3) use "="
- Third section heading level (h4) use "-"

The heading underline must be at least as long as the title it's under.

For example:

```
This is a title heading
#######################

some content goes here

First section heading
*********************
```

Content Highlighting

Some common reST inline markup samples:

- one asterisk: `*text*` for emphasis (*italics*),
- two asterisks: `**text**` for strong emphasis (**boldface**), and
- two backquotes: ``` ``text`` ``` for `inline code` samples.

If asterisks or backquotes appear in running text and could be confused with inline markup delimiters, you can eliminate the confusion by adding a backslash (\) before it.

Lists

For bullet lists, place an asterisk (*) or hyphen (-) at the start of a paragraph and indent continuation lines with two spaces.

The first item in a list (or sublist) must have a blank line before it and should be indented at the same level as the preceding paragraph (and not indented itself).

For numbered lists start with a 1. or a. for example, and continue with autonumbering by using a # sign. Indent continuation lines with three spaces:

```
* This is a bulleted list.
* It has two items, the second
  item and has more than one line of reST text.  Additional lines
  are indented to the first character of the
  text of the bullet list.

1. This is a new numbered list. If the wasn't a blank line before it,
   it would be a continuation of the previous list (or paragraph).
#. It has two items too.

a. This is a numbered list using alphabetic list headings
#. It has three items (and uses autonumbering for the rest of the list)
#. Here's the third item
```

(continues on next page)

(continued from previous page)

```
#. This is an autonumbered list (default is to use numbers starting
   with 1).

   #. This is a second-level list under the first item (also
      autonumbered).  Notice the indenting.
   #. And a second item in the nested list.
#. And a second item back in the containing list.  No blank line
   needed, but it wouldn't hurt for readability.
```

Definition lists (with a term and its definition) are a convenient way to document a word or phrase with an explanation. For example this reST content:

```
The Makefile has targets that include:

html
   Build the HTML output for the project

clean
   Remove all generated output, restoring the folders to a
   clean state.
```

Would be rendered as:

The Makefile has targets that include:

html
: Build the HTML output for the project

clean
: Remove all generated output, restoring the folders to a clean state.

Multi-column lists

If you have a long bullet list of items, where each item is short, you can indicate the list items should be rendered in multiple columns with a special `.. rst-class:: rst-columns` directive. The directive will apply to the next non-comment element (e.g., paragraph), or to content indented under the directive. For example, this unordered list:

```
.. rst-class:: rst-columns

* A list of
* short items
* that should be
* displayed
* horizontally
* so it doesn't
* use up so much
* space on
* the page
```

would be rendered as:

- A list of
- short items
- that should be
- displayed
- horizontally
- so it doesn't

- use up so much
- space on
- the page

A maximum of three columns will be displayed, and change based on the available width of the display window, reducing to one column on narrow (phone) screens if necessary. We've deprecated use of the `hlist` directive because it misbehaves on smaller screens.

Tables

There are a few ways to create tables, each with their limitations or quirks. Grid tables offer the most capability for defining merged rows and columns, but are hard to maintain:

```
+------------------------+------------+----------+----------+
| Header row, column 1   | Header 2   | Header 3 | Header 4 |
| (header rows optional) |            |          |          |
+========================+============+==========+==========+
| body row 1, column 1   | column 2   | column 3 | column 4 |
+------------------------+------------+----------+----------+
| body row 2             | ...        | ...      | you can  |
+------------------------+------------+----------+ easily   +
| body row 3 with a two column span   | ...      | span     |
+------------------------+------------+----------+ rows     +
| body row 4             | ...        | ...      | too      |
+------------------------+------------+----------+----------+
```

This example would render as:

Header row, column 1 (header rows optional)	Header 2	Header 3	Header 4
body row 1, column 1	column 2	column 3	column 4
body row 2	...	...	you can easily span rows too
body row 3 with a two column span		...	
body row 4	...	...	

List tables are much easier to maintain, but don't support row or column spans:

```
.. list-table:: Table title
   :widths: 15 20 40
   :header-rows: 1

   * - Heading 1
     - Heading 2
     - Heading 3
   * - body row 1, column 1
     - body row 1, column 2
     - body row 1, column 3
   * - body row 2, column 1
     - body row 2, column 2
     - body row 2, column 3
```

This example would render as:

Table 4: Table title

Heading 1	Heading 2	Heading 3
body row 1, column 1	body row 1, column 2	body row 1, column 3
body row 2, column 1	body row 2, column 2	body row 2, column 3

The `:widths:` parameter lets you define relative column widths. The default is equal column widths. If you have a three-column table and you want the first column to be half as wide as the other two equal-width columns, you can specify `:widths: 1 2 2`. If you'd like the browser to set the column widths automatically based on the column contents, you can use `:widths: auto`.

File names and Commands

Sphinx extends reST by supporting additional inline markup elements (called "roles") used to tag text with special meanings and allow style output formatting. (You can refer to the Sphinx Inline Markup documentation for the full list).

For example, there are roles for marking `filenames` (`:file:\`name\``) and command names such as **`make`** (`:command:\`make\``). You can also use the "inline code" markup (double backticks) to indicate a `filename`.

For references to files that are in the Zephyr GitHub tree, a special role can be used that creates a hyperlink to that file. For example a reference to the reST file used to create this document can be generated using `:zephyr_file:\`doc/contribute/documentation/guidelines.rst\`` that will show up as doc/contribute/documentation/guidelines.rst, a link to the "blob" file in the github repo. There's also a `:zephyr_raw:\`doc/contribute/documentation/guidelines.rst\`` role that will link to the "raw" content, doc/contribute/documentation/guidelines.rst. (You can click on these links to see the difference.)

Internal Cross-Reference Linking

Traditional ReST links are only supported within the current file using the notation:

```
Refer to the `internal-linking`_ page
```

which renders as,

> Refer to the *internal-linking* page

Note the use of a trailing underscore to indicate an outbound link. In this example, the label was added immediately before a heading, so the text that's displayed is the heading text itself. You can change the text that's displayed as the link writing this as:

```
Refer to the `show this text instead <internal-linking_>`_ page
```

which renders as,

> Refer to the *show this text instead* page

External Cross-Reference Linking

With Sphinx's help, we can create link-references to any tagged text within the Zephyr Project documentation.

Target locations in a document are defined with a label directive:

```
.. _my label name:

Heading
=======
```

Note the leading underscore indicating an inbound link. The content immediately following this label must be a heading, and is the target for a :ref:`my label name` reference from anywhere within the Zephyr documentation. The heading text is shown when referencing this label. You can also change the text that's displayed for this link, such as:

```
:ref:`some other text <my label name>`
```

To enable easy cross-page linking within the site, each file should have a reference label before its title so it can be referenced from another file. These reference labels must be unique across the whole site, so generic names such as "samples" should be avoided. For example the top of this document's .rst file is:

```
.. _doc_guidelines:

Documentation Guidelines for the Zephyr Project
###############################################
```

Other .rst documents can link to this document using the :ref:`doc_guidelines` tag and it will show up as *Documentation Guidelines*. This type of internal cross reference works across multiple files, and the link text is obtained from the document source so if the title changes, the link text will update as well.

You can also define links to any URL and then reference it in your document. For example, with this label definition in the document:

```
.. _Zephyr Wikipedia Page:
   https://en.wikipedia.org/wiki/Zephyr_(operating_system)
```

you can reference it with:

```
Read the `Zephyr Wikipedia Page`_ for more information about the
project.
```

'any' links

Within the Zephyr project, we've defined the default role to be "any", meaning if you just write a phrase in back-ticks, e.g., `doc_guidelines`, Sphinx will search through all domains looking for something called doc_guidelines to link to. In this case it will find the label at the top of this document, and link to *doc_guidelines*. This can be useful for linking to doxygen-generated links for function names and such, but will cause a warning such as:

```
WARNING: 'any' reference target not found: doc_giudelines
```

if you misspelled `doc_guidelines` as `doc_giudelines`.

Non-ASCII Characters

You can insert non-ASCII characters such as a Trademark symbol (™), by using the notation |trade|. Available replacement names are defined in an include file used during the Sphinx processing of the reST files. The names of these replacement characters are the same as used in HTML entities used to insert characters in HTML, e.g., ™ and are defined in the file sphinx_build/substitutions.txt as listed here:

```
.. |br|     raw:: html        .. force a line break in HTML output (blank lines needed here)

   <br />

.. |p|     raw:: html        .. force a blank line in HTML output (blank lines needed here)

   <p></p>

.. These are replacement strings for non-ASCII characters used within the project
   using the same name as the html entity names (e.g., &copy;) for that character

.. |copy|   unicode:: U+000A9 .. COPYRIGHT SIGN
   :ltrim:
.. |trade|  unicode:: U+02122 .. TRADEMARK SIGN
   :ltrim:
.. |reg|    unicode:: U+000AE .. REGISTERED TRADEMARK SIGN
   :ltrim:
.. |deg|    unicode:: U+000B0 .. DEGREE SIGN
   :ltrim:
.. |plusminus|  unicode:: U+000B1 .. PLUS-MINUS SIGN
   :rtrim:
.. |micro|  unicode:: U+000B5 .. MICRO SIGN
   :rtrim:
.. |sup2|  unicode:: U+00B2 .. SUPERSCRIPT TWO
   :ltrim:
```

We've kept the substitutions list small but others can be added as needed by submitting a change to the `substitutions.txt` file.

Code and Command Examples

Use the reST `code-block` directive to create a highlighted block of fixed-width text, typically used for showing formatted code or console commands and output. Smart syntax highlighting is also supported (using the Pygments package). You can also directly specify the highlighting language. For example:

```
.. code-block:: c

   struct k_object {
      char *name;
      uint8_t perms[CONFIG_MAX_THREAD_BYTES];
      uint8_t type;
      uint8_t flags;
      uint32_t data;
   } __packed;
```

Note the blank line between the `code-block` directive and the first line of the code-block body, and the body content is indented three spaces (to the first non-white space of the directive name).

This would be rendered as:

```
struct k_object {
   char *name;
   uint8_t perms[CONFIG_MAX_THREAD_BYTES];
   uint8_t type;
   uint8_t flags;
   uint32_t data;
} __packed;
```

Other languages are of course supported (see languages supported by Pygments), and in particular, you are encouraged to make use of the following when appropriate:

- `c` for C code
- `cpp` for C++ code
- `python` for Python code
- `console` for console output, i.e. interactive shell sessions where commands are prefixed by a prompt (ex. `$` for Linux, or `uart:~$` for Zephyr's shell), and where the output is also shown. The commands will be highlighted, and the output will not. What's more, copying code block using the "copy" button will automatically copy just the commands, excluding the prompt and the outputs of the commands.
- `shell` or `bash` for shell commands. Both languages get highlighted the same but you may use `bash` for conveying that the commands are bash-specific, and `shell` for generic shell commands.

Note

Do not use `bash` or `shell` if your code block includes a prompt, use `console` instead.

Reciprocally, do not use `console` if your code block does not include a prompt and is not showcasing an interactive session with command(s) and their output.

Table 5: When to use `bash`/`shell` vs. `console`

Use case	code-block snippet	Expected output

One or several commands, no output	`.. code-block:: shell`
	
	`echo "Hello World!"`
		`echo "Hello World!"`

An interactive shell session with command(s) and their output	`.. code-block:: console`
	
	`$ echo "Hello World!"`
	`Hello World!`
		`$ echo "Hello World!"`
		`Hello World!`

An interactive Zephyr shell session, with commands and their outputs	`.. code-block:: console`
	
	`uart:~$ version`
	`Zephyr version 3.5.99`
	`uart:~$ kernel uptime`
	`Uptime: 20970 ms`
		`uart:~$ version`
		`Zephyr version 3.5.99`
		`uart:~$ kernel uptime`
		`Uptime: 20970 ms`

- `bat` for Windows batch files
- `cfg` for config files with "KEY=value" entries (ex. Kconfig `.conf` files)
- `cmake` for CMake
- `devicetree` for Devicetree
- `kconfig` for Kconfig
- `yaml` for YAML
- `rst` for reStructuredText

When no language is specified, the language is set to `none` and the code block is not highlighted. You may also use `none` explicitly to achieve the same result; for example:

```
.. code-block:: none

   This would be a block of text styled with a background
   and box, but with no syntax highlighting.
```

Would display as:

```
This would be a block of text styled with a background
and box, but with no syntax highlighting.
```

There's a shorthand for writing code blocks too: end the introductory paragraph with a double colon (`::`) and indent the code block content that follows it by three spaces. On output, only one colon will be shown. The code block will have no highlighting (i.e. `none`). You may however use the `.. highlight::` directive to customize the default language used in your document (see for example how this is done at the beginning of this very document).

Images

Images are included in documentation by using an image directive:

```
.. image:: ../../../../images/doc-gen-flow.png
   :align: center
   :alt: alt text for the image
```

or if you'd like to add an image caption, use:

```
.. figure:: ../../../../images/doc-gen-flow.png
   :alt: image description

   Caption for the figure
```

The file name specified is relative to the document source file, and we recommend putting images into an `images` folder where the document source is found. The usual image formats handled by a web browser are supported: JPEG, PNG, GIF, and SVG. Keep the image size only as large as needed, generally at least 500 px wide but no more than 1000 px, and no more than 250 KB unless a particularly large image is needed for clarity.

Tabs, spaces, and indenting

Indenting is significant in reST file content, and using spaces is preferred. Extra indenting can (unintentionally) change the way content is rendered too. For lists and directives, indent the content text to the first non-white space in the preceding line. For example:

```
* List item that spans multiple lines of text
  showing where to indent the continuation line.

1. And for numbered list items, the continuation
   line should align with the text of the line above.

.. code-block::

   The text within a directive block should align with the
   first character of the directive name.
```

Refer to the Zephyr *Coding Style* for additional requirements.

zephyr-app-commands Directive

This is a Zephyr directive for generating consistent documentation of the shell commands needed to manage (build, flash, etc.) an application.

For example, to generate commands to build samples/hello_world for qemu_x86 use:

```
.. zephyr-app-commands::
   :zephyr-app: samples/hello_world
   :board: qemu_x86
   :goals: build
```

Directive options:

:tool:
which tool to use. Valid options are currently 'cmake', 'west' and 'all'. The default is 'west'.

:app:
path to the application to build.

:zephyr-app:
path to the application to build, this is an app present in the upstream zephyr repository. Mutually exclusive with :app:.

:cd-into:
if set, build instructions are given from within the :app: folder, instead of outside of it.

:generator:
which build system to generate. Valid options are currently 'ninja' and 'make'. The default is 'ninja'. This option is not case sensitive.

:host-os:
which host OS the instructions are for. Valid options are 'unix', 'win' and 'all'. The default is 'all'.

:board:
if set, the application build will target the given board.

:shield:
if set, the application build will target the given shield. Multiple shields can be provided in a comma separated list.

:conf:
if set, the application build will use the given configuration file. If multiple conf files are provided, enclose the space-separated list of files with quotes, e.g., "a.conf b.conf".

:gen-args:
if set, additional arguments to the CMake invocation

:build-args:
if set, additional arguments to the build invocation

:snippets:
if set, indicates the application should be compiled with the listed snippets. Multiple snippets can be provided in a comma separated list.

:build-dir:
if set, the application build directory will *APPEND* this (relative, Unix-separated) path to the standard build directory. This is mostly useful for distinguishing builds for one application within a single page.

:build-dir-fmt:
if set, assume that "west config build.dir-fmt" has been set to this path. Exclusive with 'build-dir' and depends on 'tool=west'.

:goals:
a whitespace-separated list of what to do with the app (in 'build', 'flash', 'debug', 'debugserver', 'run'). Commands to accomplish these tasks will be generated in the right order.

:maybe-skip-config:
if set, this indicates the reader may have already created a build directory and changed there, and will tweak the text to note that doing so again is not necessary.

:compact:
if set, the generated output is a single code block with no additional comment lines

:west-args:
if set, additional arguments to the west invocation (ignored for CMake)

:flash-args:
if set, additional arguments to the flash invocation

For example, the `.. zephyr-app-commands` listed above would render like this in the generated HTML output:

```
# From the root of the zephyr repository
west build -b qemu_x86 samples/hello_world
```

Alternative Tabbed Content

As introduced in the *Getting Started Guide*, you can provide alternative content to the reader via a tabbed interface. When the reader clicks on a tab, the content for that tab is displayed, for example:

```
.. tabs::

   .. tab:: Apples

      Apples are green, or sometimes red.

   .. tab:: Pears

      Pears are green.

   .. tab:: Oranges

      Oranges are orange.
```

will display as:

Apples

Apples are green, or sometimes red.

Pears

Pears are green.

Oranges

Oranges are orange.

Tabs can also be grouped, so that changing the current tab in one area changes all tabs with the same name throughout the page. For example:

Linux

Linux Line 1

macOS

macOS Line 1

Windows

Windows Line 1

Linux

Linux Line 2

macOS

macOS Line 2

Windows

Windows Line 2

In this latter case, we're using `.. group-tab::` instead of simply `.. tab::`. Under the hood, we're using the sphinx-tabs extension that's included in the Zephyr setup. Within a tab, you can have most any content *other than a heading* (code-blocks, ordered and unordered lists, pictures, paragraphs, and such). You can read more about sphinx-tabs from the link above.

8.2.2 Documentation Generation

These instructions will walk you through generating the Zephyr Project's documentation on your local system using the same documentation sources as we use to create the online documentation found at https://docs.zephyrproject.org

Documentation overview

Zephyr Project content is written using the reStructuredText markup language (.rst file extension) with Sphinx extensions, and processed using Sphinx to create a formatted stand-alone website. Developers can view this content either in its raw form as .rst markup files, or you can generate the HTML content and view it with a web browser directly on your workstation. This same .rst content is also fed into the Zephyr Project's public website documentation area (with a different theme applied).

You can read details about reStructuredText, and Sphinx from their respective websites.

The project's documentation contains the following items:

- ReStructuredText source files used to generate documentation found at the https://docs.zephyrproject.org website. Most of the reStructuredText sources are found in the `/doc` directory, but others are stored within the code source tree near their specific component (such as `/samples` and `/boards`)
- Doxygen-generated material used to create all API-specific documents also found at https://docs.zephyrproject.org
- Script-generated material for kernel configuration options based on Kconfig files found in the source code tree

The reStructuredText files are processed by the Sphinx documentation system, and make use of the breathe extension for including the doxygen-generated API material. Additional tools are required to generate the documentation locally, as described in the following sections.

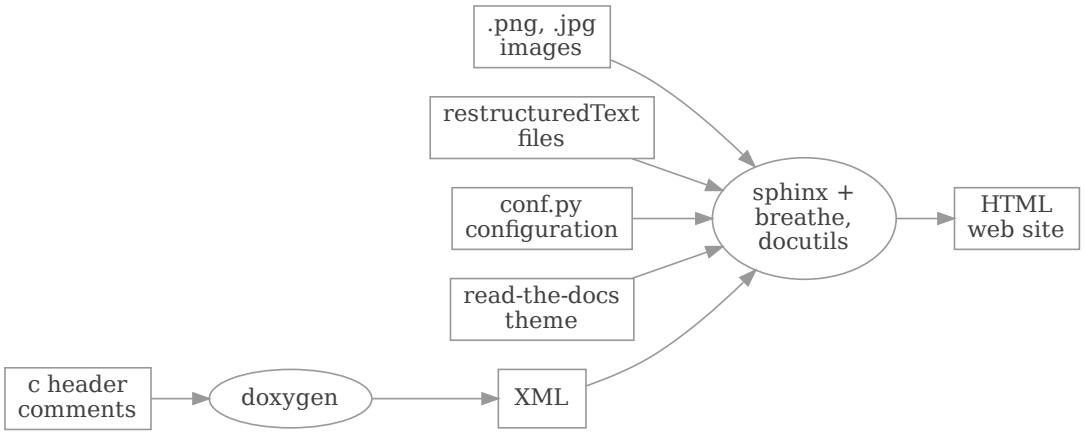


Fig. 1: Schematic of the documentation build process

Installing the documentation processors

Our documentation processing has been tested to run with:

- Doxygen version 1.8.13
- Graphviz 2.43
- Latexmk version 4.56
- All Python dependencies listed in the repository file `doc/requirements.txt`

In order to install the documentation tools, first install Zephyr as described in *Getting Started Guide*. Then install additional tools that are only required to generate the documentation, as described below:

Linux

Common to all Linux installations, install the Python dependencies required to build the documentation:

```
pip install -U -r ~/zephyrproject/zephyr/doc/requirements.txt
```

On Ubuntu Linux:

```
sudo apt-get install --no-install-recommends doxygen graphviz librsvg2-bin \
texlive-latex-base texlive-latex-extra latexmk texlive-fonts-recommended imagemagick
```

On Fedora Linux:

```
sudo dnf install doxygen graphviz texlive-latex latexmk \
texlive-collection-fontsrecommended librsvg2-tools ImageMagick
```

On Clear Linux:

```
sudo swupd bundle-add texlive graphviz ImageMagick
```

On Arch Linux:

```
sudo pacman -S graphviz doxygen librsvg texlive-core texlive-bin \
texlive-latexextra texlive-fontsextra imagemagick
```

macOS

Install the Python dependencies required to build the documentation:

```
pip install -U -r ~/zephyrproject/zephyr/doc/requirements.txt
```

Use `brew` and `tlmgr` to install the tools:

```
brew install doxygen graphviz mactex librsvg imagemagick
tlmgr install latexmk
tlmgr install collection-fontsrecommended
```

Windows

Install the Python dependencies required to build the documentation:

```
pip install -U -r %HOMEPATH$\zephyrproject\zephyr\doc\requirements.txt
```

Open a `cmd.exe` window as **Administrator** and run the following command:

```
choco install doxygen.install graphviz strawberryperl miktex rsvg-convert imagemagick
```

Note

On Windows, the Sphinx executable `sphinx-build.exe` is placed in the `Scripts` folder of your Python installation path. Depending on how you have installed Python, you might need to add this folder to your `PATH` environment variable. Follow the instructions in Windows Python Path to add those if needed.

Documentation presentation theme

Sphinx supports easy customization of the generated documentation appearance through the use of themes. Replace the theme files and do another `make html` and the output layout and style is changed. The `read-the-docs` theme is installed as part of the *Get Zephyr and install Python dependencies* step you took in the getting started guide.

Running the documentation processors

The `/doc` directory in your cloned copy of the Zephyr project git repo has all the .rst source files, extra tools, and Makefile for generating a local copy of the Zephyr project's technical documentation. Assuming the local Zephyr project copy is in a folder `zephyr` in your home folder, here are the commands to generate the html content locally:

```
# On Linux/macOS
cd ~/zephyr/doc
# On Windows
cd %userprofile%\zephyr\doc

# Use cmake to configure a Ninja-based build system:
cmake -GNinja -B_build .

# Enter the build directory
cd _build

# To generate HTML output, run ninja on the generated build system:
ninja html
# If you modify or add .rst files, run ninja again:
ninja html
```

(continues on next page)

(continued from previous page)

```
# To generate PDF output, run ninja on the generated build system:
ninja pdf
```

Warning

The documentation build system creates copies in the build directory of every .rst file used to generate the documentation, along with dependencies referenced by those .rst files.

This means that Sphinx warnings and errors refer to the **copies**, and **not the version-controlled original files in Zephyr**. Be careful to make sure you don't accidentally edit the copy of the file in an error message, as these changes will not be saved.

Depending on your development system, it will take up to 15 minutes to collect and generate the HTML content. When done, you can view the HTML output with your browser started at `doc/_build/html/index.html` and if generated, the PDF file is available at `doc/_build/latex/zephyr.pdf`.

If you want to build the documentation from scratch just delete the contents of the build folder and run `cmake` and then `ninja` again.

Note

If you add or remove a file from the documentation, you need to re-run CMake.

On Unix platforms a convenience doc/Makefile can be used to build the documentation directly from there:

```
cd ~/zephyr/doc

# To generate HTML output
make html

# To generate PDF output
make pdf
```

Filtering expected warnings

There are some known issues with Sphinx/Breathe that generate Sphinx warnings even though the input is valid C code. While these issues are being considered for fixing we have created a Sphinx extension that allows to filter them out based on a set of regular expressions. The extension is named `zephyr.warnings_filter` and it is located at `doc/_extensions/zephyr/warnings_filter.py`. The warnings to be filtered out can be added to the `doc/known-warnings.txt` file.

The most common warning reported by Sphinx/Breathe is related to duplicate C declarations. This warning may be caused by different Sphinx/Breathe issues:

- Multiple declarations of the same object are not supported
- Different objects (e.g. a struct and a function) can not share the same name
- Nested elements (e.g. in a struct or union) can not share the same name

Developer-mode Document Building

When making and testing major changes to the documentation, we provide an option to temporarily stub-out the auto-generated Devicetree bindings documentation so the doc build process runs faster.

To enable this mode, set the following option when invoking cmake:

```
-DDT_TURBO_MODE=1
```

or invoke make with the following target:

```
cd ~/zephyr

# To generate HTML output without detailed Kconfig
make html-fast
```

Viewing generated documentation locally

The generated HTML documentation can be hosted locally with python for viewing with a web browser:

```
$ python3 -m http.server -d _build/html
```

Note

WSL2 users may need to explicitly bind the address to `127.0.0.1` in order to be accessible from the host machine:

```
$ python3 -m http.server -d _build/html --bind 127.0.0.1
```

Linking external Doxygen projects against Zephyr

External projects that build upon Zephyr functionality and wish to refer to Zephyr documentation in Doxygen (through the use of @ref), can utilize the tag file exported at zephyr.tag

Once downloaded, the tag file can be used in a custom `doxyfile.in` as follows:

```
TAGFILES = "/path/to/zephyr.tag=https://docs.zephyrproject.org/latest/doxygen/html/"
```

For additional information refer to Doxygen External Documentation.

Documentation Guidelines
: This page provides some simple guidelines for writing documentation using the reSTructuredText (reST) markup language and Sphinx documentation generator.

Documentation Generation
: As you write documentation, it can be helpful to see how it will look when rendered.

  This page describes how to build the Zephyr documentation locally.

8.3 Dealing with external components

8.3.1 Contributing External Components

In some cases it is desirable to leverage existing, external source code in order to avoid re-implementing basic functionality or features that are readily available in other open source projects.

This section describes the circumstances under which external source code can be imported into Zephyr, and the process that governs the inclusion.

There are three main factors that will be considered during the inclusion process in order to determine whether it will be accepted. These will be described in the following sections.

Note that most of this page deals with external components that end up being compiled and linked into the final image, and programmed into the target hardware. For external tooling that is only used during compilation, code analysis, testing or simulation please refer to the *Contributing External Tooling* section at the end of the page.

Software License

Note

External source code licensed under the Apache-2.0 license is not subject to this section.

Integrating code into the Zephyr Project from other projects that use a license other than the Apache 2.0 license needs to be fully understood in context and approved by the Zephyr governing board, as described in the Zephyr project charter. The board will automatically reject licenses that have not been approved by the Open Source Initiative (OSI). See the *Submission and review process* section for more details.

By carefully reviewing potential contributions and also enforcing a *Developer Certification of Origin (DCO)* for contributed code, we ensure that the Zephyr community can develop products with the Zephyr Project without concerns over patent or copyright issues.

Merit

Just like with any other regular contribution, one that contains external code needs to be evaluated for merit. However, in the particular case of code that comes from an existing project, there are additional questions that must be answered in order to accept the contribution. More specifically, the following will be considered by the Technical Steering Committee and evaluated carefully before the external source code is accepted into the project:

- Is this the most optimal way to introduce the functionality to the project? Both the cost of implementing this internally and the one incurred in maintaining an externally developed codebase need to be evaluated.
- Is the external project being actively maintained? This is particularly important for source code that deals with security or cryptography.
- Have alternatives to the particular implementation proposed been considered? Are there other open source project that implement the same functionality?

Mode of integration

There are two ways of integrating external source code into the Zephyr Project, and careful consideration must be taken to choose the appropriate one for each particular case.

Integration in the main tree The first way to integrate external source code into the project is to simply import the source code files into the main `zephyr` repository. This automatically implies that the imported source code becomes part of the "mainline" codebase, which in turn requires that:

- The code is formatted according to the Zephyr *Coding Style*
- The code adheres to the project's *Coding Guidelines*
- The code is subject to the same checks and verification requirements as the rest of the code in the main tree, including static analysis
- All files contain an SPDX tag if not already present
- If the source is not Apache 2.0 licensed, an entry is added to the *licensing page*.

This mode of integration can be applicable to both small and large external codebases, but it is typically used more commonly with the former.

Integration as a module The second way of integrating external source code into the project is to import the whole or parts of the third-party open source project into a separate repository, and then include it under the form of a *module*. With this approach the code is considered as being developed externally, and thus it is not automatically subject to the requirements of the previous section.

Integration in main manifest file (west.yaml) Integrating external code into the main `west.yml` manifest file is limited to code that is used by a Zephyr subsystem (libraries), by a platform, drivers (HAL) or tooling needed to test or build Zephyr components.

The integration of modules in this group is validated by the Zephyr project CI, and verified to be working with each Zephyr release.

Integrated modules will not be removed from the tree without a detailed migration plan.

Integration as optional modules Standalone or loose integration of modules/projects without any incoming dependencies shall be made optional and shall be kept standalone. Optional projects that provide value to users directly and through a Zephyr subsystem or platform shall be added to an optional manifest file that is filtered by default. (`submanifests/optional.yml`).

Such optional projects might include samples and tests in their own repositories.

There shall not be any direct dependency added in the Zephyr code tree (Git repository) and all sample or test code shall be maintained as part of the module.

> **Note**
>
> This is valid for all new optional modules. Existing optional modules with samples and test code in the Zephyr Git repository will be transitioned out over time.

Integration as external modules Similar to optional modules, but added to the Zephyr project as an entry in the documentation using a pre-defined template. This type of modules exists outside the Zephyr project manifest with documentation instructing users and developers how to integrate the functionality.

Ongoing maintenance

Regardless of the mode of integration, external source code that is integrated in Zephyr requires regular ongoing maintenance. The submitter of the proposal to integrate external source code must therefore commit to maintain the integration of such code for the foreseeable future. This may require adding an entry in the `MAINTAINERS.yml` as part of the process.

Submission and review process

Before external source code can be included in the project, it must be reviewed and accepted by the Technical Steering Committee (TSC) and, in some cases, by the Zephyr governing board.

A request for external source code integration must be made by creating a new issue in the Zephyr project issue tracking system on GitHub with details about the source code and how it integrates into the project.

Follow the steps below to begin the submission process:

1. Make sure to read through the *Contributing External Components* section in detail, so that you are informed of the criteria used by the TSC and board in order to approve or reject a request
2. Use the New External Source Code Issue to open an issue
3. Fill out all required sections, making sure you provide enough detail for the TSC to assess the merit of the request. Optionally you can also create a Pull Request that demonstrates the integration of the external source code and link to it from the issue
4. Wait for feedback from the TSC, respond to any additional questions added as GitHub issue comments

If, after consideration by the TSC, the conclusion is that integrating external source code is the best solution, and the external source code is licensed under the Apache-2.0 license, the submission process is complete and the external source code can be integrated.

If, however, the external source code uses a license other than Apache-2.0, then these additional steps must be followed:

1. The TSC chair will forward the link to the GitHub issue created during the early submission process to the Zephyr governing board for further review
2. The Zephyr governing board has two weeks to review and ask questions:
    - If there are no objections, the matter is closed. Approval can be accelerated by unanimous approval of the board before the two weeks are up
    - If a governing board member raises an objection that cannot be resolved via email, the board will meet to discuss whether to override the TSC approval or identify other approaches that can resolve the objections
3. On approval of the Zephyr TSC and governing board the submission process is complete

The flowchart below shows an overview of the process:

Contributing External Tooling

This section deals exclusively with the inclusion of external tooling in the Zephyr project, where tooling is defined as software that assists the compilation, testing or simulation processes but in no case ends up being part of the code compiled and linked into the final image. “Inclusion” in this context means becoming part of the Zephyr default distribution either in the main tree directly under the `scripts/` folder or indirectly as a west project in the main `west.yml` manifest.

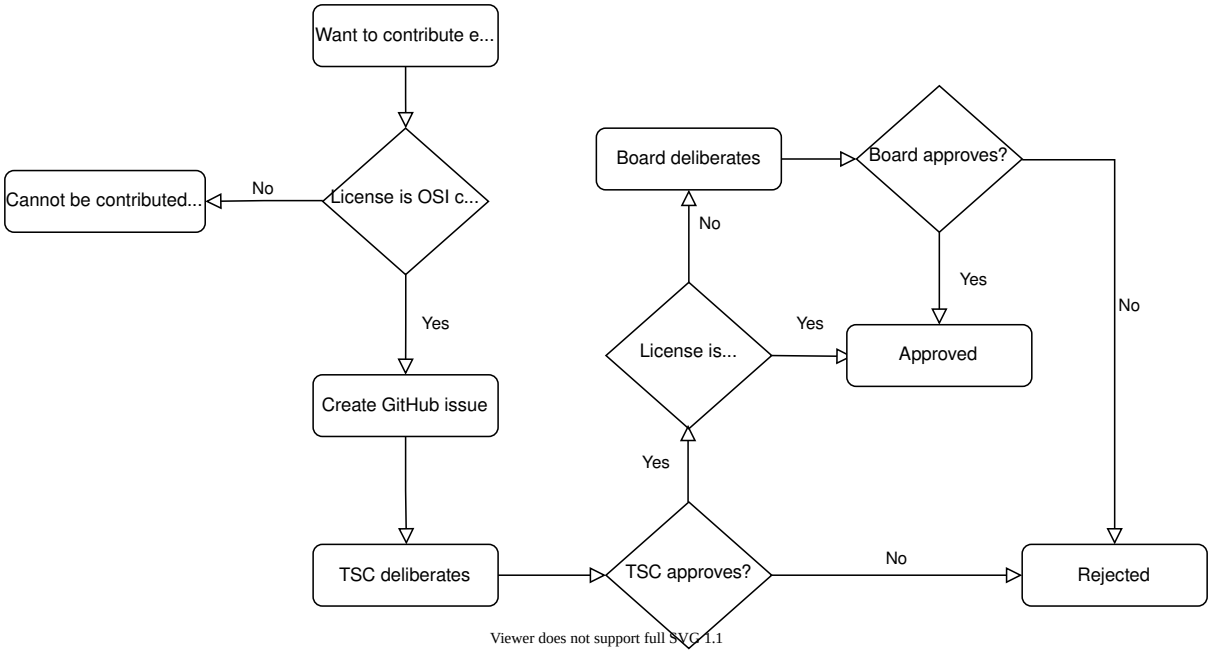


Fig. 2: Submission process

Therefore, this section does not apply to 3rd-party tooling such as toolchains, simulators or others, which may still be referenced by the Zephyr build system or docs without being included in Zephyr.

Tooling components must be released under a license approved by the Open Source Initiative (OSI).

Just like with regular external components, tooling that is imported from another project can be integrated either in the main tree or as a *west project*. Note that in this case the corresponding west project will not be a *module*, because tooling does not make use of the Zephyr build system and does not need to be processed by it. Please see *Modules vs west projects* for additional information on the differences.

If the tool is integrated in the main tree it should be placed under the `scripts/` folder. If the tool is integrated as a west project, then the project repository can be hosted outside the zephyrproject-rtos GitHub organization, provided that the project is made optional via the `group-filter:` field in the main `west.yml` manifest. More info on optional projects can be found in *this section*.

The TSC must approve every Pull Request that introduces a new external tooling component. This will be done on a case-by-case, individual analysis of the proposed addition by the TSC representatives.

Additional considerations about the main manifest

In general, any additions or removals whatsoever to the `projects:` section of the main manifest file requires TSC approval. This includes, but is not limited to:

- Adding and removing groups and group filters
- Adding and removing projects
- Adding and removing `import` statements

8.3.2 Binary Blobs

In the context of an operating system that supports multiple architectures and many different IC families, some functionality may be unavailable without the help of executable code distributed in binary form. Binary blobs (or blobs for short) are files containing proprietary machine code or data in a binary format, e.g. without corresponding source code released under an OSI approved license.

Zephyr supports downloading and using third-party binary blobs via its built-in mechanisms, with some important caveats, described in the following sections. It is important to note that all the information in this section applies only to upstream (vanilla) Zephyr.

There are no limitations whatsoever (except perhaps license compatibility) in the support for binary blobs in forks or third-party distributions of Zephyr. In fact, Zephyr's build system supports arbitrary use cases related to blobs. This includes linking against libraries, flashing images to targets, etc. Users are therefore free to create Zephyr-based downstream software which uses binary blobs if they cannot meet the requirements described in this page.

Software license

Most binary blobs are distributed under proprietary licenses which vary significantly in nature and conditions. It is up to the vendor to specify the license as part of the blob submission process. Blob vendors may impose a click-through or other EULA-like workflow when users fetch and install blobs.

Hosting

Blobs must be hosted on the Internet and managed by third-party infrastructure. Two potential examples are Git repositories and web servers managed by individual hardware vendors.

The Zephyr Project does not host binary blobs in its Git repositories or anywhere else.

Fetching blobs

Blobs are fetched from official third-party sources by the *west blobs* command.

The blobs themselves must be specified in the *module.yml* files included in separate Zephyr *module repositories* maintained by their respective vendors. This means that in order to include a reference to a binary blob to the upstream Zephyr distribution, a module repository must exist first or be created as part of the submission process.

Each blob which may be fetched must be individually identified in the corresponding `module.yml` file. A specification for a blob must contain:

- An abstract description of the blob itself
- Version information
- A reference to vendor-provided documentation
- The blob's *type*, which must be one of the allowed types
- A checksum for the blob, which `west blobs` checks after downloading. This is required for reproducibility and to allow bisecting issues as blobs change using Git and west
- License text applicable to the blob or a reference to such text, in SPDX format

See the *corresponding section* for a more formal definition of the fields.

The *west blobs* command can be used to list metadata of available blobs and to fetch blobs from user-selected modules.

The `west blobs` command only fetches and stores the binary blobs themselves. Any accompanying code, including interface header files for the blobs, must be present in the corresponding module repository.

Tainting

Inclusion of binary blobs will taint the Zephyr build. The definition of tainting originates in the Linux kernel and, in the context of Zephyr, a tainted image will be one that includes binary blobs in it.

Tainting will be communicated to the user in the following manners:

- One or more Kconfig options `TAINT_BLOBS_*` will be set to `y`
- The Zephyr build system, during its configuration phase, will issue a warning. It will be possible to disable the warning using Kconfig
- The `west spdx` command will include the tainted status in its output
- The kernel's default fatal error handler will also explicitly print out the kernel's tainted status

Allowed types

The following binary blob types are acceptable in Zephyr:

- Precompiled libraries: Hardware enablement libraries, distributed in precompiled binary form, typically for SoC peripherals. An example could be an enablement library for a wireless peripheral
- Firmware images: An image containing the executable code for a secondary processor or CPU. This can be full or partial (typically delta or patch data) and is generally copied into RAM or flash memory by the main CPU. An example could be the firmware for the core running a Bluetooth LE Controller
- Miscellaneous binary data files. An example could be pre-trained neural network model data

Hardware agnostic features provided via a proprietary library are not acceptable. For example, a proprietary and hardware agnostic TCP/IP stack distributed as a static archive would be rejected.

Note that just because a blob has an acceptable type does not imply that it will be unconditionally accepted by the project; any blob may be rejected for other reasons on a case by case basis (see library-specific requirements below). In case of disagreement, the TSC is the arbiter of whether a particular blob fits in one of the above types.

Precompiled library-specific requirements

This section contains additional requirements specific to precompiled library blobs.

Any person who wishes to submit a precompiled library must represent that it meets these requirements. The project may remove a blob from the upstream distribution if it is discovered that the blob fails to meet these requirements later on.

Interface header files The precompiled library must be accompanied by one or more header files, distributed under a non-copyleft OSI approved license, that define the interface to the library.

Allowed dependencies This section defines requirements related to external symbols that a library blob requires the build system to provide.

- The blob must not depend on Zephyr APIs directly. In other words, it must have been possible to build the binary without any Zephyr source code present at all. This is required for loose coupling and maintainability, since Zephyr APIs may change and such blobs cannot be modified by all project maintainers
- Instead, if the code in the precompiled library requires functionality provided by Zephyr (or an RTOS in general), an implementation of an OS abstraction layer (aka porting layer) can be provided alongside the library. The implementation of this OS abstraction layer must be in source code form, released under an OSI approved license and documented using Doxygen

Toolchain requirements Precompiled library blobs must be in a data format which is compatible with and can be linked by a toolchain supported by the Zephyr Project. This is required for maintainability and usability. Use of such libraries may require special compiler and/or linker flags, however. For example, a porting layer may require special flags, or a static archive may require use of specific linker flags.

Limited scope Allowing arbitrary library blobs carries a risk of degrading the degree to which the upstream Zephyr software distribution is open source. As an extreme example, a target with a zephyr kernel clock driver that is just a porting layer around a library blob would not be bootable with open source software.

To mitigate this risk, the scope of upstream library blobs is limited. The project maintainers define an open source test suite that an upstream target must be able to pass using only open source software included in the mainline distribution and its modules. The open source test suite currently consists of:

- `samples/philosophers`
- `tests/kernel`

The scope of this test suite may grow over time. The goal is to specify tests for a minimal feature set which must be supported via open source software for any target with upstream Zephyr support.

At the discretion of the release team, the project may remove support for a hardware target if it cannot pass this test suite.

Support and maintenance

The Zephyr Project is not expected to be responsible for the maintenance and support of contributed binary blobs. As a consequence, at the discretion of the Zephyr Project release team, and on a case-by-case basis:

- GitHub issues reported on the zephyr repository tracker that require use of blobs to reproduce may not be treated as bugs
- Such issues may be closed as out of scope of the Zephyr project

This does not imply that issues which require blobs to reproduce will be closed without investigation. For example, the issue may be exposing a bug in a Zephyr code path that is difficult or impossible to trigger without a blob. Project maintainers may accept and attempt to resolve such issues.

However, some flexibility is required because project maintainers may not be able to determine if a given issue is due to a bug in Zephyr or the blob itself, may be unable to reproduce the bug due to lack of hardware, etc.

Blobs must have designated maintainers that must be responsive to issue reports from users and provide updates to the blobs to address issues. At the discretion of the Zephyr Project release team, module revisions referencing blobs may be removed from `zephyr/west.yml` at any time due to lack of responsiveness or support from their maintainers. This is required to maintain project control over bit-rot, security issues, etc.

The submitter of the proposal to integrate a binary blob must commit to maintain the integration of such blob for the foreseeable future.

Regarding Continuous Integration, binary blobs will **not** be fetched in the project's CI infrastructure that builds and optionally executes tests and samples to prevent regressions and issues from entering the codebase. This includes both CI ran when a new GitHub Pull Request is opened as well as any other regularly scheduled execution of the CI infrastructure.

Submission and review process

For references to binary blobs to be included in the project, they must be reviewed and accepted by the Technical Steering Committee (TSC). This process is only required for new binary blobs, updates to binary blobs follow the *module update procedure*.

A request for integration with binary blobs must be made by creating a new issue in the Zephyr project issue tracking system on GitHub with details about the blobs and the functionality they provide to the project.

Follow the steps below to begin the submission process:

1. Make sure to read through the *Binary Blobs* section in detail, so that you are informed of the criteria used by the TSC in order to approve or reject a request
2. Use the New Binary Blobs Issue to open an issue
3. Fill out all required sections, making sure you provide enough detail for the TSC to assess the merit of the request. Additionally you must also create a Pull Request that demonstrates the integration of the binary blobs and then link to it from the issue
4. Wait for feedback from the TSC, respond to any additional questions added as GitHub issue comments

If, after consideration by the TSC, the submission of the binary blob(s) is approved, the submission process is complete and the binary blob(s) can be integrated.

Contributing External Components
: Basic functionality or features that would make useful addition to Zephyr might be readily available in other open source projects, and it is recommended and encouraged to reuse such code. This page describes in more details when and how to import external source code into Zephyr.

Contributing External Tooling
: Similarly, external tooling used during compilation, code analysis, testing or simulation, can be beneficial and is covered in this section.

Binary Blobs
: As some functionality might only be made available with the help of executable code distributed in binary form, this page describes the process and guidelines for *contributing binary blobs* to the project.

8.4 Zephyr Contributor Badge

When your first contribution to the Zephyr project gets merged, you'll become eligible to claim your Zephyr Contributor Badge. This digital badge can be displayed on your website, blog, social

media profile, etc. It will allow you to showcase your involvement in the Zephyr project and help raise its awareness.

You may apply for your Contributor Badge by filling out the Zephyr Contributor Badge form.

8.5 Need help along the way?

If you have questions related to the contribution process, the Zephyr community is here to help. You may join our Discord channel or use the Developer Mailing List.

Chapter 9

Project and Governance

9.1 Technical Steering Committee (TSC)

9.1.1 TSC Member Role

The TSC role and its responsibilities is defined in the Zephyr project charter.

Membership

A TSC member plays a pivotal role in shaping the technical direction of the Zephyr Project. TSC members work collaboratively with other TSC members, contributors, and stakeholders to ensure the project's success and sustainability.

By fulfilling the rights and responsibilities below, TSC members contribute to the overall success and growth of the Zephyr Project, ensuring that it remains a vibrant and thriving open-source community for years to come.

Rights

Decision Making
: Participate in key decisions related to the project's technical direction, including architectural changes, feature additions, and release planning.

Voting
: Exercise voting rights on important matters discussed within the TSC, including feature proposals, code contributions, and community initiatives.

Access
: Gain access to relevant project repositories, documentation, and communication channels to stay informed and contribute effectively.

Leadership
: Take on leadership roles within working groups or subcommittees dedicated to specific technical areas or initiatives.

Representation
: Act as a representative of the broader Zephyr community, advocating for the interests of contributors, users, and stakeholders.

Responsibilities TSC members are expected to fulfill the following responsibilities, though it is not mandatory to fulfill all:

Technical Oversight
: Provide guidance and oversight on technical matters, ensuring alignment with project goals, standards, and best practices through active participation as core members in working groups and committees.

Code Review
: Participate in code reviews to maintain code quality, consistency, and compatibility with project standards.

Community Engagement
: Engage with the community through forums, mailing lists, conferences, and other channels to foster collaboration, address concerns, and gather feedback.

Documentation
: Contribute to the development and maintenance of project documentation, including technical guides, API references, and best practices.

Release Management
: Collaborate with the release manager and other stakeholders to plan and coordinate project releases, ensuring timely delivery and quality assurance.

Contributor Support
: Support and mentor new contributors, helping them navigate the project's codebase, processes, and community norms.

Issue Triage
: Assist in triaging and prioritizing issues reported by users and contributors, facilitating timely resolution and communication.

Compliance and Licensing
: Ensure compliance with project licensing requirements and open-source best practices, addressing any licensing-related issues that may arise.

Conflict Resolution
: Facilitate constructive discussions and resolution of technical disagreements or conflicts within the community, promoting a healthy and inclusive environment.

Continuous Improvement
: Continuously seek opportunities to improve project governance, processes, and infrastructure, driving innovation and sustainability.

Appointed TSC Members

See Zephyr project charter for more details.

- Appointed TSC members have no term limits besides the term of their employment at the organization they represent or their organization's membership in the Zephyr Project.
- Appointed TSC members can select an Alternate from the same organization.

Elected TSC Members

Per the Zephyr project charter, TSC members can nominate representatives from the technical community at the rate of no more than one per quarter.

- Majority vote is required to confirm a candidate.
- Once elected, a TSC member serves for 2 years.
- Elected TSC members do not have the right to appoint an Alternate.

- To ensure continuity of the TSC, at the end of the 2 year term, the TSC is required to reconfirm the membership of elected members. If the elected member declines a new term or if the TSC fails to reconfirm the term, the seat will be open for new nominations.
- If an elected TSC member resigns before the end of the 2 year term, their spot will be open for new members outside of the quarterly nomination limit. The elected member will serve a 2 year term.
- The TSC has the right to terminate elected members who become inactive and are not fulfilling the responsibilities of TSC members as described in this document.
- The number of elected members shall not exceed 20% of the total of appointed members.
- Existing TSC members who were elected before May 2024 shall be re-confirmed after completing the 2 year term since they were first elected.

Suspensions

As noted under Section 8b of the Project Charter, voting rights for a representative who misses three consecutive meetings are subject to suspension and suspended representatives do not count towards the quorum requirement.

A representative's suspension will end and voting rights be restored at the start of the next attended meeting. The TSC enforces the suspension policy for voting members who miss three consecutive TSC weekly meetings.

Multi-day meetings (F2F events) are counted as "one" meeting. The TSC voted on February 16, 2022 to discontinue default enforcement of the suspension policy. The TSC voted on January 18, 2023 to re-enact default enforcement of the suspension policy.

Notice of suspension will be sent to representatives who miss three consecutive meetings, noting that rights will be restored upon next attendance of a TSC meeting.

Note

As per Section 4b of the Project Charter, Platinum and Silver Members may choose to opt out of a voting seat on the TSC.

Members who opt out and then wish to reclaim their seat later will have their voting rights restored at the start of the second consecutive meeting attended following notification to the TSC Chair.

Voting

Voting in the Zephyr Project is defined under Section 8 of the Project Charter.

Additional points of clarity / TSC interpretation have been added below. The Governing Board may opt to update the Charter to include the below refinements. Until then, additional clarifications (if/where needed) will be discussed in the Process Working Group, and approved in the TSC.

- TSC In-Meeting Voting For items requesting an in-meeting vote of the Zephyr Technical Steering Committee (TSC), assuming quorum requirements have been met, the default voting mechanism will be a verbal motion to determine if there is general consensus.
- If there are no objections to a motion being brought forward, general consensus is assumed and the motion passes.
- Should there be any objections raised, the vote will move to email, and be executed using the Voting Guidelines outlined in Section 8 of the Project Charter.

- Should a motion be deemed urgent by the TSC Chair, and assuming quorum requirements have been met, the Chair may call for a roll call vote in-meeting.

Voting Options

- Voting Options are:
  - "Yes",
  - "No" or
  - "Abstain"

Abstention Abstentions do not count in tallying the vote negatively or positively; when members abstain, they are in effect attending only to contribute to a quorum.

Abstentions do not impact the number of votes needed to decide a vote.

Quorum Quorum for TSC meetings shall require 60% of the voting representatives… (ref 8b of the Charter)

Decisions Decisions by vote shall be based on a majority vote, provided that at least sixty percent (60%) of the **TSC** representatives must be either present or participating electronically or by written action in order to conduct a valid vote. (ref 8c of the Charter)

Example A:

> 40 eligible TSC voters. 3 abstain from a vote on a motion. 12 vote Yes. 11 vote No.
>
> Quorum reached: 26 votes cast (quorum = 60% of 40 = 24) Majority vote: 12 Yes vs. 11 No. Yes wins. Motion adopted.

Example B:

> 40 eligible TSC voters. 5 abstain from a vote on a motion. 12 vote Yes. 6 vote No. Quorum reached? 23 votes cast (quorum = 60% of 40 = 24)
>
> Vote is not valid. Quorum not reached.

Example C:

> 40 eligible TSC voters. 21 abstain from a vote on a motion. 2 vote Yes. 1 votes No. Quorum reached? 24 votes cast (quorum = 60% of 40 = 24)
>
> Majority vote: 2 Yes vs. 1 No. Yes wins.

Immutable Votes

Votes are considered immutable once cast. A voter may not change their vote, once cast, between the time a Motion is brought forth and the time at which results are announced.

9.2 TSC Project Roles

9.2.1 Project Roles

TSC projects generally will involve *Maintainers*, *Collaborators*, and *Contributors*:

Maintainer: lead Collaborators on an area identified by the TSC (e.g. Architecture, code subsystems, etc.). Maintainers shall also serve as the area's representative on the TSC as needed. Maintainers may become voting members of the TSC under the guidelines stated in the project Charter.

Collaborator: A highly involved Contributor in one or more areas. May become a Maintainer with approval of existing TSC voting members.

Contributor: anyone in the community that contributes code or documentation to the project. Contributors may become Collaborators by approval of the existing Collaborators and Maintainers of the particular code base areas or subsystems.

Contributor

A *Contributor* is a developer who wishes to contribute to the project, at any level.

Contributors are granted the following rights and responsibilities:

- Right to contribute code, documentation, translations, artwork, etc.
- Right to report defects (bugs) and suggestions for enhancement.
- Right to participate in the process of reviewing contributions by others.
- Right to initiate and participate in discussions in any communication methods.
- Right to approach any member of the community with matters they believe to be important.
- Right to participate in the feature development process.
- Responsibility to abide by decisions, once made. They are welcome to provide new, relevant information to reopen decisions.
- Responsibility for issues and bugs introduced by one's own contributions.
- Responsibility to respect the rules of the community.
- Responsibility to provide constructive advice whenever participating in discussions and in the review of contributions.
- Responsibility to follow the project's code of conduct (https://github.com/zephyrproject-rtos/zephyr/blob/main/CODE_OF_CONDUCT.md)

Contributors are initially only given Read access to the Zephyr GitHub repository. Specifically, at the Read access level, Contributors are not allowed to assign reviewers to their own pull requests. An automated process will assign reviewers. You may also share the pull request on the Zephyr devel mailing list or on the Zephyr Discord Server.

Contributors who show dedication and skill are granted the Triage permission level to the Zephyr GitHub repository.

You may nominate yourself, or another GitHub user, for promotion to the Triage permission level by creating a GitHub issue, using the nomination template.

Contributors granted the Triage permission level are permitted to add reviewers to a pull request and can be added as a reviewer by other GitHub users. Contributor change requests or approval on pull requests are not counted with respect to accepting and merging a pull request. However, Contributors comments and requested changes should still be considered by the pull request author.

Collaborator

A *Collaborator* is a Contributor who is also responsible for the maintenance of Zephyr source code. Their opinions weigh more when decisions are made, in a fully meritocratic fashion.

Collaborators have the following rights and responsibilities, in addition to those listed for Contributors:

- Right to set goals for the short and medium terms for the project being maintained, alongside the Maintainer.
- Responsibility to participate in the feature development process.
- Responsibility to review relevant code changes within reasonable time.
- Responsibility to ensure the quality of the code to expected levels.
- Responsibility to participate in community discussions.
- Responsibility to mentor new contributors when appropriate
- Responsibility to participate in the quality verification and release process, when those happen.

Contributors are promoted to the Collaborator role by adding the GitHub user name to one or more `collaborators` sections of the *MAINTAINERS File* in the Zephyr repository.

Collaborator change requests on pull requests should be addressed by the original submitter. In cases where the changes requested do not follow the *expectations* and the guidelines of the project or in cases of disagreement, it is the responsibility of the assignee to advance the review process and resolve any disagreements.

Collaborator approval of pull requests are counted toward the minimum required approvals needed to merge a PR. Other criteria for merging may apply.

Maintainer

A *Maintainer* is a Collaborator who is also responsible for knowing, directing and anticipating the needs of a given zephyr source code area.

Maintainers have the following rights and responsibilities, in addition to those listed for Contributors and Collaborators:

- Right to set the overall architecture of the relevant subsystems or areas of involvement.
- Right to make decisions in the relevant subsystems or areas of involvement, in conjunction with the collaborators and submitters. See *PR Technical Escalation*.
- Responsibility to convey the direction of the relevant subsystem or areas to the TSC
- Responsibility to ensure all contributions of the project have been reviewed within reasonable time.
- Responsibility to enforce the code of conduct.
- Responsibility to triage static analysis issues in their code area. See *Static Code Analysis*.

Contributors or Collaborators are promoted to the Maintainer role by adding the GitHub user name to one or more `maintainers` sections of the *MAINTAINERS File* in the Zephyr repository.

Maintainer approval of pull requests are counted toward the minimum required approvals needed to merge a PR. Other criteria for merging may apply.

9.2.2 Role Retirement

- Individuals elected to the following Project roles, including, Maintainer, Release Engineering Team member, Release Manager, but are no longer engaged in the project as described by the rights and responsibilities of that role, may be requested by the TSC to retire from the role they are elected.

- Such a request needs to be raised as a motion in the TSC and be approved by the TSC voting members. By approval of the TSC the individual is considered to be retired from the role they have been elected.
- The above applies to elected TSC Project roles that may be defined in addition.

9.2.3 Teams and Supporting Activities

Assignee

An *Assignee* is one of the maintainers of a subsystem or code being changed. Assignees are set either automatically based on the code being changed or set by the other Maintainers, the Release Engineering team can set an assignee when the latter is not possible.

- Responsibility to drive the pull request to a mergeable state
- Right to dismiss stale and unrelated reviews or reviews not following *expectations* from reviewers and seek reviews from additional maintainers, developers and contributors
- Right to block pull requests from being merged until issues or changes requested are addressed
- Responsibility to re-assign a pull request if they are the original submitter of the code
- Solicit approvals from maintainers of the subsystems affected
- Responsibility to drive the *PR Technical Escalation* process

Static Analysis Audit Team

The Static Analysis Audit team works closely with the release engineering team to ensure that static analysis defects opened during a release cycle are properly addressed. The team has the following rights and responsibilities:

- Right to revert any triage in a static analysis tool (e.g: Coverity) that does not follow the project expectations.
- Responsibility to inform code owners about improper classifications.
- Responsibility to alert TSC if any issues are not adequately addressed by the responsible code owners.

Joining the Static Analysis Audit team

- Contributors highly involved in the project with some expertise in static analysis.

Release Engineering Team

A team of active Maintainers involved in multiple areas.

- The members of the Release Engineering team are expected to fill the Release Manager role based on a defined cadence and selection process.
- The cadence and selection process are defined by the Release Engineering team and are approved by the TSC.
- The team reports directly into the TSC.

Release Engineering team has the following rights and responsibilities:

- Right to merge code changes to the zephyr tree following the project rules.
- Right to revert any changes that have broken the code base

- Right to close any stale changes after <N> months of no activity
- Responsibility to take directions from the TSC and follow them.
- Responsibility to coordinate code merges with maintainers.
- Responsibility to merge all contributions regardless of their origin and area if they have been approved by the respective maintainers and follow the merge criteria of a change.
- Responsibility to keep the Zephyr code base in a working and passing state (as per CI)

Joining the Release Engineering team

- Maintainers highly involved in the project may be nominated by a TSC voting member to join the Release Engineering team. Nominees may become members of the team by approval of the existing TSC voting members.
- To ensure a functional Release Engineering team the TSC shall periodically review the team's followed processes, the appropriate size, and the membership composition (ensure, for example, that team members are geographically distributed across multiple locations and time-zones).

Release Manager

A *Maintainer* responsible for driving a specific release to completion following the milestones and the roadmap of the project for this specific release.

- TSC has to approve a release manager.

A Release Manager is a member of the Release Engineering team and has the rights and responsibilities of that team in addition to the following:

- Right to manage and coordinate all code merges after the code freeze milestone (M3, see program management overview.)
- Responsibility to drive and coordinate the triaging process for the release
- Responsibility to create the release notes of the release
- Responsibility to notify all stakeholders of the project, including the community at large about the status of the release in a timely manner.
- Responsibility to coordinate with QA and validation and verify changes either directly or through QA before major changes and major milestones.

Roles / Permissions

Table 1: Project Roles vs GitHub Permissions

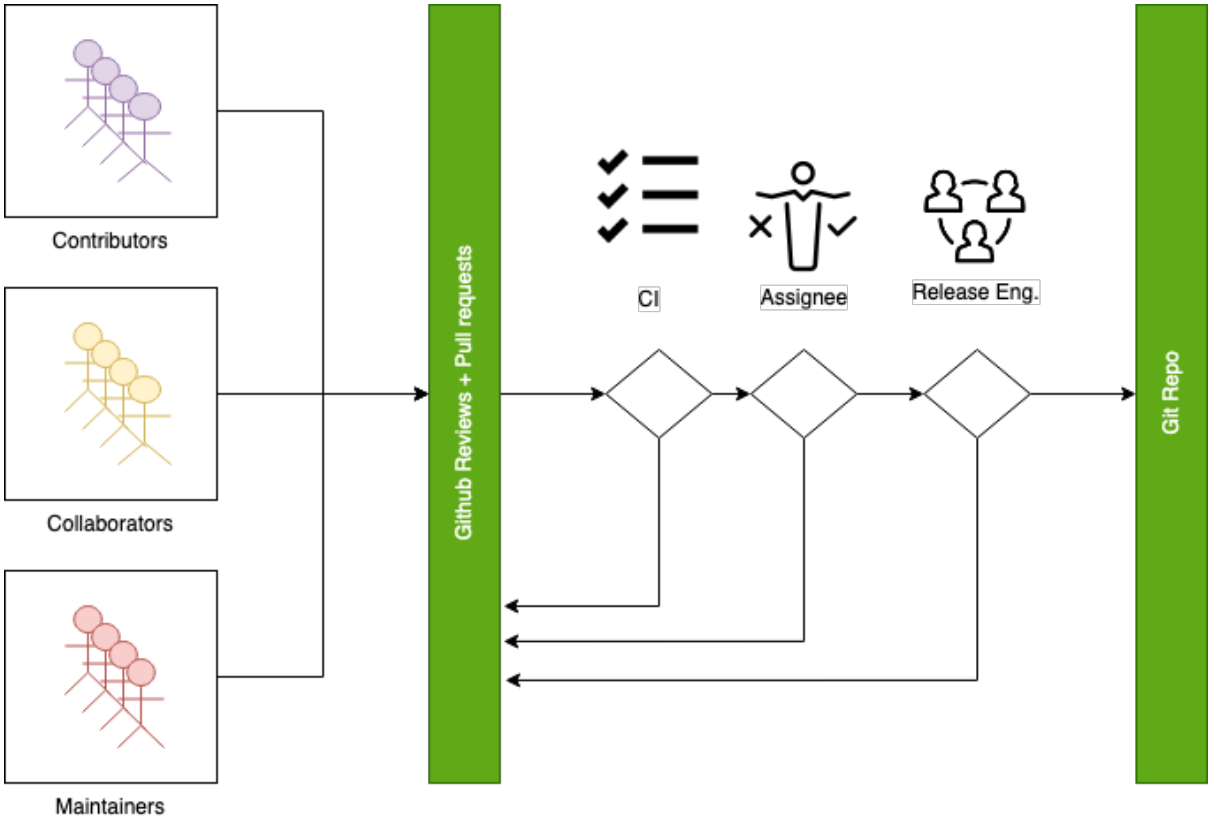
		Admin	**Merge Rights**	Member	Owner	Collaborator
Main Roles	Contributor					x
	Collaborator			x		
	Maintainer			x		
Supportive Roles	QA/Validation			x		x
	DevOps	**x**				
	System Admin	**x**			x	
	Release Engineering	**x**	**x**	x		

9.2.4 MAINTAINERS File

Generic guidelines for deciding and filling in the Maintainers' list

- The MAINTAINERS.yml file shall replace the CODEOWNERS file and will be used for both setting assignees and reviewers.
- We should keep the granularity of code maintainership at a manageable level
- We should be looking for maintainers for areas of code that are orphaned (i.e. without an explicit maintainer)
  - Un-maintained areas should be indicated clearly in the MAINTAINERS file
- All submitted pull requests should have an assignee
- We Introduce an area/subsystem hierarchy to address the above point
  - Parent-area maintainer should be acting as default substitute/fallback assignee for un-maintained sub-areas
  - Area maintainer gets precedence over parent-area maintainer
- Pull requests may be re-assigned if this is needed or more appropriate
  - Re-assigned by original assignee
- In general, updates to the MAINTAINERS file should be in a standalone commit alongside other changes introducing new files and directories to the tree.
- Major changes to the file, including the addition of new areas with new maintainers should come in as standalone pull requests and require TSC review.
- If additional review by the TSC is required, the maintainers of the file should send the requested changes to the TSC and give members of the TSC two (2) days to object to any of the changes to maintainership of areas or the addition of new maintainers or areas.
- Path, collaborator and name changes do not require a review by the TSC.
- Addition of new areas without a maintainer do not require review by the TSC.
- The MAINTAINERS file itself shall have a maintainer
- Architectures, core components, sub-systems, samples, tests
  - Each area shall have an explicit maintainer
- Boards (incl relevant samples, tests), SoCs (incl DTS) * May have a maintainer, shall have a higher-level platform maintainer
- Drivers
  - Shall have a driver-area (and API) maintainer
  - Could have individual driver implementation maintainers but preferably collaborator/contributors
  - In the above case, platform-specific PRs may be re-assigned to respective collaborator/contributor of driver implementation

9.2.5 Release Activity



Merge Criteria

- Minimal of 2 approvals, including an approval by the designated assignee.
- Pull requests should be reviewed by at least a maintainer or collaborator of each affected area; Unless the changes to a given area are considered trivial enough, in which case approvals by other affected subsystems maintainers/collaborators would suffice.
- Four eye principle on the organisation level. We already require at least 2 approvals (basic four eye principle), however, such reviews and approvals might be unintentionally biased in the case where the submitter is from the same organisation as the approvers. To allow for project wide review and approvals, the merge criteria is extended with the guidelines below:
  - Changes or additions to common and shared code shall have approvals from different organisations (at least one approval from an organisation different than the submitters'). Common and shared code is defined as anything that does not fall under `soc`, `boards` and `drivers/*/*`.
  - Changes or additions to hardware support (driver, SoC, boards) shall at least have the merger be from a different organisation. This applies only to implementation of an API supporting vendor specific hardware and not the APIs.
  - Release engineers may make exceptions for areas with contributions primarily coming from one organisation and where reviews from other organisations are not possible, however, merges shall be completed by a person from a different organisation. In such cases, the minimum review period of at least 2 days shall be strictly followed to allow for additional reviews.
  - Release engineers shall not merge code changes originating and reviewed only by their own organisation. To be able to merge such changes, at least one review shall be from a different organisation.

- A minimum review period of 2 business days, 4 hours for trivial changes (see *Give reviewers time to review before code merge*).
- Hotfixes can be merged at any time after CI has passed and are excluded from most of the conditions listed above.
- All required checks are passing:
  - Codeowners
  - Device Tree
  - Documentation
  - Gitlint
  - Identity/Emails
  - Kconfig
  - License checks
  - Checkpatch (Coding Style)
  - Pylint
  - Integration Tests (Via twister) on emulation/simulation platforms
  - Simulated Bluetooth Tests
- Planned
  - Footprint
  - Code coverage
  - Coding Guidelines
  - Static Analysis (Coverity)
  - Documentation coverage (APIs)

9.3 TSC Working Groups

9.3.1 Overview

The TSC, at its discretion, may establish working groups or subcommittees to serve as focused teams dedicated to specific technical areas, initiatives, or tasks.

9.3.2 Membership

Working Group Membership Eligibility

- Each Working group (WG) shall determine its own membership eligibility, in consultation with the TSC.
- Each working group shall have a team of members who are actively involved in its activities and decision-making processes.
- It is expected that WG membership shall be **open to all Zephyr project :ref:‘Collaborators <collaborator>‘**; however, working groups may impose restrictions such as the number of participants from a single company.
- All TSC members are eligible to join a working group as members, part of the responsibilities being a TSC member.

- The minimal number of members may vary depending on the complexity of the tasks and the breadth of expertise required to address them effectively.
- A working group should aim to have at least five to seven members to ensure diversity of perspectives, collaboration, and continuity.
- The structure of each working group within the Zephyr Project should be designed to ensure effectiveness, productivity, and inclusivity. While the optimal size of a working group can vary depending on the specific context and scope of its activities.
- Participation in WG meetings and discussions is open to all project *contributors*.

Working Group Chair / Co-chair

Each working group may elect a Chair and optionally a Co-Chair who is responsible for leading meetings and representing the working group to the TSC.

Working Group Chair / Co-Chair Elections

- The Chair and Co-Chair shall be elected by the members of the working group
- Any member of the working group has the right to nominate themselves for the chair/co-chair positions.
- The term for the chair/co-chair is one year
- If a chair/co-chair resigns from the position before the end of the term, a vote is to be held to elect a new chair/co-chair.

Working Group Voter Eligibility

- Voting for a Chair or Co-Chair is open to the members of the working group.
- Only 1 working group member from each company may vote in the election.
- The Chair and Co-Chair shall be members of the working group.

Working Group Election Confirmation

- The elected Chair (and/or Co-Chair) is submitted to the TSC for confirmation.
- The TSC decides to accept the outcome or requests a new voting.

9.3.3 Advisory role

- Working Groups are advisory in nature. They provide advice to the projects and to the TSC.
- Working groups operate on a rough consensus basis. If the working group is unable to reach consensus on what advice to offer, the working group Chair shall raise the issue with the TSC or the relevant committee (Safety and Security), where a formal vote can be taken, or advise the project that the working group cannot reach consensus.
- Working groups shall keep track of discussions and record any votes, decisions, or recommendations made and share results with the community and the TSC.
- Working group meetings and offline discussions shall be captured in a standalone document with all supporting details such as attendance, quorum, actions to be taken, and next steps.

- Decisions made within a working group are non-binding and are only considered ratified after communicating decisions and outcomes to the TSC.
- Lacking any objections from the TSC within 1 week after the communication or report of any results, decisions of a working group are considered confirmed and ratified.

9.3.4 TSC Working Group Lifecycle

Creation of a TSC working group

In order to create a TSC working group, a TSC member shall make a proposal to the TSC (via TSC email list) that shall cover at least the following:

- TSC working group name.
- TSC working group purpose
- TSC working group expected deliverables
- TSC working group starting participants with at least one TSC member acting as a sponsor.
- Optionally TSC working group definition of done

Update of a TSC working group

The TSC can modify a TSC working group via a TSC decision. To request such a modification, a request is made to the TSC email list.

Conclusion of a TSC working group

The TSC decides the termination of the TSC working group in accordance with the TSC decision procedure. The submission of a request to terminate the TSC working group should cover:

- TSC working group name
- TSC working group deliveries
- Motivation for termination of the TSC working group

9.4 Release Process

The Zephyr project releases on a time-based cycle, rather than a feature-driven one. Zephyr releases represent an aggregation of the work of many contributors, companies, and individuals from the community.

A time-based release process enables the Zephyr project to provide users with a balance of the latest technologies and features and excellent overall quality. A roughly 4-month release cycle allows the project to coordinate development of the features that have actually been implemented, allowing the project to maintain the quality of the overall release without delays because of one or two features that are not ready yet.

The Zephyr release model was loosely based on the Linux kernel model:

- Release tagging procedure:
  - linear mode on main branch,
  - release branches for maintenance after release tagging.

- Each release period will consist of a development phase followed by a stabilization phase. Release candidates will be tagged during the stabilization phase. During the stabilization phase, only stabilization changes such as bug fixes and documentation will be merged unless granted a special exemption by the Technical Steering Committee.
  - Development phase: all changes are considered and merged, subject to approval from the respective maintainers.
  - Stabilisation phase: the release manager creates a vN-rc1 tag and the tree enters the stabilization phase
  - CI sees the tag, builds and runs tests; Test teams analyse the report from the build and test run and give an ACK/NAK to the build
  - The release owner, with test teams and any other needed input, determines if the release candidate is a go for release
  - If it is a go for a release, the release owner lays a tag release vN at the same point

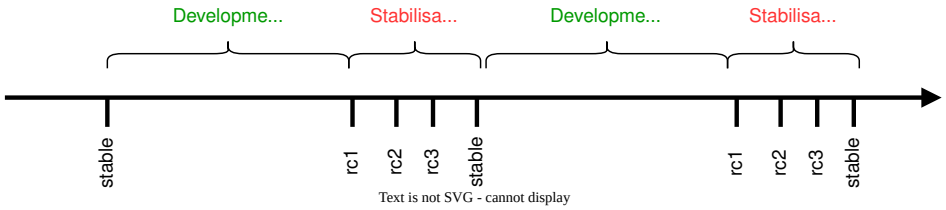


Fig. 1: Release Cycle

Note

The milestones for the current major version can be found on the Official GitHub Wiki. Information on previous releases can be found here.

9.4.1 Development Phase

A relatively straightforward discipline is followed with regard to the merging of patches for each release. At the beginning of each development cycle, the main branch is said to be open for development. At that time, code which is deemed to be sufficiently stable (and which is accepted by the maintainers and the wide community) is merged into the mainline tree. The bulk of changes for a new development cycle (and all of the major changes) will be merged during this time.

The development phase lasts for approximately three months. At the end of this time, the release owner will declare that the development phase is over and releases the first of the release candidates. For the codebase release which is destined to be 3.1.0, for example, the release which happens at the end of the development phase will be called 3.1.0-rc1. The -rc1 release is the signal that the time to merge new features has passed, and that the time to stabilize the next release of the code base has begun.

9.4.2 Stabilization Phase

Over the next weeks and depending on the release milestone, only stabilization, cosmetic changes, tests, bug and doc fixes are allowed (See *table* below).

On occasion, more significant changes and new features will be allowed, but such occasions are rare and require a TSC approval and a justification. As a general rule, if you miss submitting your code during the development phase for a given feature, the best thing to do is to wait for the next

development cycle. (An occasional exception is made for drivers for previously unsupported hardware; if they do not touch any other in-tree code, they cannot cause regressions and should be safe to add at any time).

As fixes make their way into the mainline, the patch rate will slow over time. The mainline release owner releases new -rc drops once or twice a week; a normal series will get up to somewhere between -rc4 and -rc6 before the code base is considered to be sufficiently stable and the release criteria have been achieved at which point the final 3.1.0 release is made.

At that point, the whole process starts over again.

9.4.3 Release Criteria

The main motivation is to clearly have the criteria in place that must be met for a release. This will help define when a release is "done" in terms that most people can understand and in ways that help new people to understand the process and participate in creating successful releases:

- The release criteria documents all the requirements of our target audience for each Zephyr release
- The target audiences for each release can be different, and may overlap
- The criteria at any given time are not set in stone: there may be requirements that have been overlooked, or that are new, and in these cases, the criteria should be expanded to ensure all needs are covered.

Below is the high level criteria to be met for each release:

- No blocker bugs / blocking issues
- All relevant tests shall pass on `Tier 0` platforms
- All relevant tests shall pass on Tier 0 and 1 platforms (at least 1 per architecture/architecture variant/Hardware features)
- All applicable samples/tests shall build on Tiers 0, 1 and 2
- All high and critical static analysis and security issues addressed
- Release Notes are up-to-date.

Blocker Bugs

Blocker bug process kicks in during the release process and is in effect after the feature freeze milestone. An issue labeled as a blocker practically blocks a release from happening. All blocker bugs shall be resolved before a release is created.

A fix for a bug that is granted *blocker* status can be merged to 'main' and included in the release all the way until the final release date.

Bugs of moderate severity and higher that have impact on all users are typically the candidates to be promoted to blocker bugs

Contributors and member of the release engineering team shall follow these guidelines for release blocker bugs:

- Only mark bugs as blockers if the software (Zephyr) must not be released with the bug present.
- All collaborators can add or remove blocking labels.
- Evaluate bugs as potential blockers based on their severity and prevalence.
- Provide detailed rationale whenever adding or removing a blocking label.
- Ensure all blockers have the milestone tagged.

- Release managers have final say on blocking status; contact them with any questions.

9.4.4 Release Milestones

Table 2: Release Milestones

Timeline	Checkpoint	Description	Owner
T-5M	Planning	Finalize dates for release, Assign release owner and agree on project wide goals for this release.	TSC
T-7W	Review target milestones	Finalize target milestones for features in flight.	Release Engineering
T-4W	Release Announcement	Release owner announces feature freeze and timeline for release.	Release Manager
T-3W	Feature Freeze (RC1)	No new features after RC1, ONLY stabilization and cosmetic changes, bug and doc fixes are allowed. New tests for existing features are also allowed.	Release Engineering
T-2W	2nd Release Candidate	No new features after RC2, ONLY stabilization and cosmetic changes, bug and doc fixes are allowed.	Release Manager
T-1W	Hard Freeze (RC3)	Only blocker bug fixes after RC3, documentation and changes to release notes are allowed. Release notes need to be complete by this checkpoint. Release Criteria is met.	Release Manager
T-0W	Release		Release Manager

9.4.5 Releases

The following syntax should be used for releases and tags in Git:

- Release [Major].[Minor].[Patch Level]
- Release Candidate [Major].[Minor].[Patch Level]-rc[RC Number]
- Tagging:
  - v[Major].[Minor].[Patch Level]-rc[RC Number]
  - v[Major].[Minor].[Patch Level]
  - v[Major].[Minor].99 - A tag applied to main branch to signify that work on v[Major].[Minor+1] has started. For example, v1.7.99 will be tagged at the start of v1.8 process. The tag corresponds to VERSION_MAJOR/VERSION_MINOR/PATCHLEVEL macros as defined for a work-in-progress main branch version. Presence of this tag allows generation of sensible output for “git describe” on main branch, as typically used for automated builds and CI tools.

Long Term Support (LTS)

Long-term support releases are designed to be supported and maintained for an extended period and is the recommended release for products and the auditable branch used for certification.

An LTS release is defined as:

- **Product focused**

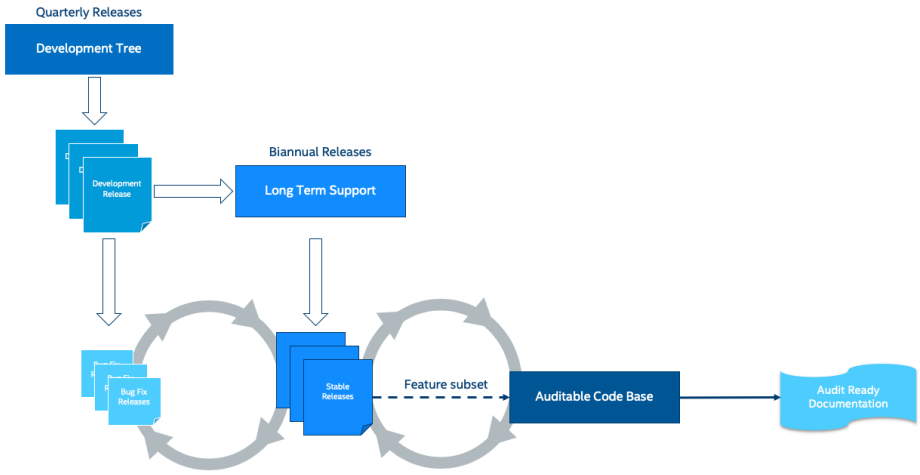


Fig. 2: Zephyr Code and Releases

- **Extended Stabilisation period**: Allow for more testing and bug fixing
- **Stable APIs**
- **Quality Driven Process**
- **Long Term**: Maintained for an extended period of time (at least 2.5 years) overlapping previous LTS release for at least half a year.

Product Focused Zephyr LTS is the recommended release for product makers with an extended support and maintenance which includes general stability and bug fixes, security fixes.

An LTS includes both mature and new features. API and feature maturity is documented and tracked. The footprint and scope of mature and stable APIs expands as we move from one LTS to the next giving users access to bleeding edge features and new hardware while keeping a stable foundation that evolves over time.

Extended Stabilisation Period Zephyr LTS development cycle differs from regular releases and has an extended stabilization period. Feature freeze of regular releases happens 3-4 weeks before the scheduled release date. The stabilization period for LTS is extended by 3 weeks with the feature freeze occurring 6-7 weeks before the anticipated release date. The time between code freeze and release date is extended in this case.

Stable APIs Zephyr LTS provides a stable and long-lived foundation for developing products. To guarantee stability of the APIs and the implementation of such APIs it is required that any release software that makes the core of the OS went through the Zephyr API lifecycle and stabilized over at least 2 releases. This guarantees that we release many of the highlighted and core features with mature and well-established implementations with stable APIs that are supported during the lifetime of the release LTS.

- API Freeze (LTS - 2)
  - All stable APIs need to be frozen 2 releases before an LTS. APIs can be extended with additional features, but the core implementation is not modified. This is valid for the following subsystems for example:
    - Device Drivers (i2c.h, spi.h)...
    - Kernel (k_*):
    - OS services (logging,debugging, ..)

    * DTS: API and bindings stability
    * Kconfig
  - New APIs for experimental features can be added at any time as long as they are standalone and documented as experimental or unstable features/APIs.
- Feature Freeze (LTS - 1) - No new features or overhaul/restructuring of code covering major LTS features.
  - Kernel + Base OS
  - Additional advertised LTS features
  - Auxiliary features on top of and/or extending the base OS and advertised LTS features can be added at any time and should be marked as experimental if applicable

Quality Driven Process The Zephyr project follows industry standards and processes with the goal of providing a quality oriented releases. This is achieved by providing the following products to track progress, integrity and quality of the software components provided by the project:

- Compliance with published coding guidelines, style guides and naming conventions and documentation of deviations.
- Static analysis reports
  - Regular static analysis on the complete tree using available commercial and open-source tools, and documentation of deviations and false positives.
- Documented components and APIS
- Requirements Catalog
- Verification Plans
- Verification Reports
- Coverage Reports
- Requirements Traceability Matrix (RTM)
- SPDX License Reports

Each release is created with the above products to document the quality and the state of the software when it was released.

Long Term Support and Maintenance A Zephyr LTS release is published every 2 years and is branched and maintained independently from the main tree for at least 2.5 years after it was released. Support and maintenance for an LTS release stops at least half a year after the following LTS release is published.

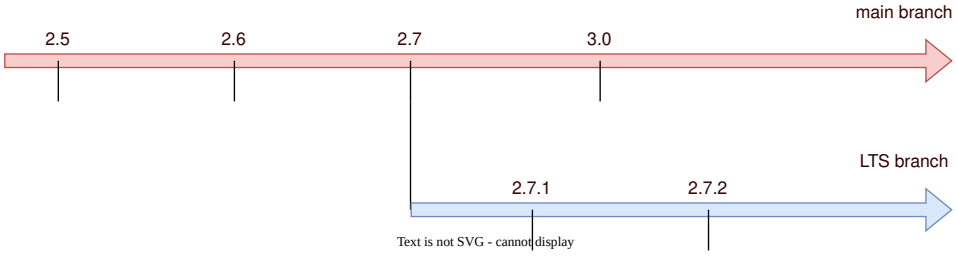


Fig. 3: Long Term Support Release

Changes and fixes flow in both directions. However, changes from main branch to an LTS branch will be limited to fixes that apply to both branches and for existing features only.

All fixes for an LTS branch that apply to the mainline tree shall be submitted to mainline tree as well.

Auditable Code Base

An auditable code base is to be established from a defined subset of Zephyr OS features and will be limited in scope. The LTS, development tree, and the auditable code bases shall be kept in sync after the audit branch is created, but with a more rigorous process in place for adding new features into the audit branch used for certification.

This process will be applied before new features move into the auditable code base.

The initial and subsequent certification targets will be decided by the Zephyr project governing board.

Processes to achieve selected certification will be determined by the Security and Safety Working Groups and coordinated with the TSC.

9.4.6 Hardware Support Tiers

Tier 0: Emulation Platforms

- Tests are both built and run in these platforms in CI, and therefore runtime failures can block Pull Requests.
- Supported by the Zephyr project itself, commitment to fix bugs in releases.
- One Tier 0 platform is required for each new architecture.
- Bugs reported against platforms of this tier are to be evaluated and treated as a general bug in Zephyr and should be dealt with the highest priority.

Tier 1: Supported Platforms

- Commitment from a specific team to run tests using twister device testing for the “Zephyr compatibility test suite” (details TBD) on a regular basis using open-source and publicly available drivers.
- Commitment to fix bugs in time for releases. Not supported by “Zephyr Project” itself.
- General availability for purchase
- Bugs reported against platforms of this tier are to be evaluated and treated as a general bug in Zephyr and should be dealt with medium to high priority.

Tier 2: Community Platforms

- Platform implementation is available in upstream, no commitment to testing, may not be generally available.
- Has a dedicated maintainer who commits to respond to issues / review patches.
- Bugs reported against platforms of this tier are NOT considered as a general bug in Zephyr.

Tier 3: Deprecated and unsupported Platforms

- Platform implementation is available, but no owner or unresponsive owner.
- No commitment to support is available.
- May be removed from upstream if no one works to bring it up to tier 2 or better.
- Bugs reported against platforms of this tier are NOT considered as a general bug in Zephyr.

9.4.7 Release Procedure

This section documents the Release manager responsibilities so that it serves as a knowledge repository for Release managers.

Release Checklist

Each release has a GitHub issue associated with it that contains the full checklist. After a release is complete, a checklist for the next release is created.

Tagging

The final release and each release candidate shall be tagged using the following steps:

> **Note**
>
> Tagging needs to be done via explicit git commands and not via GitHub's release interface. The GitHub release interface does not generate annotated tags (it generates 'lightweight' tags regardless of release or pre-release). You should also upload your gpg public key to your GitHub account, since the instructions below involve creating signed tags. However, if you do not have a gpg public key you can opt to remove the `-s` option from the commands below.

Release Candidate

> **Note**
>
> This section uses tagging 1.11.0-rc1 as an example, replace with the appropriate release candidate version.

1. Update the version variables in the VERSION file located in the root of the Git repository to match the version for this release candidate. The `EXTRAVERSION` variable is used to identify the rc[RC Number] value for this candidate:

   ```
   EXTRAVERSION = rc1
   ```

2. Post a PR with the updated VERSION file using `release: Zephyr 1.11.0-rc1` as the commit subject. Merge the PR after successful CI.

3. Tag and push the version, using an annotated tag:

   ```
   $ git pull
   $ git tag -s -m "Zephyr 1.11.0-rc1" v1.11.0-rc1
   $ git push git@github.com:zephyrproject-rtos/zephyr.git v1.11.0-rc1
   ```

4. Send an email to the mailing lists (`announce` and `devel`) with a link to the release

Final Release

> **Note**
>
> This section uses tagging 1.11.0 as an example, replace with the appropriate final release version.

When all final release criteria has been met and the final release notes have been approved and merged into the repository, the final release version will be set and repository tagged using the following procedure:

1. Update the version variables in the VERSION file located in the root of the Git repository. Set `EXTRAVERSION` variable to an empty string to indicate final release:

   ```
   EXTRAVERSION =
   ```

2. Post a PR with the updated VERSION file using `release: Zephyr 1.11.0` as the commit subject. Merge the PR after successful CI.

3. Tag and push the version, using two annotated tags:

   ```
   $ git pull
   $ git tag -s -m "Zephyr 1.11.0" v1.11.0
   $ git push git@github.com:zephyrproject-rtos/zephyr.git v1.11.0
   ```

4. Find the new `v1.11.0` tag at the top of the releases page and edit the release with the `Edit tag` button with the following:

   - Copy the overview of `docs/releases/release-notes-1.11.rst` into the release notes textbox and link to the full release notes file on docs.zephyrproject.org.

5. Send an email to the mailing lists (`announce` and `devel`) with a link to the release

9.5 Feature Tracking

For feature tracking we use Github labels to classify new features and enhancements. The following is the description of each category:

Enhancement
: Changes to existing features that are not considered a bug and would not block a release. This is an incremental enhancement to a feature that already exists in Zephyr.

Feature request
: A request for the implementation or inclusion of a new unit of functionality that is not part of any release plans yet, that has not been vetted, and needs further discussion and details.

Feature
: A committed and planned unit of functionality with a detailed design and implementation proposal and an owner. Features must go through an RFC process and must be vetted and discussed in the TSC before a target milestone is set.

Hardware Support
: A request or plan to port an existing feature or enhancement to a particular hardware platform. This ranges from porting Zephyr itself to a new architecture, SoC or board to adding an implementation of a peripheral driver API for an existing hardware platform.

Meta
: A label to group other GitHub issues that are part of a single feature or unit of work.

The following workflow should be used to process features:.

This is the formal way for asking for a new feature in Zephyr and indicating its importance to the project. Often, the requester may have a readiness and willingness to drive implementation of the feature in an upcoming release, and should assign the request to themselves. If not though, an owner will be assigned after evaluation by the TSC. A feature request can also have a companion RFC with more details on the feature and a proposed design or implementation.

- Label new features requests as `feature-request`
- The TSC discusses new `feature-request` items regularly and triages them. Items are examined for similarity with existing features, how they fit with the project goals and other timeline considerations. The priority is determined as follows:
  - High = Next milestone
  - Medium = As soon as possible
  - Low = Best effort
- After the initial discussion and triaging, the label is moved from `feature-request` to `feature` with the target milestone and an assignee.

All items marked as `feature-request` are non-binding and those without an assignee are open for grabs, meaning that they can be picked up and implemented by any project member or the community. You should contact an assigned owner if you'd like to discuss or contribute to that feature's implementation

9.5.1 Roadmap and Release Plans

Project roadmaps and release plans are both important tools for the project, but they have very different purposes and should not be confused. A project roadmap communicates the high-level overview of a project's strategy, while a release plan is a tactical document designed to capture and track the features planned for upcoming releases.

- The project roadmap communicates the why; a release plan details the what
- A release plan spans only a few months; a product roadmap might cover a year or more

Project Roadmap

The project roadmap should serve as a high-level, visual summary of the project's strategic objectives and expectations.

If built properly, the roadmap can be a valuable tool for several reasons. It can help the project present its plan in a compelling way to existing and new stakeholders, to help recruit new members and it can be a helpful resource the team and community can refer to throughout the project's development, to ensure they are still executing according to plan.

As such, the roadmap should contain only strategic-level details, major project themes, epics, and goals.

Release Plans

The release plan comes into play when the project roadmap's high-level strategy is translated into an actionable plan built on specific features, enhancements, and fixes that need to go into a specific release or milestone.

The release plan communicates those features and enhancements slated for your project' next release (or the next few releases). So it acts as more of a project plan, breaking the big ideas down into smaller projects the community and main stakeholders of the project can make progress on.

Items labeled as `features` are short or long term release items that shall have an assignee and a milestone set.

9.6 Code Flow and Branches

9.6.1 Introduction

The zephyr Git repository has three types of branches:

main
: Which contains the latest state of development

collab-*
: Collaboration branches that are used for shared development of new features to be introduced into the main branch when ready. Creating a new collaboration branch requires a justification and TSC approval. Collaboration branches shall be based off the main branch and any changes developed in the collab branch shall target the main development branch. For released versions of Zephyr, the introduction of fixes and new features, if approved by the TSC, shall be done using backport pull requests.

vx.y-branch
: Branches which track maintenance releases based on a major release

Development in collaboration branches before features go to mainline allows teams to work independently on a subsystem or a feature, improves efficiency and turnaround time, and encourages collaboration and streamlines communication between developers.

Changes submitted to a collaboration branch can evolve and improve incrementally in a branch, before they are submitted to the mainline tree for final integration.

By dedicating an isolated branch to complex features, it's possible to initiate in-depth discussions around new additions before integrating them into the official project.

Collaboration branches are ephemeral and shall be removed once the collaboration work has been completed. When a branch is requested, the proposal should include the following:

- Define exit criteria for merging the collaboration branch changes back into the main branch.
- Define a timeline for the expected life cycle of the branch. It is recommended to select a Zephyr release to set the timeline. Extensions to this timeline requires TSC approval.

9.6.2 Roles and Responsibilities

Collaboration branch owners have the following responsibilities:

- Use the infrastructure and tools provided by the project (GitHub, Git)
- All changes to collaboration branches shall come in form of github pull requests.
- Force pushing a collaboration branch is only allowed when rebasing against the main branch.
- Review changes coming from team members and request review from branch owners when submitting changes.
- Keep the branch in sync with upstream and update on a regular basis.
- Push changes frequently to upstream using the following methods:
  - GitHub pull requests: for example, when reviews have not been done in the local branch (one-man branch).

  - Merge requests: When a set of changes has been done in a local branch and has been reviewed and tested in a collaboration branch.

9.7 Modifying Contributions made by other developers

9.7.1 Scenarios

Zephyr contributors and collaborators are encouraged to assist as reviewers in pull requests, so that patches may be approved and merged to Zephyr's main branch as part of the original pull requests. The authors of the pull requests are responsible for amending their original commits following the review process.

There are occasions, however, when a contributor might need to modify patches included in pull requests that are submitted by other Zephyr contributors. For instance, this is the case when:

- a developer cherry-picks commits submitted by other contributors into their own pull requests in order to:
  - integrate useful content which is part of a stale pull request, or
  - get content merged to the project's main branch as part of a larger patch
- a developer pushes to a branch or pull request opened by another contributor in order to:
  - assist in updating pull requests in order to get the patches merged to the project's main branch
  - drive stale pull requests to completion so they can be merged

9.7.2 Accepted policies

A developer who intends to cherry-pick and potentially modify patches sent by another contributor shall:

- clarify in their pull request the reason for cherry-picking the patches, instead of assisting in getting the patches merged in their original pull request, and
- invite the original author of the patches to their pull request review.

A developer who intends to force-push to a branch or pull request of another Zephyr contributor shall clarify in the pull request the reason for pushing and for modifying the existing patches (e.g. stating that it is done to drive the pull request review to completion, when the pull request author is not able to do so).

> **Note**
>
> Developers should try to limit the above practice to pull requests identified as *stale*. Read about how to identify pull requests as stale in *development processes and tools*

If the original patches are substantially modified, the developer can either:

- (preferably) reach out to the original author and request them to acknowledge that the modified patches may be merged while having the original sign-off line and author identity, or
- submit the modified patches as their *own* work (i.e. with their *own* sign-off line and author identity). In this case, the developer shall identify in the commit message(s) the original source the submitted work is based on (mentioning, for example, the original PR number).

Note

Contributors should uncheck the box *"Allow Edits By Maintainers"* to indicate that they do not wish their patches to be amended, inside their original branch or pull request, by other Zephyr developers.

9.8 Development Environment and Tools

9.8.1 Code Review

GitHub is intended to provide a framework for reviewing every commit before it is accepted into the code base. Changes, in the form of Pull Requests (PR) are uploaded to GitHub but don't actually become a part of the project until they've been reviewed, passed a series of checks (CI), and are approved by maintainers. GitHub is used to support the standard open source practice of submitting patches, which are then reviewed by the project members before being applied to the code base.

Pull requests should be appropriately *labeled*, and linked to any relevant *bug or feature tracking issues* .

The Zephyr project uses GitHub for code reviews and Git tree management. When submitting a change or an enhancement to any Zephyr component, a developer should use GitHub. GitHub Actions automatically assigns a responsible reviewer on a component basis, as defined in the MAINTAINERS.yml file stored with the code tree in the Zephyr project repository. A limited set of release managers are allowed to merge a pull request into the main branch once reviews are complete.

Give reviewers time to review before code merge

The Zephyr project is a global project that is not tied to a certain geography or timezone. We have developers and contributors from across the globe. When changes are proposed using pull request, we need to allow for a minimal review time to give developers and contributors the opportunity to review and comment on changes. There are different categories of changes and we know that some changes do require reviews by subject matter experts and owners of the subsystem being changed. Many changes fall under the "trivial" category that can be addressed with general reviews and do not need to be queued for a maintainer or code-owner review. Additionally, some changes might require further discussions and a decision by the TSC or the Security working group. To summarize the above, the diagram below proposes minimal review times for each category:

Workflow

- An author of a change can suggest in his pull-request which category a change should belong to. A project maintainers or TSC member monitoring the inflow of changes can change the label of a pull request by adding a comment justifying why a change should belong to another category.
- The project will use the label system to categorize the pull requests.
- Changes should not be merged before the minimal time has expired.

Categories/Labels

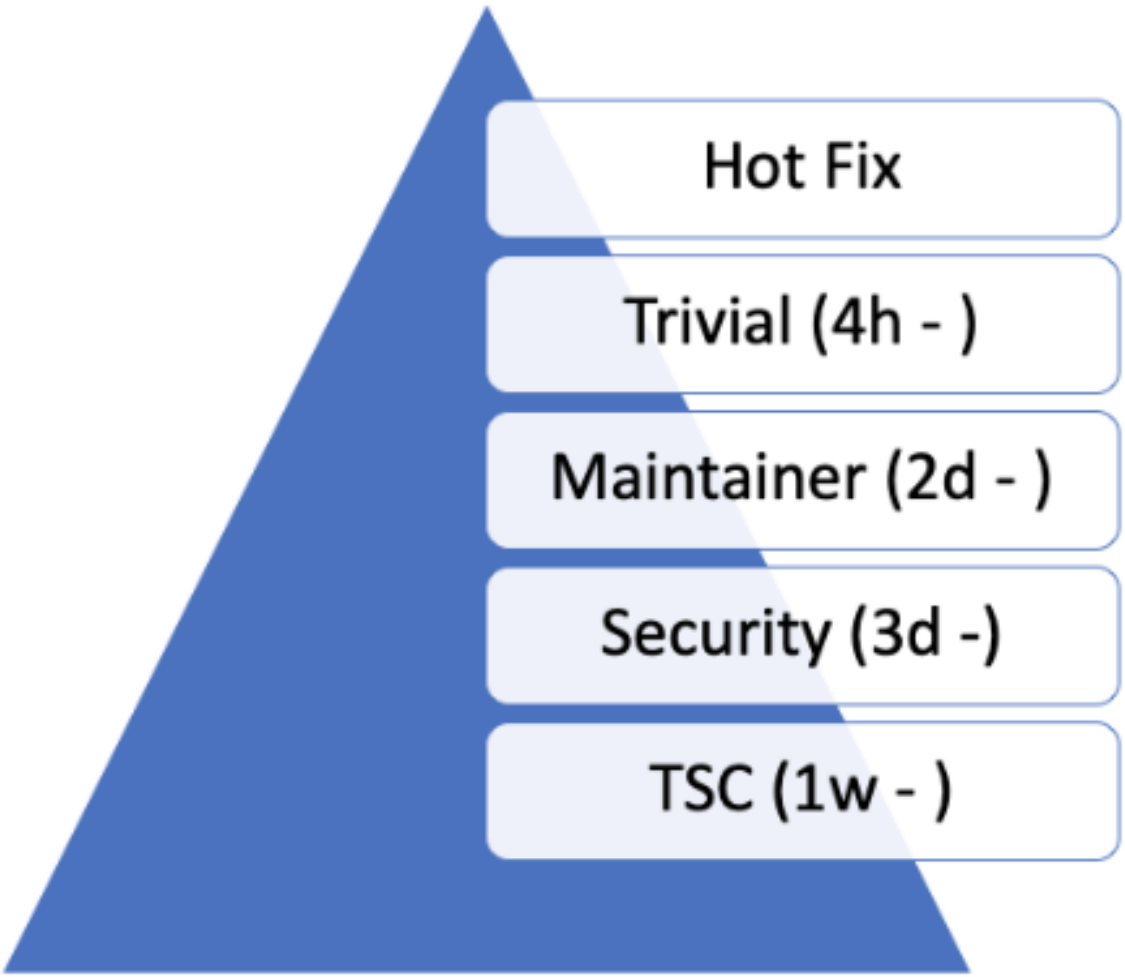


Fig. 4: Pull request classes

Hotfix Any change that is a fix to an issue that blocks developers from doing their daily work, for example CI breakage, Test breakage, Minor documentation fixes that impact the user experience.

Such fixes can be merged at any time after they have passed CI checks. Depending on the fix, severity, and availability of someone to review them (other than the author) they can be merged with justification without review by one of the project owners.

Trivial Trivial changes are those that appear obvious enough and do not require maintainer or code-owner involvement. Such changes should not change the logic or the design of a subsystem or component. For example a trivial change can be:

- Documentation changes
- Configuration changes
- Minor Build System tweaks
- Minor optimization to code logic without changing the logic
- Test changes and fixes
- Sample modifications to support additional configuration or boards etc.

Maintainer Any changes that touch the logic or the original design of a subsystem or component will need to be reviewed by the code owner or the designated subsystem maintainer. If the code changes is initiated by a contributor or developer other than the owner the pull request needs to be assigned to the code owner who will have to drive the pull request to a mergeable state by giving feedback to the author and asking for more reviews from other developers.

Security Changes that appear to have an impact to the overall security of the system need to be reviewed by a security expert from the security working group.

TSC and Working Groups Changes that introduce new features or functionality or change the way the overall system works need to be reviewed by the TSC or the responsible Working Group. For example for *breaking API changes*, the proposal needs to be presented in the Architecture meeting so that the relevant stakeholders are made aware of the change.

A Pull-Request should have an Assignee

- An assignee to a pull request should not be the same as the author of the pull-request
- An assignee to a pull request is responsible for driving the pull request to a mergeable state
- An assignee is responsible for dismissing stale reviews and seeking reviews from additional developers and contributors
- Pull requests should not be merged without an approval by the assignee.

Pull Request should not be merged by author without review

All pull requests need to be reviewed and should not be merged by the author without a review. The following exceptions apply:

- Hot fixes: Fixing CI issues, reverts, and system breakage
- Release related changes: Changing version file, applying tags and release related activities without any code changes.

Developers and contributors should always seek review, however there are cases when reviewers are not available and there is a need to get a code change into the tree as soon as possible.

Reviewers shall not 'Request Changes' without comments or justification

Any change requests (-1) on a pull request have to be justified. A reviewer should avoid blocking a pull-request with no justification. If a reviewer feels that a change should not be merged without their review, then: Request change of the category: for example:

- Trivial -> Maintainer
- Assign Pull Request to yourself, this will mean that a pull request should not be merged without your approval.

Pull Requests should have at least 2 approvals before they are merged

A pull-request shall be merged only with two positive reviews (approval). Beside the person merging the pull-request (merging != approval), two additional approvals are required to be able to merge a pull request. The person merging the request can merge without approving or approve and merge to get to the 2 approvals required.

Reviewers should keep track of pull requests they have provided feedback to

If a reviewer has requested changes in a pull request, he or she should monitor the state of the pull request and/or respond to mention requests to see if his feedback has been addressed. Failing to do so, negative reviews shall be dismissed by the assignee or an owner of the repository. Reviews will be dismissed following the criteria below:

- The feedback or concerns were visibly addressed by the author
- The reviewer did not revisit the pull request after 2 week and multiple pings by the author
- The review is unrelated to the code change or asking for unjustified structural changes such as:
  - Split the PR
  - Can you fix this unrelated code that happens to appear in the diff
  - Can you fix unrelated issues
  - Etc.

Closing Stale Issues and Pull Requests

- The Pull requests and issues sections on Github are NOT discussion forums. They are items that we need to execute and drive to closure. Use the mailing lists for discussions.
- In case of both issues and pull-requests the original poster needs to respond to questions and provide clarifications regarding the issue or the change. After one week without a response to a request, a second attempt to elicit a response from the contributor will be made. After one more week without a response the item may be closed (draft and DNM tagged pull requests are excluded).

9.8.2 Continuous Integration

All changes submitted to GitHub are subject to tests that are run on emulated platforms and architectures to identify breakage and regressions that can be immediately identified. Testing using Twister additionally performs build tests of all boards and platforms. Documentation changes are also verified through review and build testing to verify doc generation will be successful.

Any failures found during the CI test run will result in a negative review assigned automatically by the CI system. Developers are expected to fix issues and rework their patches and submit again.

The CI infrastructure currently runs the following tests:

- Run `checkpatch` for code style issues (can vote -1 on errors; see note)
- Gitlint: Git commit style based on project requirements
- License Check: Check for conflicting licenses
- Run `twister` script
  - Run kernel tests in QEMU (can vote -1 on errors)
  - Build various samples for different boards (can vote -1 on errors)
- Verify documentation builds correctly.

Note

`checkpatch` is a Perl script that uses regular expressions to extract information that requires a C language parser to process accurately. As such it sometimes issues false positives. Known cases include constructs like:

```
static uint8_t __aligned(PAGE_SIZE) page_pool[PAGE_SIZE * POOL_PAGES];
IOPCTL_Type *base = config->base;
```

Both lines produce a diagnostic regarding spaces around the `*` operator: the first is misidentified as a pointer type declaration that would be correct as `PAGE_SIZE *POOL_PAGES` while the second is misidentified as a multiplication expression that would be correct as `IOPCTL_Type * base`.

Maintainers can override the -1 in cases where the CI infrastructure gets the wrong answer.

9.8.3 Labeling issues and pull requests in GitHub

The project uses GitHub issues and pull requests (PRs) to track and manage daily and long-term work and contributions to the Zephyr project. We use GitHub **labels** to classify and organize these issues and PRs by area, type, priority, and more, making it easier to find and report on relevant items.

All GitHub issues or pull requests must be appropriately labeled. Issues and PRs often have multiple labels assigned, to help classify them in the different available categories. When reviewing a PR, if it has missing or incorrect labels, maintainers shall fix it.

This saves us all time when searching, reduces the chances of the PR or issue being forgotten, speeds up reviewing, avoids duplicate issue reports, etc.

These are the labels we currently have, grouped by applicability:

Labels applicable to issues only

Label	Description
priority: {high|medium|l	To classify the impact and importance of a bug or *feature*. Note: Issue priorities are generally set or changed during the bug-triage or TSC meetings.
Regression	Something, which was working, but does not anymore (bug subtype).
Enhancement	Changes/Updates/Additions to existing *features*.
Feature request	A request for a new *feature*.
Feature	A *planned feature* with a milestone.
Hardware Support	Covers porting an existing feature (including Zephyr itself) to new hardware.
Duplicate	This issue is a duplicate of another issue (please specify).
Good first issue	Good for a first time contributor to take.
Release Notes	Issues that need to be mentioned in release notes as known issues with additional information.

Any issue must be classified and labeled as either *Bug*, *Enhancement*, *RFC*, *Feature*, *Feature Request* or *Hardware Support*. More information on how feature requests are handled and become features can be found in *Feature Tracking*.

Labels applicable to pull requests only

The issue or PR describes a change to a stable API.

Label	Description
Hotfix	Fix for an issue blocking development.
Trivial	Simple changes that can have shorter review time and be reviewed by anyone, i.e. typos, straightforward one-liner bug fixes, etc.
Maintainer	Maintainer review required.
Security Review	To be reviewed by a security expert.
DNM	This PR should not be merged (Do Not Merge). For work in progress, GitHub "draft" PRs are preferred.
Needs review	The PR needs attention from the maintainers.
Backport	The PR is a backport or should be backported.
Licensing	The PR has licensing issues which require a licensing expert to review it.

Note

For all labels applicable to PRs: Please note that the label, together with PR complexity, affects how long a merge should be held to ensure proper review. See *review process* for details.

Labels applicable to both pull requests and issues

Label	Description
area: {area-name}	Indicates Zephyr subsystems (e.g, *area: Kernel*, *area: I2C*, *area: Memory Management*), project functions (e.g., *area: Debugging*, *area: Documentation*, *area: Process*), or other categories (e.g., *area: Coding Style*, *area: MISRA-C*) affected by the bug or the pull request. An area maintainer should be able to filter by an area label and find all issues and PRs which relate to that area.
platform: {platfor name}	An issue or PR which affects only a particular platform.
dev-review	The issue is to be discussed in the following dev-review if time permits.
TSC	TSC stands for Technical Steering Committee. The issue is to be discussed in the following TSC meeting if time permits.
Breaking API Change	The issue or PR describes a breaking change to a stable API. See additional information in *Introducing breaking API changes*.
bug	The issue is a bug, or the PR is fixing a bug.
Coverity	A Coverity detected issue or its fix.
Waiting for response	The Zephyr developers are waiting for the submitter to respond to a question, or address an issue.
Blocked	Blocked by another PR or issue.
Stale	An issue or a PR which seems abandoned, and requires attention by the author.
In progres	For PRs: is work in progress and should not be merged yet. For issues: Is being worked on.
RFC	The author would like input from the community. For a PR it should be considered a draft.
LTS	Long term release branch related.
EXT	Related to an external component.

9.9 Bug Reporting

To maintain traceability and relation between proposals, changes, features, and issues, it is recommended to cross-reference source code commits with the relevant GitHub issues and vice versa. Any changes that originate from a tracked feature or issue should contain a reference to the feature by mentioning the corresponding issue or pull-request identifiers.

At any time it should be possible to establish the origin of a change and the reason behind it by following the references in the code.

9.9.1 Reporting a regression issue

It could happen that the issue being reported is identified as a regression, as the use case is known to be working on earlier commit or release. In this case, providing directly the guilty commit when submitting the bug gains a lot of time in the eventual bug fixing.

To identify the commit causing the regression, several methods could be used, but tree bisecting method is an efficient one that doesn't require deep code expertise and can be used by every one.

For this, git bisect is the recommended tool.

Recommendations on the process:

- Run `west update` on each bisection step.
- Once the bisection is over and a culprit identified, verify manually the result.

9.10 Communication and Collaboration

The Zephyr Discord Server is the primary chat forum used by Zephyr developers, contributors, and users.

The Zephyr project mailing lists are used as an additional communication tool by project members, contributors, and the community. There are specialized mailing lists for specific interests. Several lists are public and open. Mailing lists are always available for use in situations where Discord is unavailable or an unsuitable forum.

In general, bug reports and other issues should be reported as GitHub Issues and not broadcasted to the mailing list. The same applies to code reviews.

9.11 Code Documentation

9.11.1 API Documentation

Well documented APIs enhance the experience for developers and are an essential requirement for defining an API's success. Doxygen is a general purpose documentation tool that the zephyr project uses for documenting APIs. It generates either an on-line documentation browser (in HTML) and/or provides input for other tools that is used to generate a reference manual from documented source files. In particular, doxygen's XML output is used as an input when producing the Zephyr project's online documentation.

9.11.2 Reference to Requirements

APIs for the most part document the implementation of requirements or advertised features and can be traced back to features. We use the API documentation as the main interface to trace implementation back to documented features. This is done using custom _doxygen_ tags that reference requirements maintained somewhere else in a requirement catalogue.

9.11.3 Test Documentation

To help understand what each test does and which functionality it tests we also document all test code using the same tools and in the same context and generate documentation for all unit and integration tests maintained in the same environment. Tests are documented using references to the APIs or functionality they validate by creating a link back to the APIs and by adding a reference to the original requirements.

9.11.4 Documentation Guidelines

Test Code

The Zephyr project uses several test methodologies, the most common being the *Ztest framework*. Test documentation should only be done on the entry test functions (usually prefixed with test_) and those that are called directly by the Ztest framework. Those tests are going to appear in test reports and using their name and identifier is the best way to identify them and trace back to them from requirements.

Test documentation should not interfere with the actual API documentation and needs to follow a new structure to avoid confusion. Using a consistent naming scheme and following a well-defined structure we will be able to group this documentation in its own module and identify it uniquely when parsing test data for traceability reports. Here are a few guidelines to be followed:

- All test code documentation should be grouped under the `all_tests` doxygen group
- All test documentation should be under doxygen groups that are prefixed with tests_

The custom doxygen `@verify` directive signifies that a test verifies a requirement:

```
/**
* @brief Tests for the Semaphore kernel object
* @defgroup kernel_semaphore_tests Semaphore
* @ingroup all_tests
* @{
*/

...
/**
* @brief A brief description of the tests
* Some details about the test
* more details
*
* @verify{@req{1111}}
*/
void test_sema_thread2thread(void)
{
...
}
...

/**
* @}
*/
```

To get coverage of how an implementation or a piece of code satisfies a requirements, we use the `satisfy` alias in doxygen:

```
/**
* @brief Give a semaphore.
*
* This routine gives @a sem, unless the semaphore is already at its maximum
* permitted count.
*
* @note Can be called by ISRs.
*
* @param sem Address of the semaphore.
*
* @satisfy{@req{015}}
*/
__syscall void k_sem_give(struct k_sem *sem);
```

To generate the matrix, you will first need to build the documentation, specifically you will need to build the doxygen XML output:

```
$ make doxygen
```

Parse the generated XML data from doxygen to generate the traceability matrix.

The Zephyr project defines a development process workflow using GitHub **Issues** to track feature, enhancement, and bug reports together with GitHub **Pull Requests** (PRs) for submitting and reviewing changes. Zephyr community members work together to review these Issues and PRs, managing feature enhancements and quality improvements of Zephyr through its regular releases, as outlined in the program management overview.

We can only manage the volume of Issues and PRs, by requiring timely reviews, feedback, and responses from the community and contributors, both for initial submissions and for followup questions and clarifications. Read about the project's *development processes and tools* and specifics about *review timelines* to learn about the project's goals and guidelines for our active developer community.

TSC Project Roles describes in detail the Zephyr project roles and associated permissions with respect to the development process workflow.

9.12 Terminology

- mainline: The main tree where the core functionality and core features are being developed.
- subsystem/feature branch: is a branch within the same repository. In our case, we will use the term branch also when referencing branches not in the same repository, which are a copy of a repository sharing the same history.
- upstream: A parent branch the source code is based on. This is the branch you pull from and push to, basically your upstream.
- LTS: Long Term Support

Chapter 10

Security

These documents describe the requirements, processes, and developer guidelines for ensuring security is addressed within the Zephyr project.

10.1 Zephyr Security Overview

10.1.1 Introduction

This document outlines the steps of the Zephyr Security Subcommittee towards a defined security process that helps developers build more secure software while addressing security compliance requirements. It presents the key ideas of the security process and outlines which documents need to be created. After the process is implemented and all supporting documents are created, this document is a top-level overview and entry point.

Overview and Scope

We begin with an overview of the Zephyr development process, which mainly focuses on security functionality.

In subsequent sections, the individual parts of the process are treated in detail. As depicted in Figure 1, these main steps are:

1. **Secure Development:** Defines the system architecture and development process that ensures adherence to relevant coding principles and quality assurance procedures.
2. **Secure Design:** Defines security procedures and implement measures to enforce them. A security architecture of the system and relevant sub-modules is created, threats are identified, and countermeasures designed. Their correct implementation and the validity of the threat models are checked by code reviews. Finally, a process shall be defined for reporting, classifying, and mitigating security issues.
3. **Security Certification:** Defines the certifiable part of the Zephyr RTOS. This includes an evaluation target, its assets, and how these assets are protected. Certification claims shall be determined and backed with appropriate evidence.

Intended Audience

This document is a guideline for the development of a security process by the Zephyr Security Subcommittee and the Zephyr Technical Steering Committee. It provides an overview of the Zephyr security process for (security) engineers and architects.

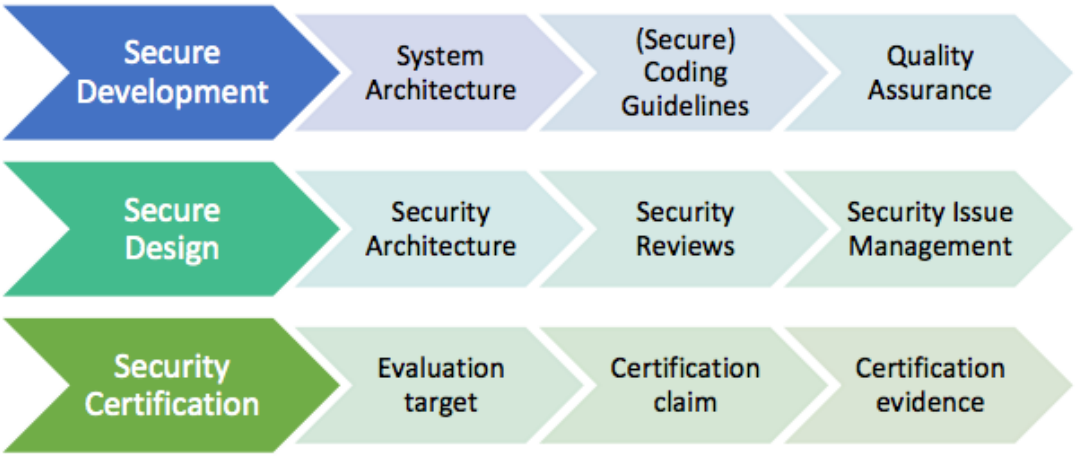


Fig. 1: Figure 1. Security Process Steps

Nomenclature

In this document, the keywords “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” are to be interpreted as described in **[?]**.

These words are used to define absolute requirements (or prohibitions), highly recommended requirements, and truly optional requirements. As noted in RFC-2119, “These terms are frequently used to specify behavior with security implications. The effects on security of not implementing a MUST or SHOULD, or doing something the specification says MUST NOT or SHOULD NOT be done may be very subtle. Document authors should take the time to elaborate the security implications of not following recommendations or requirements as most implementors will not have had the benefit of the experience and discussion that produced the specification.”

Security Document Update

This document is a living document. As new requirements, features, and changes are identified, they will be added to this document through the following process:

1. Changes will be submitted from the interested party(ies) via pull requests to the Zephyr documentation repository.
2. The Zephyr Security Subcommittee will review these changes and provide feedback or acceptance of the changes.
3. Once accepted, these changes will become part of the document.

10.1.2 Current Security Definition

This section recapitulates the current status of secure development within the Zephyr RTOS. Currently, focus is put on functional security and code quality assurance, although additional security features are scoped.

The three major security measures currently implemented are:

- **Security Functionality** with a focus on cryptographic algorithms and protocols. Support for cryptographic hardware is scoped for future releases. The Zephyr runtime architecture

is a monolithic binary and removes the need for dynamic loaders, thereby reducing the exposed attack surface.

- **Quality Assurance** is driven by using a development process that requires all code to be reviewed before being committed to the common repository. Furthermore, the reuse of proven building blocks such as network stacks increases the overall quality level and guarantees stable APIs. Static code analyses provide additional quality checks.
- **Execution Protection** including thread separation, stack and memory protection is currently available in the upstream Zephyr RTOS starting with version 1.9.0 (stack protection). Memory protection and thread separation were added in version 1.10.0 for X86 and in version 1.11.0 for ARM and ARC.

These topics are discussed in more detail in the following subsections.

Security Functionality

The security functionality in Zephyr hinges mainly on the inclusion of cryptographic algorithms, and on its monolithic system design.

The cryptographic features are provided through PSA Crypto, with mbedTLS as the underlying implementation. Applications leverage PSA Crypto APIs, ensuring a standardized and secure approach to cryptographic operations. mbedTLS, as the implementation of PSA Crypto, supports a wide range of cryptographic algorithms, making it suitable for various application requirements.

APIs for vendor specific cryptographic IPs in both hardware and software are planned, including secure key storage in the form of secure access modules (SAMs), Trusted Platform Modules (TPMs), and Trusted Execution Environments (TEEs).

The security architecture is based on a monolithic design where the Zephyr kernel and all applications are compiled into a single static binary. System calls are implemented as function calls without requiring context switches. Static linking eliminates the potential for dynamically loading malicious code.

Additional protection features are available in later releases. Stack protection mechanisms are provided to protect against stack overruns. In addition, applications can take advantage of thread separation features to split the system into privileged and unprivileged execution environments. Memory protection features provide the capability to partition system resources (memory, peripheral address space, etc.) and assign resources to individual threads or groups of threads. Stack, thread execution level, and memory protection constraints are enforced at the time of context switch.

Quality Assurance

The Zephyr project uses an automated quality assurance process. The goal is to have a process including mandatory code reviews, feature and issue management/tracking, and static code analyses.

Code reviews are documented and enforced using a voting system before getting checked into the repository by the responsible subsystem's maintainer. The main goals of the code review are:

- Verifying correct functionality of the implementation
- Increasing the readability and maintainability of the contributed source code
- Ensuring appropriate usage of string and memory functions
- Validation of the user input
- Reviewing the security relevant code for potential issues

The current coding principles focus mostly on coding styles and conventions. Functional correctness is ensured by the build system and the experience of the reviewer. Especially for security relevant code, concrete and detailed guidelines need to be developed and aligned with the developers (see: *Secure Coding*).

Static code analyses are run on the Zephyr code tree on a regular basis, see *Static Code Analysis*.

Bug and issue tracking and management is performed using Github. The term "survivability" was coined to cover pro-active security tasks such as security issue categorization and management. A problem identified as vulnerability is managed within Github security advisories.

Issues determined by static analyses should have more stringent reviews before they are closed as non-issues (at least another person educated in security processes need to agree on non-issue before closing).

A security subcommittee has been formed to develop a security process in more detail; this document is part of that process.

Execution Protection

Execution protection is supported and can be categorized into the following tasks:

- **Memory separation:** Memory will be partitioned into regions and assigned attributes based on the owner of that region of memory. Threads will only have access to regions they control.
- **Stack protection:** Stack guards would provide mechanisms for detecting and trapping stack overruns. Individual threads should only have access to their own stacks.
- **Thread separation:** Individual threads should only have access to their own memory resources. As threads are scheduled, only memory resources owned by that thread will be accessible. Topics such as program flow protection and other measures for tamper resistance are currently not in scope.

System Level Security (Ecosystem, ...)

System level security encompasses a wide variety of categories. Some examples of these would be:

- Secure/trusted boot
- Over the air (OTA) updates
- External Communication
- Device authentication
- Access control of onboard resources
  - Flash updating
  - Secure storage
  - Peripherals
- Root of trust
- Reduction of attack surface

Some of these categories are interconnected and rely on multiple pieces to be in place to produce a full solution for the application.

10.1.3 Secure Development Process

The development of secure code shall adhere to certain criteria. These include coding guidelines and development processes that can be roughly separated into two categories related to software quality and related to software security. Furthermore, a system architecture document shall be created and kept up-to-date with future development.

System Architecture

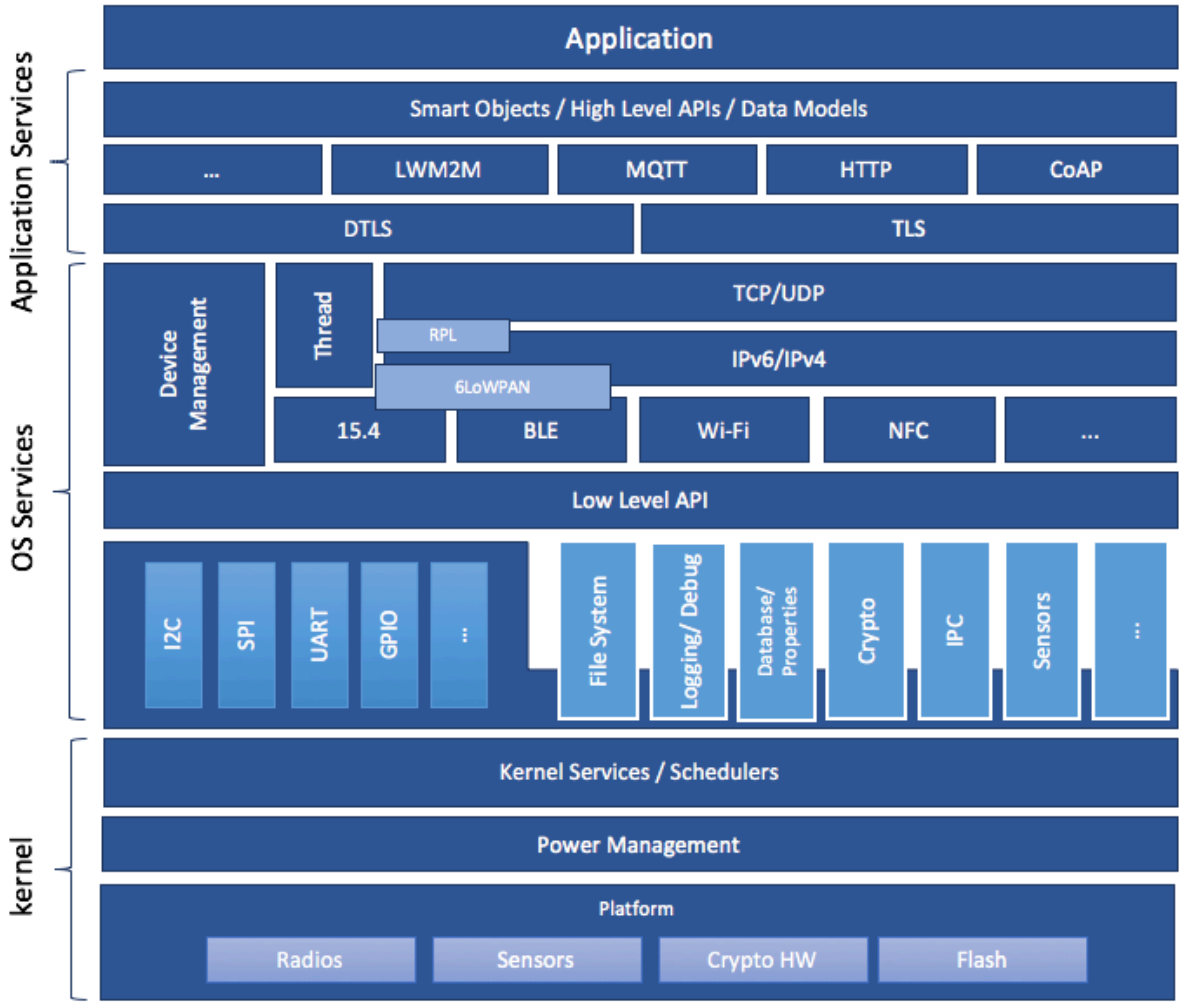


Fig. 2: Figure 2: Zephyr System Architecture

A high-level schematic of the Zephyr system architecture is given in Figure 2. It separates the architecture into an OS part (*kernel* + *OS Services*) and a user-specific part (*Application Services*). The OS part itself contains low-level, platform specific drivers and the generic implementation of I/O APIs, file systems, kernel-specific functions, and the cryptographic library.

A document describing the system architecture and design choices shall be created and kept up to date with future development. This document shall include the base architecture of the Zephyr OS and an overview of important submodules. For each of the modules, a dedicated architecture document shall be created and evaluated against the implementation. These documents shall serve as an entry point to new developers and as a basis for the security architecture. Please refer to the *Zephyr subsystem documentation* for detailed information.

Secure Coding

Designing an open software system such as Zephyr to be secure requires adhering to a defined set of design standards. These standards are included in the Zephyr Project documentation, specifically in its *Secure Coding* section. In [**?**], the following, widely accepted principles for protection mechanisms are defined to prevent security violations and limit their impact:

- **Open design** as a design principle incorporates the maxim that protection mechanisms cannot be kept secret on any system in widespread use. Instead of relying on secret, custom-tailored security measures, publicly accepted cryptographic algorithms and well established cryptographic libraries shall be used.
- **Economy of mechanism** specifies that the underlying design of a system shall be kept as simple and small as possible. In the context of the Zephyr project, this can be realized, e.g., by modular code [**?**] and abstracted APIs.
- **Complete mediation** requires that each access to every object and process needs to be authenticated first. Mechanisms to store access conditions shall be avoided if possible.
- **Fail-safe defaults** defines that access is restricted by default and permitted only in specific conditions defined by the system protection scheme, e.g., after successful authentication. Furthermore, default settings for services shall be chosen in a way to provide maximum security. This corresponds to the "Secure by Default" paradigm [**?**].
- **Separation of privilege** is the principle that two conditions or more need to be satisfied before access is granted. In the context of the Zephyr project, this could encompass split keys [**?**].
- **Least privilege** describes an access model in which each user, program and thread shall have the smallest possible subset of permissions in the system required to perform their task. This positive security model aims to minimize the attack surface of the system.
- **Least common mechanism** specifies that mechanisms common to more than one user or process shall not be shared if not strictly required. The example given in [**?**] is a function that should be implemented as a shared library executed by each user and not as a supervisor procedure shared by all users.
- **Psychological acceptability** requires that security features are easy to use by the developers in order to ensure its usage and the correctness of its application.

In addition to these general principles, the following points are specific to the development of a secure RTOS:

- **Complementary Security/Defense in Depth:** do not rely on a single threat mitigation approach. In case of the complementary security approach, parts of the threat mitigation are performed by the underlying platform. In case such mechanisms are not provided by the platform, or are not trusted, a defense in depth [**?**] paradigm shall be used.
- **Less commonly used services off by default**: to reduce the exposure of the system to potential attacks, features or services shall not be enabled by default if they are only rarely used (a threshold of 80% is given in [**?**]). For the Zephyr project, this can be realized using the configuration management. Each functionality and module shall be represented as a configuration option and needs to be explicitly enabled. Then, all features, protocols, and drivers not required for a particular use case can be disabled. The user shall be notified if low-level options and APIs are enabled but not used by the application.
- **Change management:** to guarantee a traceability of changes to the system, each change shall follow a specified process including a change request, impact analysis, ratification, implementation, and validation phase. In each stage, appropriate documentation shall be provided. All commits shall be related to a bug report or change request in the issue tracker. Commits without a valid reference shall be denied.

Based on these design principles and commonly accepted best practices, a secure development guide shall be developed, published, and implemented into the Zephyr development process.

Further details on this are given in the *Secure Design* section.

Quality Assurance

The quality assurance part encompasses the following criteria:

- **Adherence to the Coding Conventions** with respect to coding style, naming schemes of modules, functions, variables, and so forth. This increases the readability of the Zephyr code base and eases the code review. These coding conventions are enforced by automated scripts prior to check-in.
- **Adherence to Deployment Guidelines** is required to ensure consistent releases with a well-documented feature set and a trackable list of security issues.
- **Code Reviews** ensure the functional correctness of the code base and shall be performed on each proposed code change prior to check-in. Code reviews shall be performed by at least one independent reviewer other than the author(s) of the code change. These reviews shall be performed by the subsystem maintainers and developers on a functional level and are to be distinguished from security reviews as laid out in the *Secure Design* section. Refer to the *Project and Governance* documentation for more information.
- **Static Code Analysis** tools efficiently detect common coding mistakes in large code bases. All code shall be analyzed using an appropriate tool prior to merges into the main repository. This is not per individual commit, but is to be run on some interval on specific branches. It is mandatory to remove all findings or waive potential false-positives before each release. Waivers shall be documented centrally and in the form of a comment inside the source code itself. The documentation shall include the employed tool and its version, the date of the analysis, the branch and parent revision number, the reason for the waiver, the author of the respective code, and the approver(s) of the waiver. This shall as a minimum run on the main release branch and on the security branch. It shall be ensured that each release has zero issues with regard to static code analysis (including waivers). Refer to the *Project and Governance* documentation for more information.
- **Complexity Analyses** shall be performed as part of the development process and metrics such as cyclomatic complexity shall be evaluated. The main goal is to keep the code as simple as possible.
- **Automation:** the review process and checks for coding rule adherence are a mandatory part of the precommit checks. To ensure consistent application, they shall be automated as part of the precommit procedure. Prior to merging large pieces of code in from subsystems, in addition to review process and coding rule adherence, all static code analysis must have been run and issues resolved.

Release and Lifecycle Management

Lifecycle management contains several aspects:

- **Device management** encompasses the possibility to update the operating system and/or security related sub-systems of Zephyr enabled devices in the field.
- **Lifecycle management:** system stages shall be defined and documented along with the transactions between the stages in a system state diagram. For security reasons, this shall include locking of the device in case an attack has been detected, and a termination if the end of life is reached.
- **Release management** describes the process of defining the release cycle, documenting releases, and maintaining a record of known vulnerabilities and mitigations. Especially for certification purposes the integrity of the release needs to be ensured in a way that later manipulation (e.g., inserting of backdoors, etc.) can be easily detected.

- **Rights management and NDAs:** if required by the chosen certification, the confidentiality and integrity of the system needs to be ensured by an appropriate rights management (e.g., separate source code repository) and non-disclosure agreements between the relevant parties. In case of a repository shared between several parties, measures shall be taken that no malicious code is checked in.

These points shall be evaluated with respect to their impact on the development process employed for the Zephyr project.

10.1.4 Secure Design

In order to obtain a certifiable system or product, the security process needs to be clearly defined and its application needs to be monitored and driven. This process includes the development of security related modules in all of its stages and the management of reported security issues. Furthermore, threat models need to be created for currently known and future attack vectors, and their impact on the system needs to be investigated and mitigated. Please refer to the *Secure Coding* outlined in the Zephyr project documentation for detailed information.

The software security process includes:

- **Adherence to the Secure Development Coding** is mandatory to avoid that individual components breach the system security and to minimize the vulnerability of individual modules. While this can be partially achieved by automated tests, it is inevitable to investigate the correct implementation of security features such as countermeasures manually in security-critical modules.
- **Security Reviews** shall be performed by a security architect in preparation of each security-targeted release and each time a security-related module of the Zephyr project is changed. This process includes the validation of the effectiveness of implemented security measures, the adherence to the global security strategy and architecture, and the preparation of audits towards a security certification if required.
- **Security Issue Management** encompasses the evaluation of potential system vulnerabilities and their mitigation as described in *Security Issue Management*.

These criteria and tasks need to be integrated into the development process for secure software and shall be automated wherever possible. On system level, and for each security related module of the secure branch of Zephyr, a directly responsible security architect shall be defined to guide the secure development process.

Security Architecture

The general guidelines above shall be accompanied by an architectural security design on system- and module-level. The high level considerations include

- The identification of **security and compliance requirements**
- **Functional security** such as the use of cryptographic functions whenever applicable
- Design of **countermeasures** against known attack vectors
- Recording of security relevant **auditable events**
- Support for **Trusted Platform Modules (TPM)** and **Trusted Execution Environments (TEE)**
- Mechanisms to allow for **in-the-field updates** of devices using Zephyr
- Task scheduler and separation

The security architecture development is based on assets derived from the structural overview of the overall system architecture. Based on this, the individual steps include:

1. **Identification of assets** such as user data, authentication and encryption keys, key generation data (obtained from RNG), security relevant status information.
2. **Identification of threats** against the assets such as breaches of confidentiality, manipulation of user data, etc.
3. **Definition of requirements** regarding security and protection of the assets, e.g., countermeasures or memory protection schemes.

The security architecture shall be harmonized with the existing system architecture and implementation to determine potential deviations and mitigate existing weaknesses. Newly developed sub-modules that are integrated into the secure branch of the Zephyr project shall provide individual documents describing their security architecture. Additionally, their impact on the system level security shall be considered and documented.

Security Vulnerability Reporting

Please see *Security Vulnerability Reporting* for information on reporting security vulnerabilities.

Threat Modeling and Mitigation

The modeling of security threats against the Zephyr RTOS is required for the development of an accurate security architecture and for most certification schemes. The first step of this process is the definition of assets to be protected by the system. The next step then models how these assets are protected by the system and which threats against them are present. After a threat has been identified, a corresponding threat model is created. This model contains the asset and system vulnerabilities, as well as the description of the potential exploits of these vulnerabilities. Additionally, the impact on the asset, the module it resides in, and the overall system is to be estimated. This threat model is then considered in the module and system security architecture and appropriate countermeasures are defined to mitigate the threat or limit the impact of exploits.

In short, the threat modeling process can be separated into these steps (adapted from **[?]**):

1. Definition of assets
2. Application decomposition and creation of appropriate data flow diagrams (DFDs)
3. Threat identification and categorization using the **[?]** and **[?]** approaches
4. Determination of countermeasures and other mitigation approaches

This procedure shall be carried out during the design phase of modules and before major changes of the module or system architecture. Additionally, new models shall be created, or existing ones shall be updated whenever new vulnerabilities or exploits are discovered. During security reviews, the threat models and the mitigation techniques shall be evaluated by the responsible security architect.

From these threat models and mitigation techniques tests shall be derived that prove the effectiveness of the countermeasures. These tests shall be integrated into the continuous integration workflow to ensure that the security is not impaired by regressions.

Vulnerability Analyses

In order to find weak spots in the software implementation, vulnerability analyses (VA) shall be performed. Of special interest are investigations on cryptographic algorithms, critical OS tasks, and connectivity protocols.

On a pure software level, this encompasses

- **Penetration testing** of the RTOS on a particular hardware platform, which involves testing the respective Zephyr OS configuration and hardware as one system.

- **Side channel attacks** (timing invariance, power invariance, etc.) should be considered. For instance, ensuring **timing invariance** of the cryptographic algorithms and modules is required to reduce the attack surface. This applies to both the software implementations and when using cryptographic hardware.
- **Fuzzing tests** shall be performed on both exposed APIs and protocols.

The list given above serves primarily illustration purposes. For each module and for the complete Zephyr system (in general on a particular hardware platform), a suitable VA plan shall be created and executed. The findings of these analyses shall be considered in the security issue management process, and learnings shall be formulated as guidelines and incorporated into the secure coding guide.

If possible (as in case of fuzzing analyses), these tests shall be integrated into the continuous integration process.

10.1.5 Security Certification

One goal of creating a secure branch of the Zephyr RTOS is to create a certifiable system or certifiable submodules thereof. The certification scope and scheme are yet to be decided. However, many certifications such as Common Criteria [**?**] require evidence that the evaluation claims are indeed fulfilled, so a general certification process is outlined in the following. Based on the final choices for the certification scheme and evaluation level, this process needs to be refined.

Generic Certification Process

In general, the steps towards a certification or precertification (compare [**?**]) are:

1. The **definition of assets** to be protected within the Zephyr RTOS. Potential candidates are confidential information such as cryptographic keys, user data such as communication logs, and potentially IP of the vendor or manufacturer.
2. Developing a **threat model** and **security architecture** to protect the assets against exploits of vulnerabilities of the system. As a complete threat model includes the overall product including the hardware platform, this might be realized by a split model containing a precertified secure branch of Zephyr which the vendor could use to certify their Zephyr-enabled product.
3. Formulating an **evaluation target** that includes the **certification claims** on the security of the assets to be evaluated and certified, as well as assumptions on the operating conditions.
4. Providing **proof** that the claims are fulfilled. This includes consistent documentation of the security development process, etc.

These steps are partially covered in previous sections as well. In contrast to these sections, the certification process only requires to consider those components that shall be covered by the certification. The security architecture, for example, considers assets on system level and might include items not relevant for the certification.

Certification Options

For the security certification as such, the following options can be pursued:

1. **Abstract precertification of Zephyr as a pure software system:** this option requires assumptions on the underlying hardware platform and the final application running on top of Zephyr. If these assumptions are met by the hardware and the application, a full certification can be more easily achieved. This option is the most flexible approach but puts the largest burden on the product vendor.

2. **Certification of Zephyr on specific hardware platform without a specific application in mind:** this scenario describes the enablement of a secure platform running the Zephyr RTOS. The hardware manufacturer certifies the platform under defined assumptions on the application. If these are met, the final product can be certified with little effort.
3. **Certification of an actual product:** in this case, a full product including a specific hardware, the Zephyr RTOS, and an application is certified.

In all three cases, the certification scheme (e.g., FIPS 140-2 [**?**] or Common Criteria [**?**]), the scope of the certification (main-stream Zephyr, security branch, or certain modules), and the certification/assurance level need to be determined.

In case of partial certifications (options 1 and 2), assumptions on hardware and/or software are required for certifications. These can include [**?**]

- **Appropriate physical security** of the hardware platform and its environment.
- **Sufficient protection of storage and timing channels** on the hardware platform itself and all connected devices. (No mentioning of remote connections.)
- Only **trusted/assured applications** running on the device
- The device and its software stack is configured and operated by **properly trained and trusted individuals** with no malicious intent.

These assumptions shall be part of the security claim and evaluation target documents.

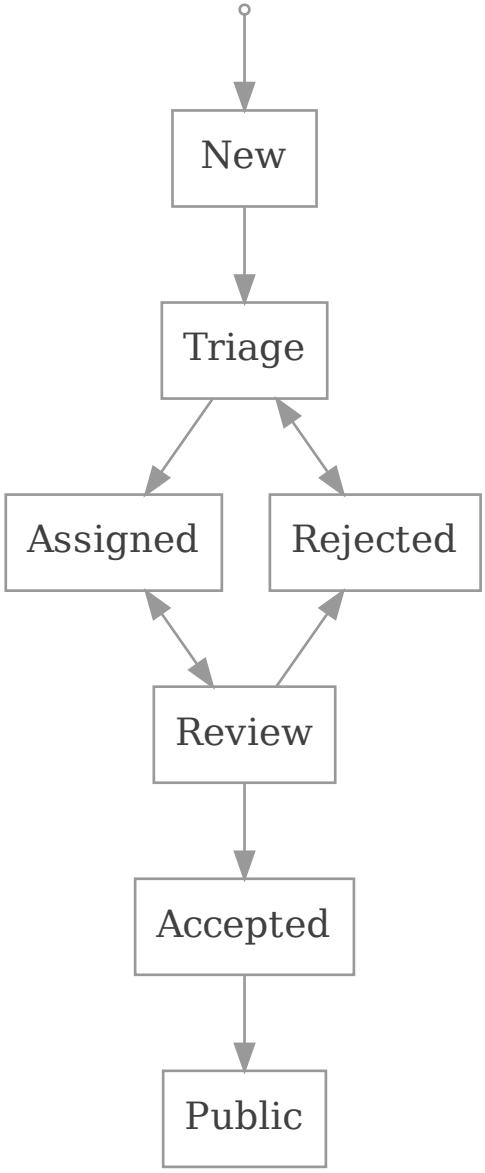
10.2 Security Vulnerability Reporting

10.2.1 Introduction

Vulnerabilities to the Zephyr project may be reported via email to the vulnerabilities@zephyrproject.org mailing list. These reports will be acknowledged and analyzed by the security response team within 1 week. Each vulnerability will be entered into the Zephyr Project security advisory GitHub. The original submitter will be granted permission to view the issues that they have reported.

10.2.2 Security Issue Management

Issues within this bug tracking system will transition through a number of states according to this diagram:



- New: This state represents new reports that have been entered directly by a reporter. When entered by the response team in response to an email, the issue shall be transitioned directly to Triage.
- Triage: This issue is awaiting Triage by the response team. The response team will analyze the issue, determine a responsible entity, assign it to that individual, and move the issue to the Assigned state. Part of triage will be to set the issue's priority.
- Assigned: The issue has been assigned, and is awaiting a fix by the assignee.
- Review: Once there is a Zephyr pull request for the issue, the PR link will be added to a comment in the issue, and the issue moved to the Review state.
- Accepted: Indicates that this issue has been merged into the appropriate branch within Zephyr.
- Public: The embargo period has ended. The issue will be made publicly visible, the associ-

ated CVE updated, and the vulnerabilities page in the docs updated to include the detailed information.

The security advisories created are kept private, due to the sensitive nature of security reports. The issues are only visible to certain parties:

- Members of the PSIRT mailing list
- the reporter
- others, as proposed and ratified by the Zephyr Security Subcommittee. In the general case, this will include:
  - The code owner responsible for the fix.
  - The Zephyr release owners for the relevant releases affected by this vulnerability.

The Zephyr Security Subcommittee shall review the reported vulnerabilities during any meeting with more than three people in attendance. During this review, they shall determine if new issues need to be embargoed.

The guideline for embargo will be based on: 1. Severity of the issue, and 2. Exploitability of the issue. Issues that the subcommittee decides do not need an embargo will be reproduced in the regular Zephyr project bug tracking system.

Security sensitive vulnerabilities shall be made public after an embargo period of at most 90 days. The intent is to allow 30 days within the Zephyr project to fix the issues, and 60 days for external parties building products using Zephyr to be able to apply and distribute these fixes.

Fixes to the code shall be made through pull requests PR in the Zephyr project github. Developers shall make an attempt to not reveal the sensitive nature of what is being fixed, and shall not refer to CVE numbers that have been assigned to the issue. The developer instead should merely describe what has been fixed.

The security subcommittee will maintain information mapping embargoed CVEs to these PRs (this information is within the Github security advisories), and produce regular reports of the state of security issues.

Each issue that is considered a security vulnerability shall be assigned a CVE number. As fixes are created, it may be necessary to allocate additional CVE numbers, or to retire numbers that were assigned.

10.2.3 Vulnerability Notification

Each Zephyr release shall contain a report of CVEs that were fixed in that release. Because of the sensitive nature of these vulnerabilities, the release shall merely include a list of CVEs that have been fixed. After the embargo period, the vulnerabilities page shall be updated to include additional details of these vulnerabilities. The vulnerability page shall give credit to the reporter(s) unless a reporter specifically requests anonymity.

The Zephyr project shall maintain a vulnerability-alerts mailing list. This list will be seeded initially with a contact from each project member. Additional parties can request to join this list by filling out the form at the Vulnerability Registry. These parties will be vetted by the project director to determine that they have a legitimate interest in knowing about security vulnerabilities during the embargo period.

Periodically, the security subcommittee will send information to this mailing list describing known embargoed issues, and their backport status within the project. This information is intended to allow them to determine if they need to backport these changes to any internal trees.

When issues have been triaged, this list will be informed of:

- The Zephyr Project security advisory link (GitHub).
- The CVE number assigned.

- The subsystem involved.
- The severity of the issue.

After acceptance of a PR fixing the issue (merged), in addition to the above, the list will be informed of:

- The association between the CVE number and the PR fixing it.
- Backport plans within the Zephyr project.

10.2.4 Backporting of Security Vulnerabilities

Each security issue fixed within zephyr shall be backported to the following releases:

- The current Long Term Stable (LTS) release.
- The most recent two releases.

The developer of the fix shall be responsible for any necessary backports, and apply them to any of the above listed release branches, unless the fix does not apply (the vulnerability was introduced after this release was made). All recommendations for *vulnerability fixes* apply for backport pull requests (and associated issues). Additionally, it is recommended that the developer privately informs the responsible release manager that the backport pull request and issue are addressing a vulnerability.

Backports will be tracked on the security advisory.

10.2.5 Need to Know

Due to the sensitive nature of security vulnerabilities, it is important to share details and fixes only with those parties that have a need to know. The following parties will need to know details about security vulnerabilities before the embargo period ends:

- Maintainers will have access to all information within their domain area only.
- The current release manager, and the release manager for historical releases affected by the vulnerability (see backporting above).
- The Project Security Incident Response (PSIRT) team will have full access to information. The PSIRT is made up of representatives from platinum members, and volunteers who do work on triage from other members.
- As needed, release managers and maintainers may be invited to attend additional security meetings to discuss vulnerabilities.

10.3 Secure Coding

Traditionally, microcontroller-based systems have not placed much emphasis on security. They have usually been thought of as isolated, disconnected from the world, and not very vulnerable, just because of the difficulty in accessing them. The Internet of Things has changed this. Now, code running on small microcontrollers often has access to the internet, or at least to other devices (that may themselves have vulnerabilities). Given the volume they are often deployed at, uncontrolled access can be devastating[1].

This document describes the requirements and process for ensuring security is addressed within the Zephyr project. All code submitted should comply with these principles.

Much of this document comes from [**?**].

[1] An attack resulted in a significant portion of DNS infrastructure being taken down.

10.3.1 Introduction and Scope

This document covers guidelines for the Zephyr Project, from a security perspective. Many of the ideas contained herein are captured from other open source efforts.

We begin with an overview of secure design as it relates to Zephyr. This is followed by a section on *Secure development knowledge*, which gives basic requirements that a developer working on the project will need to have. This section gives references to other security documents, and full details of how to write secure software are beyond the scope of this document. This section also describes vulnerability knowledge that at least one of the primary developers should have. This knowledge will be necessary for the review process described below this.

Following this is a description of the review process used to incorporate changes into the Zephyr codebase. This is followed by documentation about how security-sensitive issues are handled by the project.

Finally, the document covers how changes are to be made to this document.

10.3.2 Secure Coding

Designing an open software system such as Zephyr to be secure requires adhering to a defined set of design standards. In [**?**], the following, widely accepted principles for protection mechanisms are defined to help prevent security violations and limit their impact:

- **Open design** as a design guideline incorporates the maxim that protection mechanisms cannot be kept secret on any system in widespread use. Instead of relying on secret, custom-tailored security measures, publicly accepted cryptographic algorithms and well established cryptographic libraries shall be used.
- **Economy of mechanism** specifies that the underlying design of a system shall be kept as simple and small as possible. In the context of the Zephyr project, this can be realized, e.g., by modular code [**?**] and abstracted APIs.
- **Complete mediation** requires that each access to every object and process needs to be authenticated first. Mechanisms to store access conditions shall be avoided if possible.
- **Fail-safe defaults** defines that access is restricted by default and permitted only in specific conditions defined by the system protection scheme, e.g., after successful authentication. Furthermore, default settings for services shall be chosen in a way to provide maximum security. This corresponds to the “Secure by Default” paradigm [**?**].
- **Separation of privilege** is the principle that two conditions or more need to be satisfied before access is granted. In the context of the Zephyr project, this could encompass split keys [**?**].
- **Least privilege** describes an access model in which each user, program, and thread, shall have the smallest possible subset of permissions in the system required to perform their task. This positive security model aims to minimize the attack surface of the system.
- **Least common mechanism** specifies that mechanisms common to more than one user or process shall not be shared if not strictly required. The example given in [**?**] is a function that should be implemented as a shared library executed by each user and not as a supervisor procedure shared by all users.
- **Psychological acceptability** requires that security features are easy to use by the developers in order to ensure their usage and the correctness of its application.

In addition to these general principles, the following points are specific to the development of a secure RTOS:

- **Complementary Security/Defense in Depth**: do not rely on a single threat mitigation approach. In case of the complementary security approach, parts of the threat mitigation are

performed by the underlying platform. In case such mechanisms are not provided by the platform, or are not trusted, a defense in depth [**?**] paradigm shall be used.

- **Less commonly used services off by default**: to reduce the exposure of the system to potential attacks, features or services shall not be enabled by default if they are only rarely used (a threshold of 80% is given in [**?**]). For the Zephyr project, this can be realized using the configuration management. Each functionality and module shall be represented as a configuration option and needs to be explicitly enabled. Then, all features, protocols, and drivers not required for a particular use case can be disabled. The user shall be notified if low-level options and APIs are enabled but not used by the application.
- **Change management**: to guarantee a traceability of changes to the system, each change shall follow a specified process including a change request, impact analysis, ratification, implementation, and validation phase. In each stage, appropriate documentation shall be provided. All commits shall be related to a bug report or change request in the issue tracker. Commits without a valid reference shall be denied.

10.3.3 Secure development knowledge

Secure designer

The Zephyr project must have at least one primary developer who knows how to design secure software.

This requires understanding the following design principles, including the 8 principles from [**?**]:

- economy of mechanism (keep the design as simple and small as practical, e.g., by adopting sweeping simplifications)
- fail-safe defaults (access decisions shall deny by default, and projects' installation shall be secure by default)
- complete mediation (every access that might be limited must be checked for authority and be non-bypassable)
- open design (security mechanisms should not depend on attacker ignorance of its design, but instead on more easily protected and changed information like keys and passwords)
- separation of privilege (ideally, access to important objects should depend on more than one condition, so that defeating one protection system won't enable complete access. For example, multi-factor authentication, such as requiring both a password and a hardware token, is stronger than single-factor authentication)
- least privilege (processes should operate with the least privilege necessary)
- least common mechanism (the design should minimize the mechanisms common to more than one user and depended on by all users, e.g., directories for temporary files)
- psychological acceptability (the human interface must be designed for ease of use - designing for "least astonishment" can help)
- limited attack surface (the set of the different points where an attacker can try to enter or extract data)
- input validation with whitelists (inputs should typically be checked to determine if they are valid before they are accepted; this validation should use whitelists (which only accept known-good values), not blacklists (which attempt to list known-bad values)).

Vulnerability Knowledge

A "primary developer" in a project is anyone who is familiar with the project's code base, is comfortable making changes to it, and is acknowledged as such by most other participants in the

project. A primary developer would typically make a number of contributions over the past year (via code, documentation, or answering questions). Developers would typically be considered primary developers if they initiated the project (and have not left the project more than three years ago), have the option of receiving information on a private vulnerability reporting channel (if there is one), can accept commits on behalf of the project, or perform final releases of the project software. If there is only one developer, that individual is the primary developer.

At least one of the primary developers **must** know of common kinds of errors that lead to vulnerabilities in this kind of software, as well as at least one method to counter or mitigate each of them.

Examples (depending on the type of software) include SQL injection, OS injection, classic buffer overflow, cross-site scripting, missing authentication, and missing authorization. See the CWE/SANS top 25 or OWASP Top 10 for commonly used lists.

A free class from the nonprofit OpenSecurityTraining2 for C/C++ developers is available at OST2_1001. It teaches how to prevent, detect, and mitigate linear stack/heap buffer overflows, non-linear out of bound writes, integer overflows, and other integer issues. The follow-on class, OST2_1002, covers uninitialized data access, race conditions, use-after-free, type confusion, and information disclosure vulnerabilities.

Zephyr Security Subcommittee

There shall be a "Zephyr Security Subcommittee", responsible for enforcing this guideline, monitoring reviews, and improving these guidelines.

This team will be established according to the Zephyr Project charter.

10.3.4 Code Review

The Zephyr project shall use a code review system that all changes are required to go through. Each change shall be reviewed by at least one primary developer that is not the author of the change. This developer shall determine if this change affects the security of the system (based on their general understanding of security), and if so, shall request the developer with vulnerability knowledge, or the secure designer to also review the code. Any of these individuals shall have the ability to block the change from being merged into the mainline code until the security issues have been addressed.

10.3.5 Issues and Bug Tracking

The Zephyr project shall have an issue tracking system (such as GitHub) that can be used to record and track defects that are found in the system.

Because security issues are often sensitive, this issue tracking system shall have a field to indicate a security issue. Setting this field shall result in the issue only being visible to the Zephyr Security Subcommittee. In addition, there shall be a field to allow the Zephyr Security Subcommittee to add additional users that will have visibility to a given issue.

This embargo, or limited visibility, shall only be for a fixed duration, with a default being a project-decided value. However, because security considerations are often external to the Zephyr project itself, it may be necessary to increase this embargo time. The time necessary shall be clearly annotated in the issue itself.

The list of issues shall be reviewed at least once a month by the Zephyr Security Subcommittee. This review should focus on tracking the fixes, determining if any external parties need to be notified or involved, and determining when to lift the embargo on the issue. The embargo should **not** be lifted via an automated means, but the review team should avoid unnecessary delay in lifting issues that have been resolved.

10.3.6 Modifications to This Document

Changes to this document shall be reviewed by the Zephyr Security Subcommittee, and approved by consensus.

10.4 Sensor Device Threat Model

This document describes a threat model for an IoT sensor device. Spelling out a threat model helps direct development effort, and can be used to help prioritize these efforts as well.

This device contains a sensor of some type (for example temperature, or a pressure in a pipe), which sends this data to an SoC running a microcontroller. This microcontroller connects to a cloud service, and relays this sensor data to this service. The cloud service is also able to send configuration data to the device, as well as software update images. A general diagram can be seen in Figure 1:

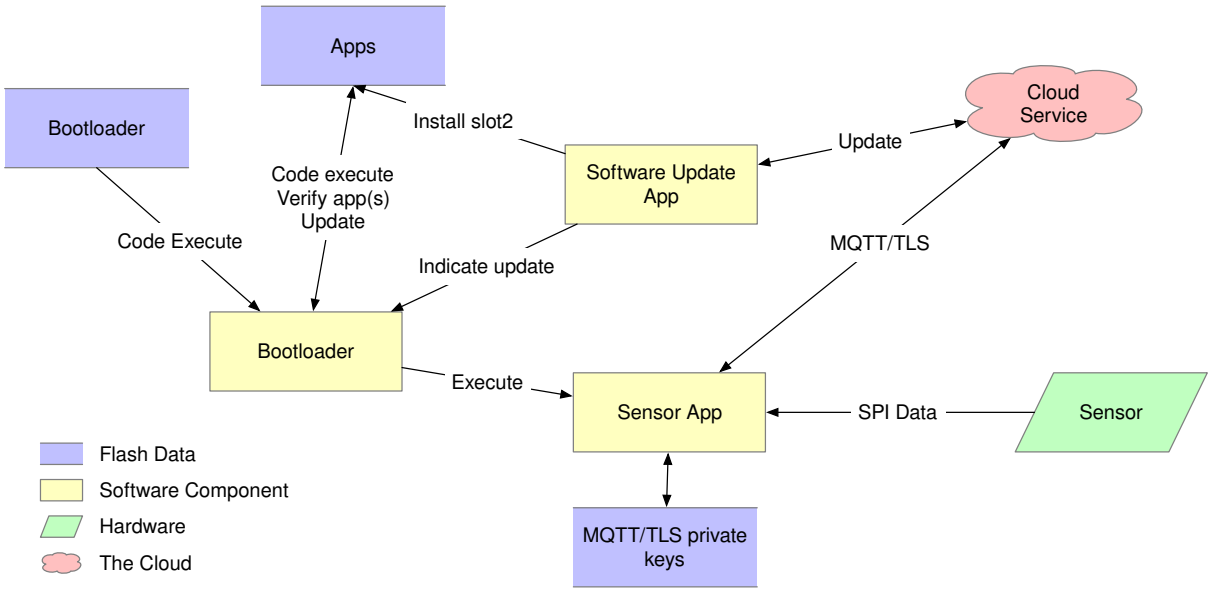


Fig. 3: Figure 1. Sensor General Diagram

In this sensor device, the sensor connects with the SoC via an SPI bus, and the SoC has a network interface that it uses to communicate with the cloud service. The particulars of these interfaces can impact the threat model in unexpected ways, and variants on this will need to be considered (for example, using a separate network interface SoC connected via some type of bus).

This model also focuses on communicating via the MQTT-over-TLS protocol, as this seems to be in wide use[1].

10.4.1 Assets

One aspect of the threat model to consider are assets involved in the operation of the device. The following list enumerates the assets included in this model:

1. **The bootloader**. This is a small code/data image contained in on-device flash that is the first code to run. In order to establish a root of trust, this image must be immutable. This

[1] See https://www.slideshare.net/kartben/iot-developer-survey-2018. As of this writing, the three major cloud IoT service providers, AWS IoT, Google Cloud IoT, and Microsoft Azure IoT all provide MQTT over TLS. Some feedback has suggested that some find difficulty with UDP protocols and routing issues on various networks.

model assumes that the SoC provides a mechanism to protect a region of the flash from future writes, and that this will be done after this image is programmed into the device, early in production [th-imboot].

2. **The application firmware image**. This asset consists of the remainder of the firmware run by the microcontroller. The distinction is made because this part of the image will need to be updated periodically as security vulnerabilities are discovered. Requirements for updates to this image are:

   a. The image shall only be replaced with an authorized image [th-authrepl].

   b. When an authorized replacement image is available, the update shall be done in a timely manner [th-timely-update].

   c. The image update shall be seen as atomic, meaning that when the image is run, the flash shall contain either the update image in its entirety, or the old image in its entirety [th-atomic-update].

3. **Root certificate list**. In order to authenticate the cloud service (server), the IoT device must have a list of root certificates that are allowed to sign the certificate on the server. For cloud-provider based services, this list will generally be provided by the service provider. Because the root certificates can expire, and possibly be revoked, this list will need to be periodically updated [th-root-certs], [th-root-check].

4. **Client secrets**. To authenticate the client to the service, the client must possess some kind of secret. This is generally a private key, usually either an RSA key or an EC private key. When establishing communication with the server, the device will use this secret either as part of the TLS establishment, or to sign a message used in the communication.

   This secret is generally generated by the service provider, or by software running elsewhere, and must be securely installed on the device. Policy may dictate that this secret be replaced periodically, which will require a way to update the client secret. Typically, the service will allow two or three active keys to allow this update to proceed while the old key is used.

   These secrets must be protected from read, and the smallest amount of code necessary shall have access to them. [th-secret-storage]

5. **Current date/time**. TLS certificate verification requires knowledge of the current date and time in order to determine if the current time falls within the certificate's current validity time. Also, token based client authentication will generally require the client to sign a message containing a time window that the token is valid. Certificate validation requires the device's notion of date and time to be accurate within a day or so. Token generation generally requires the time to be accurate within 5-10 minutes.

   It may be possible to approximate secure time by querying an external time server. Secure NTP is possibly beyond the capabilities of an IoT device. The main risks of having incorrect time are denial of service (the device rejects valid certificates), and the generation of tokens with invalid times. It could be possible to trick the device into generating tokens that are valid in the future, but the attacker would also have to spoof the server's certificate to be able to intercept this. [th-time]

6. **Sensor data**. The data received from the sensor itself, and delivered to the service shall be delivered without modification or tampering.

7. **Device configuration**. Various configuration data, such as the hostname of the service to connect to, the address of a time server, frequency and parameters of when sensor data is sent to the service, and other need to be kept by the device. This configuration data will need to be updated periodically as the configuration changes. Updates should be allowed only from authorized parties. [th-conf]

8. **Logs**. In order to assist with analysis of security issues, the device shall log information about security-pertinent events. IoT devices generally have limited storage, and as such, these logs need to be carefully selected. It may also be possible to send these log events

to the cloud service where they can be stored in a more resource-available environment. Types of events that should be logged include:

a. **Firmware image updates**. The system should log the download of new images, and when an image is successfully updated.

b. **Client secret changes**. Changes and new client secrets should be logged.

c. **Changes to the device configuration**.

[th-logs]

10.4.2 Communication

In addition to assets, the threat model also considers the locations where data or assets are communicated between entities of the system.

1. **Flash contents**. The flash device contains several regions. The contents of flash can be modified programmatically by the SoC's CPU.

   a. **The bootloader**. As described in the Assets section, the bootloader is a small section of the flash device containing the code initially run. This section shall be written early in the lifecycle of the device, and the flash device then configured to permanently disallow modification of this section. This configuration should also prevent modification via external interfaces, such as JTAG or SWD debuggers.

      The bootloader is responsible for verifying the signature of the application image as well as updating the application image from the update image when an update is needed.

      The bootloader shall verify the signature of the update image before installing it.

      The bootloader shall only accept an update image with a newer version number than the current image.

   b. **The application image**. The application image contains the code executed during normal operation of the device. Before running this image, the bootloader shall verify a digital signature of the image, to avoid running an image that has been tampered with. The flash/system shall be configured such that after the bootloader has completed, the CPU will be unable to write to the application image.

   c. **The update image**. This is an area of flash that holds a new version of the application image. This image will be downloaded and stored by the application during normal operation. When this has completed, the application can trigger a reboot, and the bootloader can install the new image.

   d. **Secret storage**. An area of the flash will be used to store client secrets. This area is written and read by a subset of the application image. The application shall be configured to protect this area from both reads and writes by code that does not need to have access to it, giving consideration to possible exploits found within a majority of the application code. Revealing the contents of the secrets would allow the attacker to spoof this device.

      Initial secrets shall be placed in the device during a provisioning activity, distinct from normal operation of the device. Later updates can be made under the direction of communication received over a secured channel to the service.

   e. **Configuration storage**. There shall be an area to store other configuration information. On resource-constrained devices, it is allowed for this to be stored in the same region as the secret storage, however, this adds additional code that has access to the secret storage area, and as such, more code that must be scrutinized.

   f. **Log storage**. The device may have an area of flash where log events can be written.

2. **Sensor/Actuator interface**. In this design, the sensor or actuator communicates with the SoC via a bus, such as SPI. The hardware design shall be made to make intercepting this bus difficult for an attack. Required techniques depend on the sensitivity and use of the sensor data, and can range from having the sensor mounted on the same PCB as the MCU to epoxy potting the entire device.

3. **Communication with cloud service**. Communication between the device, and the cloud service will be done over the general internet. As such, it shall be assumed that an attacker can arbitrarily intercept this channel and, for example, return spoofed DNS results or attempt man-in-the-middle attacks against communication with cloud services.

   The device shall use TLS for all communication with the cloud service [th-all-tls]. The TLS stack shall be configured to use only cipher suites that are generally considered secure[2], including forward secrecy. The communication shall be secured by the following:

   a. **Cipher suite selection**. The device shall only allow communication with generally agreed secure cipher suites [th-tls-ciphers].

   b. **Server certificate verification**. The server presented by the server shall be verified [th-root-check].

      i. **Naming**. The certificate shall name the host and service the cloud service server is providing. RFC6125 describes best practices for this. It is permissible for the device to require the certificate to be more restrictive than as described in this RFC, provided the service can use a certificate that can comply.

      ii. **Path validation**. The device shall verify that the certificate chain has a valid signature path from a root certificate contained within the device, to the certificate presented by the service. RFC4158 describes this is general. The device is permitted to require a more restricted path, provided the server certificate used complies with this restriction.

      iii. **Validity period**. The validity period of all presented certificates shall be checked against the device's best notion of the current time.

   c. **Client authentication**. The client shall authenticate itself to the service using a secret known only to that particular device. There are several options, and the technique used is generally mandated by the particular service provider being used [th-tls-client-auth].

      i. **TLS client certificates**. The TLS protocol allows the client to present a certificate, and assert its knowledge of the secret described by that certificate. Generally, these certificates will be stored within the service provider. These certificates can be self-signed, or signed by a CA. Since the service provider maintains a list of valid certificates (mapping them to a device identity), having these certificates signed by a CA does not add any additional security, but may be useful in the management of these certificates.

      ii. **Token-based authentication**. It is also possible for the client to authenticate itself using the *password* field of the MQTT CONNECT packet. However, the secret itself must not be transmitted in this packet. Instead, a token-based protocol, such as RFC7519's JSON Web Token (JWT) can be used. These tokens will generally have a small validity period (e.g. 1 hour), to prevent them from being reused if they are intercepted. The token shall not be sent until the device has verified the identity of the server.

   d. **Random/Entropy source**. Cryptographic communication requires the generation of secure pseudorandom numbers. The device shall use a modern, accepted cryptographic random-bit generator to generate these random numbers. It shall use either a Non-Deterministic Random Bit Generator (True RBG) implemented in hardware within the SoC, or a Deterministic Random Bit Generator (Pseudo RBG) seeded by

[2] As new exploits are discovered, what is considered secure can change. Organizations such as https://www.ssllabs.com/ provide information on current ideas of how TLS must be configured to be secure.

an entropy source within the SoC. Please see NIST SP 800-90A for information on approved RBGs and NIST SP 800-90B for information on testing a device's entropy source [th-entropy].

4. **Communication with the time service**. Ideally, the device shall contain hardware that maintains a secure time. However, most SoCs in use do not have support for this, and it will be necessary to consult an external time service. RFC4330 and referenced RFCs describe the Simple Network Time Protocol that can be used to query the current time from a network time server.
5. **Device lifecycle**. An IoT device will have a lifecycle from production to destruction and disposal of the device. Aspects of this lifecycle that impact security include initial provisioning, normal operation, re-provisioning, and destruction.
   a. **Initial provisioning**. During the initial provisioning stage, it is necessary to program the bootloader, an initial application image, a device secret, and initial configuration data [th-initial-provision]. In addition, the bootloader flash protection shall be installed. Of this information, only the device secret needs to differ per device. This secret shall be securely maintained, and destroyed in all locations outside of the device once it has been programmed [th-initial-secret].
   b. **Normal operation**. Normal operation includes the behavior described by the rest of this document.
   c. **Re-provisioning**. Sometimes it is necessary to re-provision a device, such as for a different application. One way to do this is to keep the same device secret, and replace the configuration data, as well as the cloud service data associated with the device. It is also possible to program a new device secret, but if this is done it shall be done securely, and the new secret destroyed externally once programmed into the device [th-reprovision].
   d. **Destruction**. To prevent the device secret from being used to spoof the device, upon decommissioning, the secret for a particular device shall be rendered ineffective [th-destruction]. Possibilities include:
      i. Hardware destruction of the device.
      ii. Securely wiping the flash area containing the secret[3].
      iii. Removing the device identity and certificate from the service.

10.4.3 Other Considerations

In addition to the above, network connected devices generally will need a way to configure them to connect to the network environment they are placed in. There are numerous ways of doing this, and it is important for these configuration methods to not circumvent the security requirements described above.

10.4.4 Threats

10.4.5 Notes

10.5 Hardening Tool

Before launching a product, it's crucial to ensure that your software is as secure as possible. This process, known as "hardening", involves strengthening the security of a system to protect it from potential threats and vulnerabilities.

[3] Note that merely erasing this flash area is unlikely to be sufficient.

At a high-level, hardening a Zephyr application can be seen as a two-fold process:

1. Disabling features and compilation flags that might lead to security vulnerabilities (ex. making sure that no "experimental" features are being used, disabling features typically used for debugging purposes such as assertions, shell, etc.).
2. Enabling optional features that can lead to improve security (ex. stack sentinel, hardware stack protection, etc.). Some of these features might be hardware-dependent.

To simplify this process, Zephyr offers a **hardening tool** designed to analyze an application's configuration against a set of hardening preferences defined by the **Security Working Group**. The tool looks at the KConfig options in the build target and provides tailored suggestions and recommendations to adjust security-related options.

10.5.1 Usage

Using west:

```
west build -b reel_board samples/hello_world
west build -t hardenconfig
```

Using CMake and ninja:

```
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=reel_board samples/hello_world

# Now run the build tool on the generated build system:
ninja -Cbuild hardenconfig
```

The output should be similar to the table below. For each configuration option set to a value that could lead to a security vulnerability, the table will propose a recommended value that should be used instead.

```
               name                          |   current   |  recommended   ||    check result
=============================================================================================================
CONFIG_BOOT_BANNER                           |      y      |       n        ||       FAIL
CONFIG_BUILD_OUTPUT_STRIPPED                 |      n      |       y        ||       FAIL
CONFIG_FAULT_DUMP                            |      2      |       0        ||       FAIL
CONFIG_HW_STACK_PROTECTION                   |      n      |       y        ||       FAIL
CONFIG_MPU_STACK_GUARD                       |      n      |       y        ||       FAIL
CONFIG_OVERRIDE_FRAME_POINTER_DEFAULT        |      n      |       y        ||       FAIL
CONFIG_STACK_SENTINEL                        |      n      |       y        ||       FAIL
CONFIG_EARLY_CONSOLE                         |      y      |       n        ||       FAIL
CONFIG_PRINTK                                |      y      |       n        ||       FAIL
```

10.6 Vulnerabilities

This page collects all of the vulnerabilities that are discovered and fixed in each release. It will also often have more details than is available in the releases. Some vulnerabilities are deemed to be sensitive, and will not be publicly discussed until there is sufficient time to fix them. Because the release notes are locked to a version, the information here can be updated after the embargo is lifted.

10.6.1 CVE-2017

CVE-2017-14199

Buffer overflow in `getaddrinfo()`.

- CVE-2017-14199
- Zephyr project bug tracker ZEPSEC-12
- PR6158 fix for 1.11.0

CVE-2017-14201

The shell DNS command can cause unpredictable results due to misuse of stack variables.

Use After Free vulnerability in the Zephyr shell allows a serial or telnet connected user to cause denial of service, and possibly remote code execution.

This has been fixed in release v1.14.0.

- CVE-2017-14201
- Zephyr project bug tracker ZEPSEC-17
- PR13260 fix for v1.14.0

CVE-2017-14202

The shell implementation does not protect against buffer overruns resulting in unpredictable behavior.

Improper Restriction of Operations within the Bounds of a Memory Buffer vulnerability in the shell component of Zephyr allows a serial or telnet connected user to cause a crash, possibly with arbitrary code execution.

This has been fixed in release v1.14.0.

- CVE-2017-14202
- Zephyr project bug tracker ZEPSEC-18
- PR13048 fix for v1.14.0

10.6.2 CVE-2019

CVE-2019-9506

The Bluetooth BR/EDR specification up to and including version 5.1 permits sufficiently low encryption key length and does not prevent an attacker from influencing the key length negotiation. This allows practical brute-force attacks (aka “KNOB”) that can decrypt traffic and inject arbitrary ciphertext without the victim noticing.

- CVE-2019-9506
- Zephyr project bug tracker ZEPSEC-20
- PR18702 fix for v1.14.0
- PR18659 fix for v2.0.0

10.6.3 CVE-2020

CVE-2020-10019

Buffer Overflow vulnerability in USB DFU of zephyr allows a USB connected host to cause possible remote code execution.

This has been fixed in releases v1.14.2, v2.2.0, and v2.1.1.

- CVE-2020-10019
- Zephyr project bug tracker ZEPSEC-25
- PR23460 fix for 1.14.x
- PR23457 fix for 2.1.x
- PR23190 fix in 2.2.0

CVE-2020-10021

Out-of-bounds write in USB Mass Storage with unaligned sizes

Out-of-bounds Write in the USB Mass Storage memoryWrite handler with unaligned Sizes.

See NCC-ZEP-024, NCC-ZEP-025, NCC-ZEP-026

This has been fixed in releases v1.14.2, and v2.2.0.

- CVE-2020-10021
- Zephyr project bug tracker ZEPSEC-26
- PR23455 fix for v1.14.2
- PR23456 fix for the v2.1 branch
- PR23240 fix for v2.2.0

CVE-2020-10022

UpdateHub Module Copies a Variable-Size Hash String Into a Fixed-Size Array

A malformed JSON payload that is received from an UpdateHub server may trigger memory corruption in the Zephyr OS. This could result in a denial of service in the best case, or code execution in the worst case.

See NCC-ZEP-016

This has been fixed in the below pull requests for main, branch from v2.1.0, and branch from v2.2.0.

- CVE-2020-10022
- Zephyr project bug tracker ZEPSEC-28
- PR24154 fix for main
- PR24065 fix for branch from v2.1.0
- PR24066 fix for branch from v2.2.0

CVE-2020-10023

Shell Subsystem Contains a Buffer Overflow Vulnerability In shell_spaces_trim

The shell subsystem contains a buffer overflow, whereby an adversary with physical access to the device is able to cause a memory corruption, resulting in denial of service or possibly code execution within the Zephyr kernel.

See NCC-ZEP-019

This has been fixed in releases v1.14.2, v2.2.0, and in a branch from v2.1.0,

- CVE-2020-10023
- Zephyr project bug tracker ZEPSEC-29
- PR23646 fix for v1.14.2
- PR23649 fix for branch from v2.1.0
- PR23304 fix for v2.2.0

CVE-2020-10024

ARM Platform Uses Signed Integer Comparison When Validating Syscall Numbers

The arm platform-specific code uses a signed integer comparison when validating system call numbers. An attacker who has obtained code execution within a user thread is able to elevate privileges to that of the kernel.

See NCC-ZEP-001

This has been fixed in releases v1.14.2, and v2.2.0, and in a branch from v2.1.0,

- CVE-2020-10024
- Zephyr project bug tracker ZEPSEC-30
- PR23535 fix for v1.14.2
- PR23498 fix for branch from v2.1.0
- PR23323 fix for v2.2.0

CVE-2020-10027

ARC Platform Uses Signed Integer Comparison When Validating Syscall Numbers

An attacker who has obtained code execution within a user thread is able to elevate privileges to that of the kernel.

See NCC-ZEP-001

This has been fixed in releases v1.14.2, and v2.2.0, and in a branch from v2.1.0.

- CVE-2020-10027
- Zephyr project bug tracker ZEPSEC-35
- PR23500 fix for v1.14.2
- PR23499 fix for branch from v2.1.0
- PR23328 fix for v2.2.0

CVE-2020-10028

Multiple Syscalls In GPIO Subsystem Performs No Argument Validation

Multiple syscalls with insufficient argument validation

See NCC-ZEP-006

This has been fixed in releases v1.14.2, and v2.2.0, and in a branch from v2.1.0.

- CVE-2020-10028
- Zephyr project bug tracker ZEPSEC-32
- PR23733 fix for v1.14.2
- PR23737 fix for branch from v2.1.0
- PR23308 fix for v2.2.0 (gpio patch)

CVE-2020-10058

Multiple Syscalls In kscan Subsystem Performs No Argument Validation

Multiple syscalls in the Kscan subsystem perform insufficient argument validation, allowing code executing in userspace to potentially gain elevated privileges.

See NCC-ZEP-006

This has been fixed in a branch from v2.1.0, and release v2.2.0.

- CVE-2020-10058
- Zephyr project bug tracker ZEPSEC-34
- PR23748 fix for branch from v2.1.0
- PR23308 fix for v2.2.0 (kscan patch)

CVE-2020-10059

UpdateHub Module Explicitly Disables TLS Verification

The UpdateHub module disables DTLS peer checking, which allows for a man in the middle attack. This is mitigated by firmware images requiring valid signatures. However, there is no benefit to using DTLS without the peer checking.

See NCC-ZEP-018

This has been fixed in a PR against Zephyr main.

- CVE-2020-10059
- Zephyr project bug tracker ZEPSEC-36
- PR24954 fix on main (to be fixed in v2.3.0)
- PR24954 fix v2.1.0
- PR24954 fix v2.2.0

CVE-2020-10060

UpdateHub Might Dereference An Uninitialized Pointer

In updatehub_probe, right after JSON parsing is complete, objects[1] is accessed from the output structure in two different places. If the JSON contained less than two elements, this access would reference uninitialized stack memory. This could result in a crash, denial of service, or possibly an information leak.

Recommend disabling updatehub until such a time as a fix can be made available.

See NCC-ZEP-030

This has been fixed in a PR against Zephyr main.

- CVE-2020-10060
- Zephyr project bug tracker ZEPSEC-37
- PR27865 fix on main (to be fixed in v2.4.0)
- PR27865 fix for v2.3.0
- PR27865 fix for v2.2.0
- PR27865 fix for v2.1.0

CVE-2020-10061

Error handling invalid packet sequence

Improper handling of the full-buffer case in the Zephyr Bluetooth implementation can result in memory corruption.

This has been fixed in branches for v1.14.0, v2.2.0, and will be included in v2.3.0.

- CVE-2020-10061
- Zephyr project bug tracker ZEPSEC-75
- PR23516 fix for v2.3 (split driver)
- PR23517 fix for v2.3 (legacy driver)
- PR23091 fix for branch from v1.14.0
- PR23547 fix for branch from v2.2.0

CVE-2020-10062

Packet length decoding error in MQTT

CVE: An off-by-one error in the Zephyr project MQTT packet length decoder can result in memory corruption and possible remote code execution. NCC-ZEP-031

The MQTT packet header length can be 1 to 4 bytes. An off-by-one error in the code can result in this being interpreted as 5 bytes, which can cause an integer overflow, resulting in memory corruption.

This has been fixed in main for v2.3.

- CVE-2020-10062
- Zephyr project bug tracker ZEPSEC-84
- commit 11b7a37d for v2.3
- NCC-ZEP report (NCC-ZEP-031)

CVE-2020-10063

Remote Denial of Service in CoAP Option Parsing Due To Integer Overflow

A remote adversary with the ability to send arbitrary CoAP packets to be parsed by Zephyr is able to cause a denial of service.

This has been fixed in main for v2.3.

- CVE-2020-10063
- Zephyr project bug tracker ZEPSEC-55
- PR24435 fix in main for v2.3
- PR24531 fix for branch from v2.2
- PR24535 fix for branch from v2.1
- PR24530 fix for branch from v1.14
- NCC-ZEP report (NCC-ZEP-032)

CVE-2020-10064

Improper Input Frame Validation in ieee802154 Processing

- CVE-2020-10064
- Zephyr project bug tracker ZEPSEC-65
- PR24971 fix for v2.4
- PR33451 fix for v1.4

CVE-2020-10065

OOB Write after not validating user-supplied length (<= 0xffff) and copying to fixed-size buffer (default: 77 bytes) for HCI_ACL packets in bluetooth HCI over SPI driver.

- CVE-2020-10065
- Zephyr project bug tracker ZEPSEC-66
- This issue has not been fixed.

CVE-2020-10066

Incorrect Error Handling in Bluetooth HCI core

In hci_cmd_done, the buf argument being passed as null causes nullpointer dereference.

- CVE-2020-10066
- Zephyr project bug tracker ZEPSEC-67
- PR24902 fix for v2.4
- PR25089 fix for v1.4

CVE-2020-10067

Integer Overflow In is_in_region Allows User Thread To Access Kernel Memory

A malicious userspace application can cause a integer overflow and bypass security checks performed by system call handlers. The impact would depend on the underlying system call and can range from denial of service to information leak to memory corruption resulting in code execution within the kernel.

See NCC-ZEP-005

This has been fixed in releases v1.14.2, and v2.2.0.

- CVE-2020-10067
- Zephyr project bug tracker ZEPSEC-27
- PR23653 fix for v1.14.2
- PR23654 fix for the v2.1 branch
- PR23239 fix for v2.2.0

CVE-2020-10068

Zephyr Bluetooth DLE duplicate requests vulnerability

In the Zephyr project Bluetooth subsystem, certain duplicate and back-to-back packets can cause incorrect behavior, resulting in a denial of service.

This has been fixed in branches for v1.14.0, v2.2.0, and will be included in v2.3.0.

- CVE-2020-10068
- Zephyr project bug tracker ZEPSEC-78
- PR23707 fix for v2.3 (split driver)
- PR23708 fix for v2.3 (legacy driver)
- PR23091 fix for branch from v1.14.0
- PR23964 fix for v2.2.0

CVE-2020-10069

Zephyr Bluetooth unchecked packet data results in denial of service

An unchecked parameter in bluetooth data can result in an assertion failure, or division by zero, resulting in a denial of service attack.

This has been fixed in branches for v1.14.0, v2.2.0, and will be included in v2.3.0.

- CVE-2020-10069
- Zephyr project bug tracker ZEPSEC-81
- PR23705 fix for v2.3 (split driver)
- PR23706 fix for v2.3 (legacy driver)
- PR23091 fix for branch from v1.14.0
- PR23963 fix for branch from v2.2.0

CVE-2020-10070

MQTT buffer overflow on receive buffer

In the Zephyr Project MQTT code, improper bounds checking can result in memory corruption and possibly remote code execution. NCC-ZEP-031

When calculating the packet length, arithmetic overflow can result in accepting a receive buffer larger than the available buffer space, resulting in user data being written beyond this buffer.

This has been fixed in main for v2.3.

- CVE-2020-10070
- Zephyr project bug tracker ZEPSEC-85
- commit 0b39cbf3 for v2.3
- NCC-ZEP report (NCC-ZEP-031)

CVE-2020-10071

Insufficient publish message length validation in MQTT

The Zephyr MQTT parsing code performs insufficient checking of the length field on publish messages, allowing a buffer overflow and potentially remote code execution. NCC-ZEP-031

This has been fixed in main for v2.3.

- CVE-2020-10071
- Zephyr project bug tracker ZEPSEC-86
- commit 989c4713 fix for v2.3
- NCC-ZEP report (NCC-ZEP-031)

CVE-2020-10072

All threads can access all socket file descriptors

There is no management of permissions to network socket API file descriptors. Any thread running on the system may read/write a socket file descriptor knowing only the numerical value of the file descriptor.

- CVE-2020-10072
- Zephyr project bug tracker ZEPSEC-87
- PR25804 fix for v2.4
- PR27176 fix for v1.4

CVE-2020-10136

IP-in-IP protocol routes arbitrary traffic by default zephyrproject

- CVE-2020-10136
- Zephyr project bug tracker ZEPSEC-64

CVE-2020-13598

FS: Buffer Overflow when enabling Long File Names in FAT_FS and calling fs_stat

Performing fs_stat on a file with a filename longer than 12 characters long will cause a buffer overflow.

- CVE-2020-13598
- Zephyr project bug tracker ZEPSEC-88
- PR25852 fix for v2.4
- PR28782 fix for v2.3
- PR33577 fix for v1.4

CVE-2020-13599

Security problem with settings and littlefs

When settings is used in combination with littlefs all security related information can be extracted from the device using MCUmgr and this could be used e.g in bt-mesh to get the device key, network key, app keys from the device.

- CVE-2020-13599
- Zephyr project bug tracker ZEPSEC-57
- PR26083 fix for v2.4

CVE-2020-13600

Malformed SPI in response for eswifi can corrupt kernel memory

- CVE-2020-13600
- Zephyr project bug tracker ZEPSEC-91
- PR26712 fix for v2.4

CVE-2020-13601

Possible read out of bounds in dns read

- CVE-2020-13601
- Zephyr project bug tracker ZEPSEC-92
- PR27774 fix for v2.4
- PR30503 fix for v1.4

CVE-2020-13602

Remote Denial of Service in LwM2M do_write_op_tlv

In the Zephyr LwM2M implementation, malformed input can result in an infinite loop, resulting in a denial of service attack.

- CVE-2020-13602
- Zephyr project bug tracker ZEPSEC-56

- PR26571 fix for v2.4
- PR33578 fix for v1.4

CVE-2020-13603

Possible overflow in mempool

- Zephyr offers pre-built 'malloc' wrapper function instead.
- The 'malloc' function is wrapper for the 'sys_mem_pool_alloc' function
- sys_mem_pool_alloc allocates 'size + WB_UP(sizeof(struct sys_mem_pool_block))' in an unsafe manner.
- Asking for very large size values leads to internal integer wrap-around.
- Integer wrap-around leads to successful allocation of very small memory.
- For example: calling malloc(0xffffffff) leads to successful allocation of 7 bytes.
- That leads to heap overflow.
- CVE-2020-13603
- Zephyr project bug tracker ZEPSEC-111
- PR31796 fix for v2.4
- PR32808 fix for v1.4

10.6.4 CVE-2021

CVE-2021-3319

DOS: Incorrect 802154 Frame Validation for Omitted Source / Dest Addresses

Improper processing of omitted source and destination addresses in ieee802154 frame validation (ieee802154_validate_frame)

This has been fixed in main for v2.5.0

- CVE-2020-3319
- Zephyr project bug tracker GHSA-94jg-2p6q-5364
- PR31908 fix for main

CVE-2021-3320

Mismatch between validation and handling of 802154 ACK frames, where ACK frames are considered during validation, but not during actual processing, leading to a type confusion.

- CVE-2020-3320
- PR31908 fix for main

CVE-2021-3321

Incomplete check of minimum IEEE 802154 fragment size leading to an integer underflow.

- CVE-2020-3321
- Zephyr project bug tracker ZEPSEC-114

- PR33453 fix for v2.4

CVE-2021-3323

Integer Underflow in 6LoWPAN IPHC Header Uncompression
This has been fixed in main for v2.5.0

- CVE-2020-3323
- Zephyr project bug tracker GHSA-89j6-qpxf-pfpc
- PR 31971 fix for main

CVE-2021-3430

Assertion reachable with repeated LL_CONNECTION_PARAM_REQ.
This has been fixed in main for v2.6.0

- CVE-2021-3430
- Zephyr project bug tracker GHSA-46h3-hjcq-2jjr
- PR 33272 fix for main
- PR 33369 fix for 2.5
- PR 33759 fix for 1.14.2

CVE-2021-3431

BT: Assertion failure on repeated LL_FEATURE_REQ
This has been fixed in main for v2.6.0

- CVE-2021-3431
- Zephyr project bug tracker GHSA-7548-5m6f-mqv9
- PR 33340 fix for main
- PR 33369 fix for 2.5

CVE-2021-3432

Invalid interval in CONNECT_IND leads to Division by Zero
This has been fixed in main for v2.6.0

- CVE-2021-3432
- Zephyr project bug tracker GHSA-7364-p4wc-8mj4
- PR 33278 fix for main
- PR 33369 fix for 2.5

CVE-2021-3433

BT: Invalid channel map in CONNECT_IND results to Deadlock

This has been fixed in main for v2.6.0

- CVE-2021-3433
- Zephyr project bug tracker GHSA-3c2f-w4v6-qxrp
- PR 33278 fix for main
- PR 33369 fix for 2.5

CVE-2021-3434

L2CAP: Stack based buffer overflow in le_ecred_conn_req()

This has been fixed in main for v2.6.0

- CVE-2021-3434
- Zephyr project bug tracker GHSA-8w87-6rfp-cfrm
- PR 33305 fix for main
- PR 33419 fix for 2.5
- PR 33418 fix for 1.14.2

CVE-2021-3435

L2CAP: Information leakage in le_ecred_conn_req()

This has been fixed in main for v2.6.0

- CVE-2021-3435
- Zephyr project bug tracker GHSA-xhg3-gvj6-4rqh
- PR 33305 fix for main
- PR 33419 fix for 2.5
- PR 33418 fix for 1.14.2

CVE-2021-3436

Bluetooth: Possible to overwrite an existing bond during keys distribution phase when the identity address of the bond is known

During the distribution of the identity address information we don't check for an existing bond with the same identity address.This means that a duplicate entry will be created in RAM while the newest entry will overwrite the existing one in persistent storage.

This has been fixed in main for v2.6.0

- CVE-2021-3436
- Zephyr project bug tracker GHSA-j76f-35mc-4h63
- PR 33266 fix for main
- PR 33432 fix for 2.5
- PR 33433 fix for 2.4

- PR 33718 fix for 1.14.2

CVE-2021-3454

Truncated L2CAP K-frame causes assertion failure

For example, sending L2CAP K-frame where SDU length field is truncated to only one byte, causes assertion failure in previous releases of Zephyr. This has been fixed in master by commit 0ba9437 but has not yet been backported to older release branches.

This has been fixed in main for v2.6.0

- CVE-2021-3454
- Zephyr project bug tracker GHSA-fx88-6c29-vrp3
- PR 32588 fix for main
- PR 33513 fix for 2.5
- PR 33514 fix for 2.4

CVE-2021-3455

Disconnecting L2CAP channel right after invalid ATT request leads freeze

When Central device connects to peripheral and creates L2CAP connection for Enhanced ATT, sending some invalid ATT request and disconnecting immediately causes freeze.

This has been fixed in main for v2.6.0

- CVE-2021-3455
- Zephyr project bug tracker GHSA-7g38-3x9v-v7vp
- PR 35597 fix for main
- PR 36104 fix for 2.5
- PR 36105 fix for 2.4

CVE-2021-3510

Zephyr JSON decoder incorrectly decodes array of array

When using JSON_OBJ_DESCR_ARRAY_ARRAY, the subarray is has the token type JSON_TOK_LIST_START, but then assigns to the object part of the union. arr_parse then takes the offset of the array-object (which has nothing todo with the list) treats it as relative to the parent object, and stores the length of the subarray in there.

This has been fixed in main for v2.7.0

- CVE-2021-3510
- Zephyr project bug tracker GHSA-289f-7mw3-2qf4
- PR 36340 fix for main
- PR 37816 fix for 2.6

CVE-2021-3581

HCI data not properly checked leads to memory overflow in the Bluetooth stack

In the process of setting SCAN_RSP through the HCI command, the Zephyr Bluetooth protocol stack did not effectively check the length of the incoming HCI data. Causes memory overflow, and then the data in the memory is overwritten, and may even cause arbitrary code execution.

This has been fixed in main for v2.6.0

- CVE-2021-3581
- Zephyr project bug tracker GHSA-8q65-5gqf-fmw5
- PR 35935 fix for main
- PR 35984 fix for 2.5
- PR 35985 fix for 2.4
- PR 35985 fix for 1.14

CVE-2021-3625

Buffer overflow in Zephyr USB DFU DNLOAD

This has been fixed in main for v2.6.0

- CVE-2021-3625
- Zephyr project bug tracker GHSA-c3gr-hgvr-f363
- PR 36694 fix for main

CVE-2021-3835

Buffer overflow in Zephyr USB device class

This has been fixed in main for v3.0.0

- CVE-2021-3835
- Zephyr project bug tracker GHSA-fm6v-8625-99jf
- PR 42093 fix for main
- PR 42167 fix for 2.7

CVE-2021-3861

Buffer overflow in the RNDIS USB device class

This has been fixed in main for v3.0.0

- CVE-2021-3861
- Zephyr project bug tracker GHSA-hvfp-w4h8-gxvj
- PR 39725 fix for main

CVE-2021-3966

Usb bluetooth device ACL read cb buffer overflow

This has been fixed in main for v3.0.0

- Zephyr project bug tracker GHSA-hfxq-3w6x-fv2m
- PR 42093 fix for main
- PR 42167 fix for v2.7.0

10.6.5 CVE-2022

CVE-2022-0553

Possible to retrieve unencrypted firmware image

This has been fixed in main for v3.0.0

- Zephyr project bug tracker GHSA-wrj2-9vj9-rrcp
- PR 42424 fix for main

CVE-2022-1041

Out-of-bound write vulnerability in the Bluetooth Mesh core stack can be triggered during provisioning

This has been fixed in main for v3.1.0

- Zephyr project bug tracker GHSA-p449-9hv9-pj38
- PR 45136 fix for main
- PR 45188 fix for v3.0.0
- PR 45187 fix for v2.7.0

CVE-2022-1042

Out-of-bound write vulnerability in the Bluetooth Mesh core stack can be triggered during provisioning

This has been fixed in main for v3.1.0

- Zephyr project bug tracker GHSA-j7v7-w73r-mm5x
- PR 45066 fix for main
- PR 45135 fix for v3.0.0
- PR 45134 fix for v2.7.0

CVE-2022-1841

Out-of-Bound Write in tcp_flags

This has been fixed in main for v3.1.0

- Zephyr project bug tracker GHSA-5c3j-p8cr-2pgh
- PR 45796 fix for main

CVE-2022-2741

can: denial-of-service can be triggered by a crafted CAN frame

This has been fixed in main for v3.2.0

- Zephyr project bug tracker GHSA-hx5v-j59q-c3j8
- PR 47903 fix for main
- PR 47957 fix for v3.1.0
- PR 47958 fix for v3.0.0
- PR 47959 fix for v2.7.0

CVE-2022-2993

bt: host: Wrong key validation check

This has been fixed in main for v3.2.0

- Zephyr project bug tracker GHSA-3286-jgjx-8cvr
- PR 48733 fix for main

CVE-2022-3806

DoS: Invalid Initialization in le_read_buffer_size_complete()

- Zephyr project bug tracker GHSA-w525-fm68-ppq3

10.6.6 CVE-2023

CVE-2023-0396

Buffer Overreads in Bluetooth HCI

- Zephyr project bug tracker GHSA-8rpp-6vxq-pqg3

CVE-2023-0397

DoS: Invalid Initialization in le_read_buffer_size_complete()

- Zephyr project bug tracker GHSA-wc2h-h868-q7hj

This has been fixed in main for v3.3.0

- PR 54905 fix for main
- PR 47957 fix for v3.2.0
- PR 47958 fix for v3.1.0
- PR 47959 fix for v2.7.4

CVE-2023-0779

net: shell: Improper input validation

- Zephyr project bug tracker GHSA-9xj8-6989-r549

This has been fixed in main for v3.3.0

- PR 54371 fix for main
- PR 54380 fix for v3.2.0
- PR 54381 fix for v2.7.4

CVE-2023-1901

HCI send_sync Dangling Semaphore Reference Re-use

- Zephyr project bug tracker GHSA-xvvm-8mcm-9cq3

This has been fixed in main for v3.4.0

- PR 56709 fix for main

CVE-2023-1902

HCI Connection Creation Dangling State Reference Re-use

- Zephyr project bug tracker GHSA-fx9g-8fr2-q899

This has been fixed in main for v3.4.0

- PR 56709 fix for main

CVE-2023-3725

Potential buffer overflow vulnerability in the Zephyr CANbus subsystem.

- Zephyr project bug tracker GHSA-2g3m-p6c7-8rr3

This has been fixed in main for v3.5.0

- PR 61502 fix for main
- PR 61518 fix for 3.4
- PR 61517 fix for 3.3
- PR 61516 fix for 2.7

CVE-2023-4257

Unchecked user input length in the Zephyr WiFi shell module can cause buffer overflows.

- Zephyr project bug tracker GHSA-853q-q69w-gf5j

This has been fixed in main for v3.5.0

- PR 605377 fix for main
- PR 61383 fix for 3.4

CVE-2023-4258

bt: mesh: vulnerability in provisioning protocol implementation on provisionee side

- Zephyr project bug tracker GHSA-m34c-cp63-rwh7

This has been fixed in main for v3.5.0

- PR 59467 fix for main
- PR 60078 fix for 3.4
- PR 60079 fix for 3.3

CVE-2023-4259

Buffer overflow vulnerabilities in the Zephyr eS-WiFi driver

- Zephyr project bug tracker GHSA-gghm-c696-f4j4

This has been fixed in main for v3.5.0

- PR 63074 fix for main
- PR 63750 fix for main

CVE-2023-4260

Off-by-one buffer overflow vulnerability in the Zephyr FS subsystem

- Zephyr project bug tracker GHSA-gj27-862r-55wh

This has been fixed in main for v3.5.0

- PR 63079 fix for main

CVE-2023-4262

Potential buffer overflow vulnerabilities in the Zephyr Mgmt subsystem

- Zephyr project bug tracker GHSA-56p9-5p3v-hhrc
- This issue has not been fixed.

CVE-2023-4263

Potential buffer overflow vulnerability in the Zephyr IEEE 802.15.4 nRF 15.4 driver.

- Zephyr project bug tracker GHSA-rf6q-rhhp-pqhf

This has been fixed in main for v3.5.0

- PR 60528 fix for main
- PR 61384 fix for 3.4
- PR 61216 fix for 2.7

CVE-2023-4264

Potential buffer overflow vulnerabilities in the Zephyr Bluetooth subsystem

- Zephyr project bug tracker GHSA-rgx6-3w4j-gf5j

This has been fixed in main for v3.5.0

- PR 58834 fix for main
- PR 60465 fix for main
- PR 61845 fix for main
- PR 61385 fix for 3.4

CVE-2023-4265

Two potential buffer overflow vulnerabilities in Zephyr USB code

- Zephyr project bug tracker GHSA-4vgv-5r6q-r6xh

This has been fixed in main for v3.4.0

- PR 59157 fix for main
- PR 59018 fix for main

CVE-2023-4424

bt: hci: DoS and possible RCE

- Zephyr project bug tracker GHSA-j4qm-xgpf-qjw3

This has been fixed in main for v3.5.0

- PR 61651 fix for main
- PR 61696 fix for 3.4
- PR 61695 fix for 3.3
- PR 61694 fix for 2.7

CVE-2023-5055

L2CAP: Possible Stack based buffer overflow in le_ecred_reconf_req()

- Zephyr project bug tracker GHSA-wr8r-7f8x-24jj

This has been fixed in main for v3.5.0

- PR 62381 fix for main

CVE-2023-5139

Potential buffer overflow vulnerability in the Zephyr STM32 Crypto driver.

- Zephyr project bug tracker GHSA-rhrc-pcxp-4453

This has been fixed in main for v3.5.0

- PR 61839 fix for main

CVE-2023-5184

Potential signed to unsigned conversion errors and buffer overflow vulnerabilities in the Zephyr IPM driver

- Zephyr project bug tracker GHSA-8x3p-q3r5-xh9g

This has been fixed in main for v3.5.0

- PR 63069 fix for main

CVE-2023-5563

The SJA1000 CAN controller driver backend automatically attempts to recover from a bus-off event when built with CONFIG_CAN_AUTO_BUS_OFF_RECOVERY=y. This results in calling k_sleep() in IRQ context, causing a fatal exception.

- Zephyr project bug tracker GHSA-98mc-rj7w-7rpv

This has been fixed in main for v3.5.0

- PR 63713 fix for main
- PR 63718 fix for 3.4
- PR 63717 fix for 3.3

CVE-2023-5753

Potential buffer overflow vulnerabilities in the Zephyr Bluetooth subsystem source code when asserts are disabled.

- Zephyr project bug tracker GHSA-hmpr-px56-rvww

This has been fixed in main for v3.5.0

- PR 63605 fix for main

CVE-2023-5779

Out of bounds issue in remove_rx_filter in multiple can drivers.

- Zephyr project bug tracker GHSA-7cmj-963q-jj47

This has been fixed in main for v3.6.0

- PR 64399 fix for main
- PR 64416 fix for 3.5
- PR 64415 fix for 3.4
- PR 64427 fix for 3.3
- PR 64431 fix for 2.7

CVE-2023-6249

Signed to unsigned conversion problem in esp32_ipm_send may lead to buffer overflow

- Zephyr project bug tracker GHSA-32f5-3p9h-2rqc

This has been fixed in main for v3.6.0

- PR 65546 fix for main

CVE-2023-6749

Potential buffer overflow due unchecked data coming from user input in settings shell.

- Zephyr project bug tracker GHSA-757h-rw37-66hw

This has been fixed in main for v3.6.0

- PR 66451 fix for main
- PR 66584 fix for 3.5

CVE-2023-6881

Potential buffer overflow vulnerability in Zephyr fuse file system.

- Zephyr project bug tracker GHSA-mh67-4h3q-p437

This has been fixed in main for v3.6.0

- PR 66592 fix for main

CVE-2023-7060

Missing Security Control in Zephyr OS IP Packet Handling

- Zephyr project bug tracker GHSA-fjc8-223c-qgqr

This has been fixed in main for v3.6.0

- PR 66645 fix for main
- PR 66739 fix for 3.5
- PR 66738 fix for 3.4
- PR 66887 fix for 2.7

10.6.7 CVE-2024

CVE-2024-1638

Bluetooth characteristic LESC security requirement not enforced without additional flags

- Zephyr project bug tracker GHSA-p6f3-f63q-5mc2

This has been fixed in main for v3.6.0

- PR 69170 fix for main

CVE-2024-3077

Bluetooth: Integer underflow in gatt_find_info_rsp. A malicious BLE device can crash BLE victim device by sending malformed gatt packet.

- Zephyr project bug tracker GHSA-gmfv-4vfh-2mh8

This has been fixed in main for v3.7.0

- PR 69396 fix for main

CVE-2024-3332

Bluetooth: DoS caused by null pointer dereference.

A malicious BLE device can send a specific order of packet sequence to cause a DoS attack on the victim BLE device.

- Zephyr project bug tracker GHSA-jmr9-xw2v-5vf4

This has been fixed in main for v3.7.0

- PR 71030 fix for main

CVE-2024-4785

Under embargo until 2024-08-07

CVE-2024-5754

Under embargo until 2024-09-04

CVE-2024-5931

Under embargo until 2024-09-10

CVE-2024-6135

Under embargo until 2024-09-11

CVE-2024-6137

Under embargo until 2024-09-11

CVE-2024-6258

Under embargo until 2024-09-05

CVE-2024-6259

Under embargo until 2024-09-12

CVE-2024-6442

Under embargo until 2024-09-22

CVE-2024-6443

Under embargo until 2024-09-22

CVE-2024-6444

Under embargo until 2024-09-22

10.7 Security standards and Zephyr

For a long period organizations were, more or less, left responsible to deal with cyber security on their own. This included how to assess the scale and impact of the problem and who to properly respond it.

Now, governments started looking how to regulate it and several regulations and enforcements are rapidly emerging, and consequently, security standards. These standards provide guidelines and outline requirements that products have to follow to achieve compliance.

This section aims to identify and assess which Zephyr project components are impacted by security standards requirements and provide the right information to enable organizations developing certifiable products using Zephyr project.

10.7.1 ETSI 303-645

ETSI EN 303 645, also known as "Cyber Security for Consumer Internet of Things: Baseline Requirements," is a standard developed by the European Telecommunications Standards Institute (ETSI).

The standard includes provisions for secure software updates, data protection, secure communication, and the minimization of exposed attack surfaces, among other things. It is part of a broader effort to address the challenges and risks associated with IoT devices.

Full version of the standard can be found here.

Terminology

Table 1: ETSI 303645 terminology

administrator	user who has the highest-privilege level possible for a user of the device, which can mean they are able to change any configuration related to the intended functionality.
associated services	digital services that, together with the device, are part of the overall consumer IoT product and that are typically required to provide the product's intended functionality.
authentication mechanism	method used to prove the authenticity of an entity.
authentication value	individual value of an attribute used by an authentication mechanism.
best practice cryptography	cryptography that is suitable for the corresponding use case and has no indications of a feasible attack with current readily available techniques.
constrained device	device which has physical limitations in either the ability to process data, the ability to communicate data, the ability to store data or the ability to interact with the user, due to restrictions that arise from its intended use.
consumer	natural person who is acting for purposes that are outside her/his trade, business, craft or profession.

continues on next page

Table 1 – continued from previous page

consumer IoT device	network-connected (and network-connectable) device that has relationships to associated services and are used by the consumer typically in the home or as electronic wearables.
critical security parameter	security-related secret information whose disclosure or modification can compromise the security of a security module.
debug interface	physical interface used by the manufacturer to communicate with the device during development or to perform triage of issues with the device and that is not used as part of the consumer-facing functionality
defined support period	minimum length of time, expressed as a period or by an end-date, for which a manufacturer will provide security updates.
device manufacturer	entity that creates an assembled final consumer IoT product, which is likely to contain the products and components of many other suppliers.
factory default	state of the device after factory reset or after final production/assembly.
initialization	process that activates the network connectivity of the device for operation and optionally sets authentication features for a user or for network access.
initialized state	state of the device after initialization.
IoT product	consumer IoT device and its associated services.
isolable	able to be removed from the network it is connected to, where any functionality loss caused is related only to that connectivity and not to its main function; alternatively, able to be placed in a self-contained environment with other devices if and only if the integrity of devices within that environment can be ensured
logical interface	software implementation that utilizes a network interface to communicate over the network via channels or ports.
manufacturer	relevant economic operator in the supply chain (including the device manufacturer).
network interface	physical interface that can be used to access the functionality of consumer IoT via a network.
owner	user who owns or who purchased the device.
personal data	Any information relating to an identified or identifiable natural person.
physical interface	physical port or air interface (such as radio, audio or optical) used to communicate with the device at the physical layer.
public security parameter	security related public information whose modification can compromise the security of a security module.
remotely accessible	intended to be accessible from outside the local network.
security module	set of hardware, software, and/or firmware that implements security functions.
security update	software update that addresses security vulnerabilities either discovered by or reported to the manufacturer.
sensitive security parameters	critical security parameters and public security parameters.
software service	software component of a device that is used to support functionality.
telemetry	data from a device that can provide information to help the manufacturer identify issues or information related to device usage.
unique per device	unique for each individual device of a given product class or type.
user	natural person or organization.

Provisions Assessment

The following table is a self-assessment using Table B.1 from ETSI EN 303 645, specifically focusing on the Zephyr RTOS as a component within IoT products.

According with ETSI 303 645, table B.1 provides a mechanism to give information about the implementation of the provisions presented in the standard. Zephyr has adopted the following notations used in the standard:

Table 2: ETSI 303645 Table B.1 notations

M	the provision is a mandatory requirement
R	the provision is a recommendation
M C	the provision is a mandatory requirement and conditional
R C	the provision is a recommendation and conditional
Y	The provision is supported by Zephyr
N	The provision is not supported by Zephyr
N/A	The provision is not applicable to Zephyr or it is product makers responsibility

Table 3: ETSI 303645 provisions assessment using table B.1

Provision	Description	Status	Support	Detail
Provision 5.1-1	Where passwords are used and in any state other than the factory default, all consumer IoT device passwords shall be unique per device or defined by the user.	M C	N/A	
Provision 5.1-2	Where pre-installed unique per device passwords are used, these shall be generated with a mechanism that reduces the risk of automated attacks against a class or type of device.	M C	N/A	
Provision 5.1-3	Authentication mechanisms used to authenticate users against a device shall use best practice cryptography, appropriate to the properties of the technology, risk and usage.	M	N/A	
Provision 5.1-4	Where a user can authenticate against a device, the device shall provide to the user or an administrator a simple mechanism to change the authentication value used.	M C	N/A	
Provision 5.1-5	When the device is not a constrained device, it shall have a mechanism available which makes brute-force attacks on authentication mechanisms via network interfaces impracticable.	M C	N	**TODO**
Provision 5.2-1	The manufacture shall make a vulnerability disclosure policy publicly available.	M	Y	*Vulnerability Management*
Provision 5.2-2	Disclosed vulnerabilities should be acted on in a timely manner.	R	Y	*Vulnerability Timeline*
Provision 5.2-3	Manufacturers should continually monitor for, identify and rectify security vulnerabilities within products and services they sell, produce, have produced and services they operate during the defined support period.	R	Y	Modules are covered
Provision 5.3-1	All software components in consumer IoT devices should be securely updatable.	R	Y	*Device firwmware upgrade*
Provision 5.3-2	When the device is not a constrained device, it shall have an update mechanism for the secure installation of updates.	M C	Y	*Device firwmware upgrade*

continues on next page

Table 3 – continued from previous page

Provision	Description	Status	Support	Detail
Provision 5.3-3	An update shall be simple for the user to apply.	M C	N/A	
Provision 5.3-4	Automatic mechanisms should be used for software updates.	R C	N/A	
Provision 5.3-5	The device should check after initialization, and then periodically, whether security updates are available.	R C	N/A	
Provision 5.3-6	If the device supports automatic updates and/or update notifications, these should be enabled in the initialized state and configurable so that the user can enable, disable, or postpone installation of security updates and/or update notifications.	R C	N/A	
Provision 5.3-7	The device shall use best practice cryptography to facilitate secure update mechanisms.	M C	Y	*West Sign*
Provision 5.3-8	Security updates shall be timely.	M C	N/A	
Provision 5.3-9	The device should verify the authenticity and integrity of software updates.	R C	Y	Functionality provided by *MCUboot <https://github.com/zephyrproject-rtos/mcuboot>*. Also see *Device Firwmware Upgrade*
Provision 5.3-10	Where updates are delivered over a network interface, the device shall verify the authenticity and integrity of each update via a trust relationship.	M	N/A	
Provision 5.3-11	The manufacturer should inform the user in a recognizable and apparent manner that a security update is required together with information on the risks mitigated by that update.	R C	N/A	
Provision 5.3-12	The device should notify the user when the application of a software update will disrupt the basic functioning of the device.	R C	N/A	Zephyr provides this information for its updates. Anyone using Zephyr in their products must check if they are affected
Provision 5.3-13	The manufacturer shall publish, in an accessible way that is clear and transparent to the user, the defined support period.	M	Y	Release Life Cycle and Maintenance

continues on next page

Table 3 – continued from previous page

Provision	Description	Status	Support	Detail
Provision 5.3-14	For constrained devices that cannot have their software updated, the rationale for the absence of software updates, the period and method of hardware replacement support and a defined support period should be published by the manufacturer in an accessible way that is clear and transparent to the user.	R C	N/A	
Provision 5.3-15	For constrained devices that cannot have their software updated, the product should be isolable and the hardware replaceable.	R C	N/A	
Provision 5.3-16	The model designation of the consumer IoT device shall be clearly recognizable, either by labelling on the device or via a physical interface.	M	N/A	
Provision 5.4-1	Sensitive security parameters in persistent storage shall be stored securely by the device.	M	N	There is not secure storage within Zephyr
Provision 5.4-2	Where a hard-coded unique per device identity is used in a device for security purposes, it shall be implemented in such a way that it resists tampering by means such as physical, electrical or software.	M C	N/A	
Provision 5.4-3	Hard-coded critical security parameters in device software source code shall not be used.	M	Y	*Hardening Tool*
Provision 5.4-4	Any critical security parameters used for integrity and authenticity checks of software updates and for protection of communication with associated services in device software shall be unique per device and shall be produced with a mechanism that reduces the risk of automated attacks against classes of devices.	M	N/A	
Provision 5.5-1	The consumer IoT device shall use best practice cryptography to communicate securely.	M	Y	
Provision 5.5-2	The consumer IoT device should use reviewed or evaluated implementations to deliver network and security functionalities, particularly in the field of cryptography.	R	Y	
Provision 5.5-3	Cryptographic algorithms and primitives should be updatable.	R	N	The whole image must be updated

continues on next page

Table 3 – continued from previous page

Provision	Description	Status	Support	Detail
Provision 5.5-4	Access to device functionality via a network interface in the initialized state should only be possible after authentication on that interface.	R	N/A	
Provision 5.5-5	Device functionality that allows security-relevant changes in configuration via a network interface shall only be accessible after authentication. The exception is for network service protocols that are relied upon by the device and where the manufacturer cannot guarantee what configuration will be required for the device to operate.	M	N/A	
Provision 5.5-6	Critical security parameters should be encrypted in transit, with such encryption appropriate to the properties of the technology, risk and usage.	R	Y	
Provision 5.5-7	The consumer IoT device shall protect the confidentiality of critical security parameters that are communicated via remotely accessible network interfaces.	M	Y	
Provision 5.5-8	The manufacturer shall follow secure management processes for critical security parameters that relate to the device.	M	N/A	
Provision 5.6-1	All unused network and logical interfaces shall be disabled.	M	Y	*Kconfig*
Provision 5.6-2	In the initialized state, the network interfaces of the device shall minimize the unauthenticated disclosure of security-relevant information.	M	Y	
Provision 5.6-3	Device hardware should not unnecessarily expose physical interfaces to attack.	R	Y	*Kconfig* and *Hardening Tool*
Provision 5.6-4	Where a debug interface is physically accessible, it shall be disabled in software.	M C	Y	*Hardening Tool*
Provision 5.6-5	The manufacturer should only enable software services that are used or required for the intended use or operation of the device.	R	Y	*Kconfig* and *Hardening Tool*
Provision 5.6-6	Code should be minimized to the functionality necessary for the service/device to operate.	R	Y	*Kconfig*

continues on next page

Table 3 – continued from previous page

Provision	Description	Status	Support	Detail
Provision 5.6-7	Software should run with least necessary privileges, taking account of both security and functionality.	R	Y	*Security Overview*
Provision 5.6-8	The device should include a hardware-level access control mechanism for memory.	R	Y	*Memory protection*
Provision 5.6-9	The manufacturer should follow secure development processes for software deployed on the device.	R	Y	*Security Overview* and *Coding guidelines*
Provision 5.7-1	The consumer IoT device should verify its software using secure boot mechanisms.	R	Y	Functionality provided by *MCUboot <https://github.com/zephyrproject-rtos/mcuboot>*. Also see *Security Overview*
Provision 5.7-2	If an unauthorized change is detected to the software, the device should alert the user and/or administrator to the issue and should not connect to wider networks than those necessary to perform the alerting function.	R	N	Zephyr does not provide runtime detection / notification.
Provision 5.8-1	The confidentiality of personal data transiting between a device and a service, especially associated services, should be protected, with best practice cryptography.	R	Y	
Provision 5.8-2	The confidentiality of sensitive personal data communicated between the device and associated services shall be protected, with cryptography appropriate to the properties of the technology and usage.	M	Y	
Provision 5.8-3	All external sensing capabilities of the device shall be documented in an accessible way that is clear and transparent for the user.	M	Y	*Sensing Subsystem*
Provision 5.9-1	Resilience should be built in to consumer IoT devices and services, taking into account the possibility of outages of data networks and power.	R	Y	
Provision 5.9-2	Consumer IoT devices should remain operating and locally functional in the case of a loss of network access and should recover cleanly in the case of restoration of a loss of power.	R	Y	

continues on next page

Table 3 – continued from previous page

Provision	Description	Status	Support	Detail
Provision 5.9-3	The consumer IoT device should connect to networks in an expected, operational and stable state and in an orderly fashion, taking the capability of the infrastructure into consideration.	R	Y	
Provision 5.10-1	If telemetry data is collected from consumer IoT devices and services, such as usage and measurement data, it should be examined for security anomalies.	R C	N/A	
Provision 5.11-1	The user shall be provided with functionality such that user data can be erased from the device in a simple manner.	M	N/A	
Provision 5.11-2	The consumer should be provided with functionality on the device such that personal data can be removed from associated services in a simple manner.	R	N/A	
Provision 5.11-3	Users should be given clear instructions on how to delete their personal data.	R	N/A	
Provision 5.11-4	Users should be provided with clear confirmation that personal data has been deleted from services, devices and applications.	R	N/A	
Provision 5.12-1	Installation and maintenance of consumer IoT should involve minimal decisions by the user and should follow security best practice on usability.	R	N/A	
Provision 5.12-2	The manufacturer should provide users with guidance on how to securely set up their device.	R	N/A	
Provision 5.12-3	The manufacturer should provide users with guidance on how to check whether their device is securely set up.	R	N/A	
Provision 5.13-1	The consumer IoT device software shall validate data input via user interfaces or transferred via Application Programming Interfaces (APIs) or between networks in services and devices.	M	Y	*Syscall verification* and *Coding guidelines*

continues on next page

Table 3 – continued from previous page

Provision	Description	Status	Support	Detail
Provision 6.1-1	The manufacturer shall provide consumers with clear and transparent information about what personal data is processed, how it is being used, by whom, and for what purposes, for each device and service. This also applies to third parties that can be involved, including advertisers.	M	N/A	
Provision 6.1-2	Where personal data is processed on the basis of consumers' consent, this consent shall be obtained in a valid way.	M C	N/A	
Provision 6.1-3	Consumers who gave consent for the processing of their personal data shall have the capability to withdraw it at any time.	M	N/A	
Provision 6.1-4	If telemetry data is collected from consumer IoT devices and services, the processing of personal data should be kept to the minimum necessary for the intended functionality.	R C	N/A	
Provision 6.1-5	If telemetry data is collected from consumer IoT devices and services, consumers shall be provided with information on what telemetry data is collected, how it is being used, by whom, and for what purposes.	M C	N/A	

Chapter 11

Safety

These documents describe the processes, developer guidelines and requirements for ensuring safety is addressed within the Zephyr project.

11.1 Zephyr Safety Overview

11.1.1 Introduction

This document is the safety documentation providing an overview over the safety-relevant activities and what the Zephyr Project and the Zephyr Safety Working Group / Committee try to achieve.

This overview is provided for people who are interested in the functional safety development part of the Zephyr RTOS and project members who want to contribute to the safety aspects of the project.

11.1.2 Overview

In this section we give the reader an overview of what the general goal of the safety certification is, what standard we aim to achieve and what quality standards and processes need to be implemented to reach such a safety certification.

11.1.3 Safety Document update

This document is a living document and may evolve over time as new requirements, guidelines, or processes are introduced.

1. Changes will be submitted from the interested party(ies) via pull requests to the Zephyr documentation repository.
2. The Zephyr Safety Committee will review these changes and provide feedback or acceptance of the changes.
3. Once accepted, these changes will become part of the document.

11.1.4 General safety scope

The general scope of the Safety Committee is to achieve a certification for the IEC 61508 standard and the Safety Integrity Level (SIL) 3 / Systematic Capability (SC) 3 for a limited source scope (see certification scope TBD). Since the code base is pre-existing, we use the route 3s/1s approach defined by the IEC 61508 standard.

Route 3s
Assessment of non-compliant development. Which is basically the route 1s with existing sources.

Route 1s
Compliant development. Compliance with the requirements of this standard for the avoidance and control of systematic faults in software.

Summarization IEC 61508 standard

The IEC 61508 standard is a widely recognized international standard for functional safety of electrical, electronic, and programmable electronic safety-related systems. Here's an overview of some of the key safety aspects of the standard:

1. **Hazard and Risk Analysis**: The IEC 61508 standard requires a thorough analysis of potential hazards and risks associated with a system in order to determine the appropriate level of safety measures needed to reduce those risks to acceptable levels.
2. **Safety Integrity Level (SIL)**: The standard introduces the concept of Safety Integrity Level (SIL) to classify the level of risk reduction required for each safety function. The higher the SIL, the greater the level of risk reduction required.
3. **System Design**: The IEC 61508 standard requires a systematic approach to system design that includes the identification of safety requirements, the development of a safety plan, and the use of appropriate safety techniques and measures to ensure that the system meets the required SIL.
4. **Verification and Validation**: The standard requires rigorous testing and evaluation of the safety-related system to ensure that it meets the specified SIL and other safety requirements. This includes verification of the system design, validation of the system's functionality, and ongoing monitoring and maintenance of the system.
5. **Documentation and Traceability**: The IEC 61508 standard requires a comprehensive documentation process to ensure that all aspects of the safety-related system are fully documented and that there is full traceability from the safety requirements to the final system design and implementation.

Overall, the IEC 61508 standard provides a framework for the design, development, and implementation of safety-related systems that aims to reduce the risk of accidents and improve overall safety. By following the standard, organizations can ensure that their safety-related systems are designed and implemented to the highest level of safety integrity.

11.1.5 Quality

Quality is a mandatory expectation for software across the industry. The code base of the project must achieve various software quality goals in order to be considered an auditable code base from a safety perspective and to be usable for certification purposes. But software quality is not an additional requirement caused by functional safety standards. Functional safety considers quality as an existing pre-condition and therefore the "quality managed" status should be pursued for any project regardless of the functional safety goals. The following list describes the quality goals which need to be reached to achieve an auditable code base:

1. Basic software quality standards

   a. *Coding Guidelines* (including: static code analysis, coding style, etc.)
   b. *Safety Requirements* and requirements tracing
   c. Test coverage
2. Software architecture design principles
   a. Layered architecture model
   b. Encapsulated components
   c. Encapsulated single functionality (if not fitable and manageable in safety)

Basic software quality standards - Safety view

In this chapter the Safety Committee describes why they need the above listed quality goals as pre-condition and what needs to be done to achieve an auditable code base from the safety perspective. Generally speaking, it can be said that all of these quality measures regarding safety are used to minimize the error rate during code development.

Coding Guidelines The coding guidelines are the basis to a common understanding and a unified ruleset and development style for industrial software products. For safety the coding guidelines are essential and have another purpose beside the fact of a unified ruleset. It is also necessary to prove that the developers follow a unified development style to prevent **systematic errors** in the process of developing software and thus to minimize the overall **error rate** of the complete software system.

Also the **IEC 61508 standard** sets a pre-condition and recommendation towards the use of coding standards / guidelines to reduce likelihood of errors.

Requirements and requirements tracing Requirements and requirement management are not only important for software development, but also very important in terms of safety. On the one hand, this specifies and describes in detail and on a technical level what the software should do, and on the other hand, it is an important and necessary tool to verify whether the described functionality is implemented as expected. For this purpose, tracing the requirements down to the code level is used. With the requirements management and tracing in hand, it can now be verified whether the functionality has been tested and implemented correctly, thus minimizing the systematic error rate.

Also the IEC 61508 standard highly recommends (which is like a must-have for the certification) requirements and requirements tracing.

Test coverage A high test coverage, in turn, is evidence of safety that the code conforms precisely to what it was developed for and does not execute any unforeseen instructions. If the entire code is tested and has a high (ideally 100%) test coverage, it has the additional advantage of quickly detecting faulty changes and further minimizing the error rate. However, it must be noted that different requirements apply to safety for test coverage, and various metrics must be considered, which are prescribed by the IEC 61508 standard for the SIL 3 / SC3 target. The following must be fulfilled, among other things:

- Structural test coverage (entry points) 100%
- Structural test coverage (statements) 100%
- Structural test coverage (branches) 100%

If the 100% cannot be reached (e.g. statement coverage of defensive code) that part needs to be described and justified in the documentation.

Software architecture design principles

To create and maintain a structured software product it is also necessary to consider individual software architecture designs and implement them in accordance with safety standards because some designs and implementations are not reasonable in safety, so that the overall software and code base can be used as auditable code. However, most of these software architecture designs have already been implemented in the Zephyr project and need to be verified by the Safety Committee / Safety Working Group and the safety architect.

Layered architecture model The **IEC 61508 standard** strongly recommends a modular approach to software architecture. This approach has been pursued in the Zephyr project from the beginning with its layered architecture. The idea behind this architecture is to organize modules or components with similar functionality into layers. As a result, each layer can be assigned a specific role in the system. This model has the advantage in safety that interfaces between different components and layers can be shown at a very high level, and thus it can be determined which functionalities are safety-relevant and can be limited. Furthermore, various analyses and documentations can be built on top of this architecture, which are important for certification and the responsible certification body.

Encapsulated components Encapsulated components are an essential part of the architecture design for safety at this point. The most important aspect is the separation of safety-relevant components from non-safety-relevant components, including their associated interfaces. This ensures that the components have no **repercussions** on other components.

Encapsulated single functionality (if not reasonable and manageable in safety) Another requirement for the overall system and software environment is that individual functionalities can be disabled within components. This is because if a function is absolutely unacceptable for safety (e.g. complete dynamic memory management), then these individual functionalities should be able to be turned off. The Zephyr Project already offers such a possibility through the use of Kconfig and its flexible configurability.

11.1.6 Processes and workflow

The diagram describes the rough process defined by the Safety Committee to ensure safety in the development of the Zephyr project. To ensure understanding, a few points need to be highlighted and some details explained regarding the role of the safety architect and the role of the safety committee in the whole process. The diagram only describes the paths that are possible when a change is related to safety.

1. On the main branch, the safety scope of the project should be identified, which typically represents a small subset of the entire code base. This subset should then be made auditable during normal development on “main”, which means that special attention is paid to quality goals (*Quality*) and safety processes within this scope. The Safety Architect works alongside the Technical Steering Committee (TSC) in this area, monitoring the development process to ensure that the architecture meets the safety requirements.
2. At this point, the safety architect plays an increasingly important role. For PRs/issues that fall within the safety scope, the safety architect should ideally be involved in the discussions and decisions of minor changes in the safety scope to be able to react to safety-relevant changes that are not conformant. If a pull request or issue introduces a significant and influential change or improvement that requires extended discussion or decision-making, the safety architect should bring it to the attention of the Safety Committee or the Technical Steering Committee (TSC) as appropriate, so that they can make a decision on the best course of action.

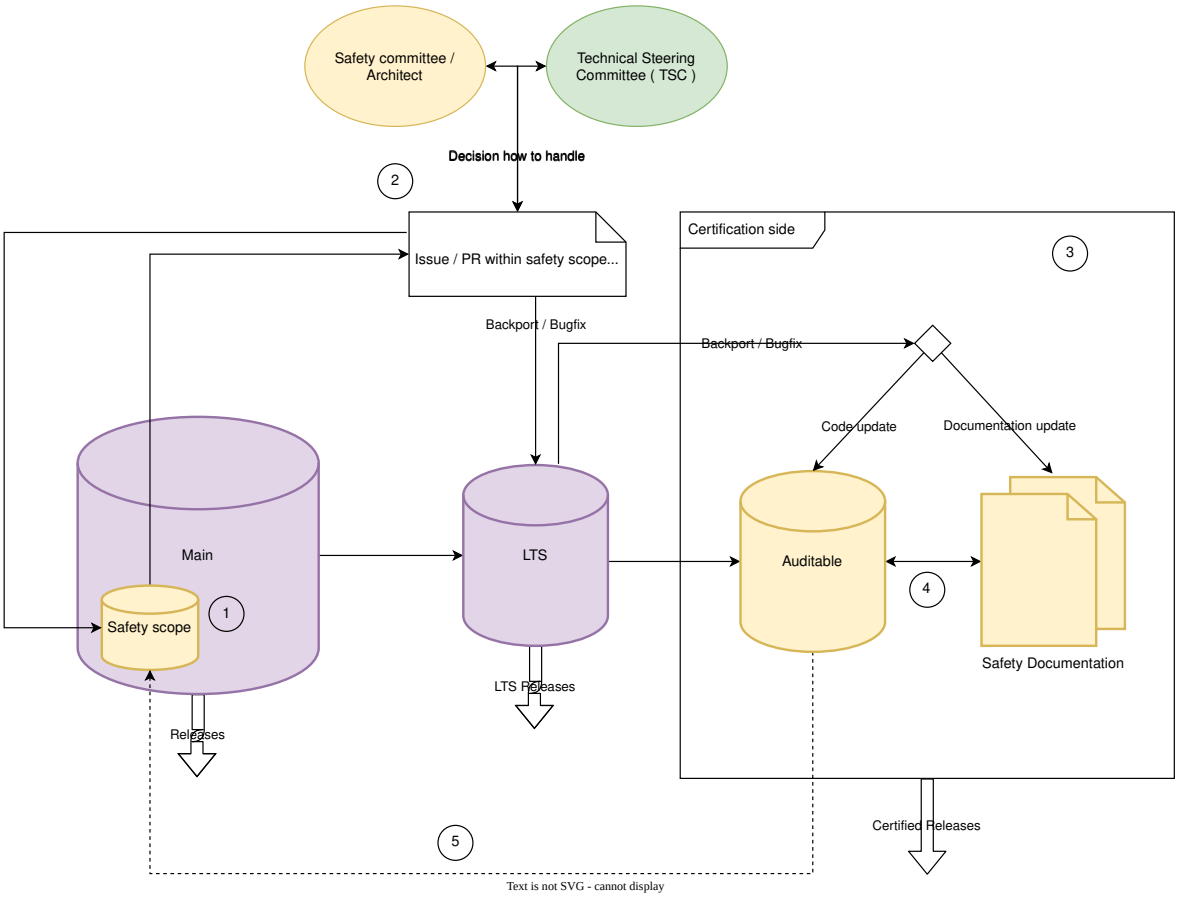


Fig. 1: Safety process and workflow overview

3. This section describes the certification side. At this point, the code base has to be in an "auditable" state, and ideally no further changes should be necessary or made to the code base. There is still a path from the main branch to this area. This is needed in case a serious bug or important change is found or implemented on the main branch in the safety scope, after the LTS and the auditable branch were created. In this case, the Safety Committee, together with the safety architect, must decide whether this bug fix or change should be integrated into the LTS so that the bug fix or change could also be integrated into the auditable branch. This integration can take three forms: First either as only a code change or second as only an update to the safety documentation or third as both.
4. This describes the necessary safety process required for certification itself. Here, the final analyses, tests, and documents are created and conducted which must be created and conducted during the certification, and which are prescribed by the certifying authority and the standard being certified. If the certification body approves everything at this stage and the safety process is completed, a safety release can be created and published.
5. This transition from the auditable branch to the main branch should only occur in exceptional circumstances, specifically when something has been identified during the certification process that needs to be quickly adapted on the "auditable" branch in order to obtain certification. In order to prevent this issue from arising again during the next certification, there needs to be a path to merge these changes back into the main branch so that they are not lost, and to have them ready for the next certification if necessary.

Important

Safety should not block the project and minimize the room to grow in any way.

Important

TODO: Find and define ways, guidelines and processes which minimally impact the daily work of the maintainers, reviewers and contributors and also the safety architect itself. But which are also suitable for safety.

11.2 Safety Requirements

11.2.1 Introduction

The safety committee leads the effort to gather requirements that reflect the **actual** state of the implementation, following the route 3s approach of the project's safety effort. The goal is **NOT** to create new requirements to request additional features for the project.

The requirements are gathered in the separate repository: Requirement repository

11.2.2 Guidelines

Below are the guidelines for the requirements repository and the expectations of the safety committee when adding requirements to the repository.

Scope

The scope of the requirements covers the KERNEL functionalities.

Consistency

Maintain consistency across all requirements. The language and choice of words shall be consistent. (See: *Syntax*)

Levels of requirements in the repository

System Requirements
: System requirements describe the behaviour of the Zephyr RTOS (= the system here). They describe the functionality of the Zephyr RTOS from a very high-level perspective, without going into details of the functionality itself. The purpose of the system requirements is to get an overview of the currently implemented features of the Zephyr RTOS. In other words a person writing these requirements usually has some knowledge of the Zephyr RTOS Project as the requirements are specific to an RTOS.

Software Requirements
: Software requirements refine the system-level requirements at a more granular level so that each requirement can be tested. These requirements define the specific actions the feature shall be able to execute and the behavior of the feature.

Requirement UID (Unique identifier) Handling

The tool used to manage requirements, strictDoc, is responsible for handling the Unique Identifier (UID) associated with each requirement. To manage UIDs, follow these steps:

1. Don't add a requirement UID and UID field for new requirements
2. After completing work on the new requirements execute: `strictDoc manage auto-uid .`
3. Establish links between the requirements with the new attributed UIDs if needed

After doing this, the requirements are ready and a pull request can be created. The CI in the PR will check if the requirements UIDs are valid or if there are duplicates in it. If there are duplicates in the PR, these need to be resolved by rebasing and re-executing the steps above.

Requirement Types

- Functional
- Non-Functional

Requirement title convention

Use short and succinct requirement titles.

Pull Request requirement repository

- Adhere to the *Contribution Guidelines* of the Zephyr Project.
  - As long as they are applicable to the requirements repository.
- Avoid creating large commits that contain both trivial and non-trivial changes.
- Avoid moving and changing requirements in the same commit.

Characteristics of a good requirement

- Unambiguous
- Verifiable (e.g. testable for functional requirements)
- Clear (concise, succinct, simple, precise)
- Correct
- Understandable
- Feasible (realistic, possible)
- Independent
- Atomic
- Necessary
- Implementation-free (abstract)

Characteristics of a set of requirements

- Complete
- Consistent
- Non redundant

Syntax

- Use of a recognized Requirements Syntax is recommended.
  - EARS is a good reference. Particularly if you are unfamiliar with requirements writing.
  - Other formats are accepted as long as the characteristics of a requirement from above are met.

Chapter 12

Glossary of Terms

API
: (Application Program Interface) A defined set of routines and protocols for building application software.

application
: The set of user-supplied files that the Zephyr build system uses to build an application image for a specified board configuration. It can contain application-specific code, kernel configuration settings, and at least one CMakeLists.txt file. The application's kernel configuration settings direct the build system to create a custom kernel that makes efficient use of the board's resources. An application can sometimes be built for more than one type of board configuration (including boards with different CPU architectures), if it does not require any board-specific capabilities.

application image
: A binary file that is loaded and executed by the board for which it was built. Each application image contains both the application's code and the Zephyr kernel code needed to support it. They are compiled as a single, fully-linked binary. Once an application image is loaded onto a board, the image takes control of the system, initializes it, and runs as the system's sole application. Both application code and kernel code execute as privileged code within a single shared address space.

architecture
: An instruction set architecture (ISA) along with a programming model.

board
: A target system with a defined set of devices and capabilities, which can load and execute an application image. It may be an actual hardware system or a simulated system running under QEMU. A board can contain one or more *SoCs*. The Zephyr kernel supports a variety of boards.

board configuration
: A set of kernel configuration options that specify how the devices present on a board are used by the kernel. The Zephyr build system defines one or more board configurations for each board it supports. The kernel configuration settings that are specified by the build system can be over-ridden by the application, if desired.

board name
: The human-readable name of a *board*. Uniquely and descriptively identifies a particular system, but does not include additional information that may be required to actually build a Zephyr image for it. See *Board terminology* for additional details.

board qualifiers
: The set of additional tokens, separated by a forward slash (/) that follow the *board name* (and optionally *board revision*) to form the *board target*. The currently accepted qualifiers are *SoC*, *CPU cluster* and *variant*. See *Board terminology* for additional details.

board revision
An optional version string that identifies a particular revision of a hardware system. This is useful to avoid duplication of board files whenever small changes are introduced to a hardware system. See *Multiple board revisions* and *Building for a board revision* for more information.

board target
The full string that can be provided to any of the Zephyr build tools to compile and link an image for a particular hardware system. This string uniquely identifies the combination of *board name*, *board revision* and *board qualifiers*. See *Board terminology* for additional details.

CPU cluster
A group of one or more *CPU cores*, all executing the same image within the same address space and in a symmetrical (SMP) configuration. Only *CPU cores* of the same *architecture* can be in a single cluster. Multiple CPU clusters (each of one or more cores) can coexist in the same *SoC*.

CPU core
A single processing unit, with its own Program Counter, executing program instructions sequentially. CPU cores are part of a *CPU cluster*, which can contain one or more cores.

device runtime power management
Device Runtime Power Management (PM) refers the capability of devices to save energy independently of the system power state. Devices will keep reference of their usage and will automatically be suspended or resumed. This feature is enabled via the `CONFIG_PM_DEVICE_RUNTIME` Kconfig option.

idle thread
A system thread that runs when there are no other threads ready to run.

IDT
(Interrupt Descriptor Table) a data structure used by the x86 architecture to implement an interrupt vector table. The IDT is used to determine the correct response to interrupts and exceptions.

ISR
(Interrupt Service Routine) Also known as an interrupt handler, an ISR is a callback function whose execution is triggered by a hardware interrupt (or software interrupt instructions) and is used to handle high-priority conditions that require interrupting the current code executing on the processor.

kernel
The set of Zephyr-supplied files that implement the Zephyr kernel, including its core services, device drivers, network stack, and so on.

power domain
A power domain is a collection of devices for which power is applied and removed collectively in a single action. Power domains are represented by *device*.

power gating
Power gating reduces power consumption by shutting off areas of an integrated circuit that are not in use.

SoC
A System on a chip, that is, an integrated circuit that contains at least one *CPU cluster* (in turn with at least one *CPU core*), as well as peripherals and memory.

SoC family
One or more *SoCs* or *SoC series* that share enough in common to consider them related and under a single family denomination.

SoC series
A number of different *SoCs* that share similar characteristics and features, and that the vendor typically names and markets together.

subsystem
: A subsystem refers to a logically distinct part of the operating system that handles specific functionality or provides certain services.

system power state
: System power states describe the power consumption of the system as a whole. System power states are represented by *pm_state*.

variant
: In the context of *board qualifiers*, a variant designates a particular type or configuration of a build for a combination of *SoC* and *CPU cluster*. Common uses of the variant concept include introducing both secure and non-secure builds for platforms with Trusted Execution Environment support, or selecting the type of RAM used in a build.

west
: A multi-repo meta-tool developed for the Zephyr project. See *West (Zephyr's meta-tool)*.

west installation
: An obsolete term for a *west workspace* used prior to west 0.7.

west manifest
: A YAML file, usually named `west.yml`, which describes projects, or the Git repositories which make up a *west workspace*, along with additional metadata. See *Basics* for general information and *West Manifests* for details.

west manifest repository
: The Git repository in a *west workspace* which contains the *west manifest*. Its location is given by the *manifest.path configuration option*. See *Basics*.

west project
: Each of the entries in a *west manifest*, which describe a Git repository that will be cloned and managed by west when working with the corresponding *west manifest repository*. Note that a west project is different from a *zephyr module*, although many projects are also modules. See *Projects* for additional information.

west workspace
: A folder on your system with a `.west` subdirectory and a *west manifest repository* in it. You clone the Zephyr source code, as well as that of its *west projects* onto your system by creating a west workspace using the `west init` command. See *Basics*.

XIP
: (eXecute In Place) a method of executing programs directly from long term storage rather than copying it into RAM, saving writable memory for dynamic data and not the static program code.

zephyr module
: A Git repository containing a `zephyr/module.yml` file, used by the Zephyr build system to integrate the source code and configuration files of the module into a regular Zephyr build. Zephyr modules may be west projects, but they do not have to. See *Modules (External projects)* for additional details.

Bibliography

[a] Real Time Counter (RTC)

[b] Programmable Peripheral Interconnect (PPI)

[c] Distributed Programmable Peripheral Interconnect (DPPI)

[d] Software Interrupt (SWI)

[e] Random Number Generator (RNG)

[f] AES Electronic Codebook Mode Encryption (ECB)

[g] Cipher Block Chaining (CBC) - Message Authentication Code with Counter Mode encryption (CCM)

[h] Accelerated Address Resolver (AAR)

[i] General Purpose Input Output (GPIO)

[j] GPIO tasks and events (GPIOTE)

[k] Temperature sensor (TEMP)

[l] Universal Asynchronous Receiver Transmitter (UART)

[m] Interprocess Communication peripheral (IPC)

[th-imboot] Must boot with an immutable bootloader.

[th-authrepl] Application image shall only be replaced with an authorized image.

[th-timely-update] Application updates shall be done in a timely manner.

[th-atomic-update] Application updates shall be atomic.

[th-root-certs] TLS must have a list of trusted root certificates.

[th-root-check] TLS must verify root certificate from server is valid.

[th-secret-storage] There must be a mechanism to securely store client secrets. The least amount of code necessary shall have access to these secrets.

[th-time] System must have moderately accurate notion of the current date/time.

[th-conf] The system must receive, and keep configuration data.

[th-logs] The system must log security-related events, and either store them locally, or send to a service.

[th-all-tls] All communications with the cloud service shall use TLS.

[th-tls-ciphers] TLS shall be configured to allow only generally agreed cipher suites (including forward secrecy).

[th-tls-client-auth] The device shall authenticate itself with the cloud provider using one of the methods described.

[th-entropy] The TLS layer shall use a modern, accepted cryptographic random-bit generator seeded by an entropy source within the SoC.

[th-initial-provision] The device shall have a per-device secret loaded before deployment.

[th-initial-secret] The initial secret shall be securely maintained, and destroyed in any external location as soon as the device is provisioned.

[th-reprovision] Reprovisioning a device shall be done securely.

[th-destruction] Upon decommissioning, the device secret shall be rendered ineffective.

Python Module Index

r
runners.core, 195

Index

A

accept (*C function*), 2488
adc_action (*C enum*), 3165
adc_action.ADC_ACTION_CONTINUE (*C enumerator*), 3165
adc_action.ADC_ACTION_FINISH (*C enumerator*), 3165
adc_action.ADC_ACTION_REPEAT (*C enumerator*), 3165
adc_api_channel_setup (*C type*), 3163
adc_api_read (*C type*), 3163
adc_api_read_async (*C type*), 3163
adc_channel_cfg (*C struct*), 3169
ADC_CHANNEL_CFG_DT (*C macro*), 3158
adc_channel_cfg.acquisition_time (*C var*), 3169
adc_channel_cfg.channel_id (*C var*), 3169
adc_channel_cfg.differential (*C var*), 3169
adc_channel_cfg.gain (*C var*), 3169
adc_channel_cfg.reference (*C var*), 3169
adc_channel_setup (*C function*), 3166
adc_channel_setup_dt (*C function*), 3166
adc_driver_api (*C struct*), 3172
adc_dt_spec (*C struct*), 3170
ADC_DT_SPEC_GET (*C macro*), 3162
ADC_DT_SPEC_GET_BY_IDX (*C macro*), 3160
ADC_DT_SPEC_GET_BY_NAME (*C macro*), 3159
ADC_DT_SPEC_INST_GET (*C macro*), 3162
ADC_DT_SPEC_INST_GET_BY_IDX (*C macro*), 3161
ADC_DT_SPEC_INST_GET_BY_NAME (*C macro*), 3160
adc_dt_spec.channel_cfg (*C var*), 3170
adc_dt_spec.channel_cfg_dt_node_exists (*C var*), 3170
adc_dt_spec.channel_id (*C var*), 3170
adc_dt_spec.dev (*C var*), 3170
adc_dt_spec.oversampling (*C var*), 3170
adc_dt_spec.resolution (*C var*), 3170
adc_dt_spec.vref_mv (*C var*), 3170
adc_gain (*C enum*), 3163
adc_gain_invert (*C function*), 3165
adc_gain.ADC_GAIN_1 (*C enumerator*), 3164
adc_gain.ADC_GAIN_1_2 (*C enumerator*), 3163
adc_gain.ADC_GAIN_1_3 (*C enumerator*), 3163
adc_gain.ADC_GAIN_1_4 (*C enumerator*), 3163
adc_gain.ADC_GAIN_1_5 (*C enumerator*), 3163
adc_gain.ADC_GAIN_1_6 (*C enumerator*), 3163
adc_gain.ADC_GAIN_2 (*C enumerator*), 3164
adc_gain.ADC_GAIN_2_3 (*C enumerator*), 3164
adc_gain.ADC_GAIN_2_5 (*C enumerator*), 3163
adc_gain.ADC_GAIN_3 (*C enumerator*), 3164
adc_gain.ADC_GAIN_4 (*C enumerator*), 3164

adc_gain.ADC_GAIN_4_5 (*C enumerator*), 3164
adc_gain.ADC_GAIN_6 (*C enumerator*), 3164
adc_gain.ADC_GAIN_8 (*C enumerator*), 3164
adc_gain.ADC_GAIN_12 (*C enumerator*), 3164
adc_gain.ADC_GAIN_16 (*C enumerator*), 3164
adc_gain.ADC_GAIN_24 (*C enumerator*), 3164
adc_gain.ADC_GAIN_32 (*C enumerator*), 3164
adc_gain.ADC_GAIN_64 (*C enumerator*), 3164
adc_gain.ADC_GAIN_128 (*C enumerator*), 3164
adc_is_ready_dt (*C function*), 3169
adc_raw_to_millivolts (*C function*), 3167
adc_raw_to_millivolts_dt (*C function*), 3168
adc_read (*C function*), 3166
adc_read_async (*C function*), 3167
adc_read_dt (*C function*), 3167
adc_ref_internal (*C function*), 3167
adc_reference (*C enum*), 3164
adc_reference.ADC_REF_EXTERNAL0 (*C enumerator*), 3165
adc_reference.ADC_REF_EXTERNAL1 (*C enumerator*), 3165
adc_reference.ADC_REF_INTERNAL (*C enumerator*), 3165
adc_reference.ADC_REF_VDD_1 (*C enumerator*), 3164
adc_reference.ADC_REF_VDD_1_2 (*C enumerator*), 3165
adc_reference.ADC_REF_VDD_1_3 (*C enumerator*), 3165
adc_reference.ADC_REF_VDD_1_4 (*C enumerator*), 3165
adc_sequence (*C struct*), 3171
adc_sequence_callback (*C type*), 3162
adc_sequence_init_dt (*C function*), 3168
adc_sequence_options (*C struct*), 3170
adc_sequence_options.callback (*C var*), 3171
adc_sequence_options.extra_samplings (*C var*), 3171
adc_sequence_options.interval_us (*C var*), 3171
adc_sequence_options.user_data (*C var*), 3171
adc_sequence.buffer (*C var*), 3171
adc_sequence.buffer_size (*C var*), 3171
adc_sequence.calibrate (*C var*), 3172
adc_sequence.channels (*C var*), 3171
adc_sequence.options (*C var*), 3171
adc_sequence.oversampling (*C var*), 3172
adc_sequence.resolution (*C var*), 3172
add_parser() (*runners.core.ZephyrBinaryRunner class method*), 199
addrinfo (*C macro*), 2489
AF_CAN (*C macro*), 2508
AF_INET (*C macro*), 2508
AF_INET6 (*C macro*), 2508
AF_LOCAL (*C macro*), 2508
AF_NET_MGMT (*C macro*), 2508
AF_PACKET (*C macro*), 2508
AF_UNIX (*C macro*), 2508
AF_UNSPEC (*C macro*), 2507
AI_ADDRCONFIG (*C macro*), 2492
AI_ALL (*C macro*), 2492
AI_CANONNAME (*C macro*), 2492
AI_EXTFLAGS (*C macro*), 2492
AI_NUMERICHOST (*C macro*), 2492
AI_NUMERICSERV (*C macro*), 2492
AI_PASSIVE (*C macro*), 2492
AI_V4MAPPED (*C macro*), 2492
analog_axis_calibration (*C struct*), 900

analog_axis_calibration_get (*C function*), 899
analog_axis_calibration_save (*C function*), 900
analog_axis_calibration_set (*C function*), 899
analog_axis_calibration.in_deadzone (*C var*), 900
analog_axis_calibration.in_max (*C var*), 900
analog_axis_calibration.in_min (*C var*), 900
analog_axis_num_axes (*C function*), 899
analog_axis_raw_data_t (*C type*), 899
analog_axis_set_raw_data_cb (*C function*), 899
API, **3935**
application, **3935**
application image, **3935**
arch_buffer_validate (*C function*), 3745
arch_busy_wait (*C function*), 3733
arch_cohere_stacks (*C function*), 3746
arch_coredump_info_dump (*C function*), 737
arch_coredump_tgt_code_get (*C function*), 737
arch_cpu_active (*C function*), 3738
arch_cpu_atomic_idle (*C function*), 3737
arch_cpu_idle (*C function*), 3737
arch_cpu_start (*C function*), 3738
arch_cpustart_t (*C type*), 3738
arch_curr_cpu (*C function*), 3738
ARCH_DATA_PAGE_ACCESSED (*C macro*), 3747
ARCH_DATA_PAGE_DIRTY (*C macro*), 3747
ARCH_DATA_PAGE_LOADED (*C macro*), 3748
ARCH_DATA_PAGE_NOT_MAPPED (*C macro*), 3748
arch_elf_relocate (*C function*), 926
arch_elf_relocate_local (*C function*), 927
arch_float_disable (*C function*), 3735
arch_float_enable (*C function*), 3736
arch_gdb_add_breakpoint (*C function*), 3754
arch_gdb_continue (*C function*), 3753
arch_gdb_init (*C function*), 3753
arch_gdb_reg_readall (*C function*), 3753
arch_gdb_reg_readone (*C function*), 3753
arch_gdb_reg_writeall (*C function*), 3753
arch_gdb_reg_writeone (*C function*), 3753
arch_gdb_remove_breakpoint (*C function*), 3754
arch_gdb_step (*C function*), 3753
arch_irq_allocate (*C function*), 3741
arch_irq_connect_dynamic (*C function*), 3740
arch_irq_disable (*C function*), 3740
arch_irq_disconnect_dynamic (*C function*), 3741
arch_irq_enable (*C function*), 3740
arch_irq_is_enabled (*C function*), 3740
arch_irq_is_used (*C function*), 3741
arch_irq_lock (*C function*), 3739
arch_irq_set_used (*C function*), 3741
arch_irq_unlock (*C function*), 3739
arch_irq_unlocked (*C function*), 3739
arch_is_in_isr (*C function*), 3739
arch_is_user_context (*C function*), 3744
arch_k_cycle_get_32 (*C function*), 3733
arch_k_cycle_get_64 (*C function*), 3734
arch_kernel_init (*C function*), 3752
arch_mem_coherent (*C function*), 3746
arch_mem_domain_max_partitions_get (*C function*), 3744

arch_mem_map (*C function*), 3748
arch_mem_page_in (*C function*), 3750
arch_mem_page_out (*C function*), 3750
arch_mem_scratch (*C function*), 3750
arch_mem_unmap (*C function*), 3749
arch_new_thread (*C function*), 3734
arch_nop (*C function*), 3752
arch_num_cpus (*C function*), 3739
arch_page_info_get (*C function*), 3751
arch_page_location (*C enum*), 3748
arch_page_location_get (*C function*), 3750
arch_page_location.ARCH_PAGE_LOCATION_BAD (*C enumerator*), 3748
arch_page_location.ARCH_PAGE_LOCATION_PAGED_IN (*C enumerator*), 3748
arch_page_location.ARCH_PAGE_LOCATION_PAGED_OUT (*C enumerator*), 3748
arch_page_phys_get (*C function*), 3749
arch_printk_char_out (*C function*), 3752
arch_proc_id (*C function*), 3738
arch_reserved_pages_update (*C function*), 3750
arch_sched_broadcast_ipi (*C function*), 3739
arch_sched_directed_ipi (*C function*), 3739
arch_smp_init (*C function*), 3739
arch_switch (*C function*), 3734
arch_switch_to_main_thread (*C function*), 3735
arch_syscall_invoke0 (*C function*), 3742
arch_syscall_invoke1 (*C function*), 3742
arch_syscall_invoke2 (*C function*), 3742
arch_syscall_invoke3 (*C function*), 3743
arch_syscall_invoke4 (*C function*), 3743
arch_syscall_invoke5 (*C function*), 3743
arch_syscall_invoke6 (*C function*), 3744
arch_syscall_oops (*C function*), 3746
arch_system_halt (*C function*), 3737
arch_timing_counter_get (*C function*), 654
arch_timing_cycles_get (*C function*), 654
arch_timing_cycles_to_ns (*C function*), 655
arch_timing_cycles_to_ns_avg (*C function*), 655
arch_timing_freq_get (*C function*), 655
arch_timing_freq_get_mhz (*C function*), 656
arch_timing_init (*C function*), 653
arch_timing_start (*C function*), 653
arch_timing_stop (*C function*), 653
arch_tls_stack_setup (*C function*), 3736
arch_user_mode_enter (*C function*), 3745
arch_user_string_nlen (*C function*), 3746
arch_virt_region_align (*C function*), 3745
architecture, **3935**
ARCMWDT_TOOLCHAIN_PATH, 290
arithmetic_shift_right (*C function*), 696
ARM_PRODUCT_DEF, 288
ARMCLANG_TOOLCHAIN_PATH, 288
ARMLMD_LICENSE_FILE, 288
ARRAY_FOR_EACH (*C macro*), 681
ARRAY_FOR_EACH_PTR (*C macro*), 681
ARRAY_INDEX (*C macro*), 680
ARRAY_INDEX_FLOOR (*C macro*), 681
ARRAY_SIZE (*C macro*), 680
atomic_add (*C function*), 495
atomic_and (*C function*), 499

ATOMIC_BITMAP_SIZE (*C macro*), 492
atomic_cas (*C function*), 494
atomic_clear (*C function*), 498
atomic_clear_bit (*C function*), 494
atomic_dec (*C function*), 496
ATOMIC_DEFINE (*C macro*), 492
atomic_get (*C function*), 496
atomic_inc (*C function*), 496
ATOMIC_INIT (*C macro*), 492
atomic_nand (*C function*), 499
atomic_or (*C function*), 498
atomic_ptr_cas (*C function*), 495
atomic_ptr_clear (*C function*), 498
atomic_ptr_get (*C function*), 497
ATOMIC_PTR_INIT (*C macro*), 492
atomic_ptr_set (*C function*), 497
atomic_set (*C function*), 497
atomic_set_bit (*C function*), 494
atomic_set_bit_to (*C function*), 494
atomic_sub (*C function*), 496
atomic_test_and_clear_bit (*C function*), 493
atomic_test_and_set_bit (*C function*), 493
atomic_test_bit (*C function*), 493
atomic_xor (*C function*), 499
audio_channel_t (*C enum*), 3183
audio_channel_t.AUDIO_CHANNEL_ALL (*C enumerator*), 3183
audio_channel_t.AUDIO_CHANNEL_FRONT_CENTER (*C enumerator*), 3183
audio_channel_t.AUDIO_CHANNEL_FRONT_LEFT (*C enumerator*), 3183
audio_channel_t.AUDIO_CHANNEL_FRONT_RIGHT (*C enumerator*), 3183
audio_channel_t.AUDIO_CHANNEL_LFE (*C enumerator*), 3183
audio_channel_t.AUDIO_CHANNEL_REAR_CENTER (*C enumerator*), 3183
audio_channel_t.AUDIO_CHANNEL_REAR_LEFT (*C enumerator*), 3183
audio_channel_t.AUDIO_CHANNEL_REAR_RIGHT (*C enumerator*), 3183
audio_channel_t.AUDIO_CHANNEL_SIDE_LEFT (*C enumerator*), 3183
audio_channel_t.AUDIO_CHANNEL_SIDE_RIGHT (*C enumerator*), 3183
audio_codec_apply_properties (*C function*), 3185
audio_codec_cfg (*C struct*), 3185
audio_codec_cfg.dai_cfg (*C var*), 3186
audio_codec_cfg.dai_type (*C var*), 3186
audio_codec_cfg.mclk_freq (*C var*), 3186
audio_codec_clear_errors (*C function*), 3185
audio_codec_configure (*C function*), 3184
audio_codec_error_callback_t (*C type*), 3181
audio_codec_error_type (*C enum*), 3183
audio_codec_error_type.AUDIO_CODEC_ERROR_DC (*C enumerator*), 3184
audio_codec_error_type.AUDIO_CODEC_ERROR_OVERCURRENT (*C enumerator*), 3183
audio_codec_error_type.AUDIO_CODEC_ERROR_OVERTEMPERATURE (*C enumerator*), 3183
audio_codec_error_type.AUDIO_CODEC_ERROR_OVERVOLTAGE (*C enumerator*), 3184
audio_codec_error_type.AUDIO_CODEC_ERROR_UNDERVOLTAGE (*C enumerator*), 3184
audio_codec_register_error_callback (*C function*), 3185
audio_codec_set_property (*C function*), 3184
audio_codec_start_output (*C function*), 3184
audio_codec_stop_output (*C function*), 3184
audio_dai_cfg_t (*C union*), 3185
audio_dai_cfg_t.i2s (*C var*), 3185
audio_dai_type_t (*C enum*), 3182
audio_dai_type_t.AUDIO_DAI_TYPE_I2S (*C enumerator*), 3182
audio_dai_type_t.AUDIO_DAI_TYPE_INVALID (*C enumerator*), 3182

audio_pcm_rate_t (*C enum*), 3181
audio_pcm_rate_t.AUDIO_PCM_RATE_8K (*C enumerator*), 3181
audio_pcm_rate_t.AUDIO_PCM_RATE_16K (*C enumerator*), 3181
audio_pcm_rate_t.AUDIO_PCM_RATE_24K (*C enumerator*), 3181
audio_pcm_rate_t.AUDIO_PCM_RATE_32K (*C enumerator*), 3182
audio_pcm_rate_t.AUDIO_PCM_RATE_44P1K (*C enumerator*), 3182
audio_pcm_rate_t.AUDIO_PCM_RATE_48K (*C enumerator*), 3182
audio_pcm_rate_t.AUDIO_PCM_RATE_96K (*C enumerator*), 3182
audio_pcm_rate_t.AUDIO_PCM_RATE_192K (*C enumerator*), 3182
audio_pcm_width_t (*C enum*), 3182
audio_pcm_width_t.AUDIO_PCM_WIDTH_16_BITS (*C enumerator*), 3182
audio_pcm_width_t.AUDIO_PCM_WIDTH_20_BITS (*C enumerator*), 3182
audio_pcm_width_t.AUDIO_PCM_WIDTH_24_BITS (*C enumerator*), 3182
audio_pcm_width_t.AUDIO_PCM_WIDTH_32_BITS (*C enumerator*), 3182
audio_property_t (*C enum*), 3182
audio_property_t.AUDIO_PROPERTY_OUTPUT_MUTE (*C enumerator*), 3183
audio_property_t.AUDIO_PROPERTY_OUTPUT_VOLUME (*C enumerator*), 3182
audio_property_value_t (*C union*), 3186
audio_property_value_t.mute (*C var*), 3186
audio_property_value_t.vol (*C var*), 3186
auxdisplay_backlight_get (*C function*), 3177
auxdisplay_backlight_set (*C function*), 3177
auxdisplay_brightness_get (*C function*), 3177
auxdisplay_brightness_set (*C function*), 3177
auxdisplay_capabilities (*C struct*), 3179
auxdisplay_capabilities_get (*C function*), 3176
auxdisplay_capabilities.backlight (*C var*), 3180
auxdisplay_capabilities.brightness (*C var*), 3180
auxdisplay_capabilities.columns (*C var*), 3179
auxdisplay_capabilities.custom_character_height (*C var*), 3180
auxdisplay_capabilities.custom_character_width (*C var*), 3180
auxdisplay_capabilities.custom_characters (*C var*), 3180
auxdisplay_capabilities.mode (*C var*), 3179
auxdisplay_capabilities.rows (*C var*), 3179
auxdisplay_character (*C struct*), 3180
auxdisplay_character.character_code (*C var*), 3180
auxdisplay_character.data (*C var*), 3180
auxdisplay_character.index (*C var*), 3180
auxdisplay_clear (*C function*), 3176
auxdisplay_cursor_position_get (*C function*), 3175
auxdisplay_cursor_position_set (*C function*), 3175
auxdisplay_cursor_set_enabled (*C function*), 3174
auxdisplay_cursor_shift_set (*C function*), 3175
auxdisplay_custom_character_set (*C function*), 3178
auxdisplay_custom_command (*C function*), 3179
auxdisplay_custom_data (*C struct*), 3180
auxdisplay_custom_data.data (*C var*), 3180
auxdisplay_custom_data.len (*C var*), 3180
auxdisplay_custom_data.options (*C var*), 3180
auxdisplay_direction (*C enum*), 3173
auxdisplay_direction.AUXDISPLAY_DIRECTION_COUNT (*C enumerator*), 3174
auxdisplay_direction.AUXDISPLAY_DIRECTION_LEFT (*C enumerator*), 3174
auxdisplay_direction.AUXDISPLAY_DIRECTION_RIGHT (*C enumerator*), 3174
auxdisplay_display_off (*C function*), 3174
auxdisplay_display_on (*C function*), 3174
auxdisplay_display_position_get (*C function*), 3176
auxdisplay_display_position_set (*C function*), 3176
auxdisplay_is_busy (*C function*), 3178

auxdisplay_light (*C struct*), 3179
AUXDISPLAY_LIGHT_NOT_SUPPORTED (*C macro*), 3173
auxdisplay_light.maximum (*C var*), 3179
auxdisplay_light.minimum (*C var*), 3179
auxdisplay_mode_t (*C type*), 3173
auxdisplay_position (*C enum*), 3173
auxdisplay_position_blinking_set_enabled (*C function*), 3174
auxdisplay_position.AUXDISPLAY_POSITION_ABSOLUTE (*C enumerator*), 3173
auxdisplay_position.AUXDISPLAY_POSITION_COUNT (*C enumerator*), 3173
auxdisplay_position.AUXDISPLAY_POSITION_RELATIVE (*C enumerator*), 3173
auxdisplay_position.AUXDISPLAY_POSITION_RELATIVE_DIRECTION (*C enumerator*), 3173
auxdisplay_write (*C function*), 3178

B

barrier_dmem_fence_full (*C function*), 3130
barrier_dsync_fence_full (*C function*), 3130
barrier_isync_fence_full (*C function*), 3131
bbram_api_check_invalid_t (*C type*), 3212
bbram_api_check_power_t (*C type*), 3212
bbram_api_check_standby_power_t (*C type*), 3212
bbram_api_get_size_t (*C type*), 3212
bbram_api_read_t (*C type*), 3212
bbram_api_write_t (*C type*), 3212
bbram_check_invalid (*C function*), 3213
bbram_check_power (*C function*), 3213
bbram_check_standby_power (*C function*), 3213
bbram_driver_api (*C struct*), 3215
bbram_emul_set_invalid (*C function*), 3214
bbram_emul_set_power_state (*C function*), 3214
bbram_emul_set_standby_power_state (*C function*), 3214
bbram_get_size (*C function*), 3213
bbram_read (*C function*), 3213
bbram_write (*C function*), 3214
bc12_callback_t (*C type*), 3216
BC12_CHARGER_MAX_CURR_UA (*C macro*), 3216
BC12_CHARGER_MIN_CURR_UA (*C macro*), 3216
BC12_CHARGER_VOLTAGE_UV (*C macro*), 3216
bc12_emul_set_charging_partner (*C function*), 3218
bc12_emul_set_pd_partner (*C function*), 3219
bc12_partner_state (*C struct*), 3218
bc12_role (*C enum*), 3217
bc12_role.BC12_CHARGING_PORT (*C enumerator*), 3217
bc12_role.BC12_DISCONNECTED (*C enumerator*), 3217
bc12_role.BC12_PORTABLE_DEVICE (*C enumerator*), 3217
bc12_set_result_cb (*C function*), 3218
bc12_set_role (*C function*), 3217
bc12_type (*C enum*), 3217
bc12_type.BC12_TYPE_CDP (*C enumerator*), 3217
bc12_type.BC12_TYPE_COUNT (*C enumerator*), 3217
bc12_type.BC12_TYPE_DCP (*C enumerator*), 3217
bc12_type.BC12_TYPE_NONE (*C enumerator*), 3217
bc12_type.BC12_TYPE_PROPRIETARY (*C enumerator*), 3217
bc12_type.BC12_TYPE_SDP (*C enumerator*), 3217
bc12_type.BC12_TYPE_UNKNOWN (*C enumerator*), 3217
bcd2bin (*C function*), 697
bin2bcd (*C function*), 697
bin2hex (*C function*), 697
bin_file (*runners.core.RunnerConfig attribute*), 197

bind (*C function*), 2488
BINDESC_ID_APP_VERSION_MAJOR (*C macro*), 711
BINDESC_ID_APP_VERSION_MINOR (*C macro*), 711
BINDESC_ID_APP_VERSION_NUMBER (*C macro*), 711
BINDESC_ID_APP_VERSION_PATCHLEVEL (*C macro*), 711
BINDESC_ID_APP_VERSION_STRING (*C macro*), 711
BINDESC_ID_BUILD_DATE_STRING (*C macro*), 712
BINDESC_ID_BUILD_DATE_TIME_STRING (*C macro*), 712
BINDESC_ID_BUILD_TIME_DAY (*C macro*), 712
BINDESC_ID_BUILD_TIME_HOUR (*C macro*), 712
BINDESC_ID_BUILD_TIME_MINUTE (*C macro*), 712
BINDESC_ID_BUILD_TIME_MONTH (*C macro*), 712
BINDESC_ID_BUILD_TIME_SECOND (*C macro*), 712
BINDESC_ID_BUILD_TIME_STRING (*C macro*), 712
BINDESC_ID_BUILD_TIME_UNIX (*C macro*), 712
BINDESC_ID_BUILD_TIME_YEAR (*C macro*), 711
BINDESC_ID_C_COMPILER_NAME (*C macro*), 712
BINDESC_ID_C_COMPILER_VERSION (*C macro*), 712
BINDESC_ID_CXX_COMPILER_NAME (*C macro*), 712
BINDESC_ID_CXX_COMPILER_VERSION (*C macro*), 712
BINDESC_ID_HOST_NAME (*C macro*), 712
BINDESC_ID_KERNEL_VERSION_MAJOR (*C macro*), 711
BINDESC_ID_KERNEL_VERSION_MINOR (*C macro*), 711
BINDESC_ID_KERNEL_VERSION_NUMBER (*C macro*), 711
BINDESC_ID_KERNEL_VERSION_PATCHLEVEL (*C macro*), 711
BINDESC_ID_KERNEL_VERSION_STRING (*C macro*), 711
BINDESC_TAG_DESCRIPTORS_END (*C macro*), 712
BIT (*C macro*), 686
BIT64 (*C macro*), 686
BIT64_MASK (*C macro*), 686
BIT_MASK (*C macro*), 686
BITS_PER_LONG (*C macro*), 679
BITS_PER_LONG_LONG (*C macro*), 679
blinfo_lookup (*C function*), 1239
block_op_t (*C type*), 721
BOARD, 186
board, **3935**
board configuration, **3935**
board name, **3935**
board qualifiers, **3935**
board revision, **3936**
board target, **3936**
board_dir (*runners.core.RunnerConfig attribute*), 197
board_timing_counter_get (*C function*), 659
board_timing_cycles_get (*C function*), 660
board_timing_cycles_to_ns (*C function*), 660
board_timing_cycles_to_ns_avg (*C function*), 661
board_timing_freq_get (*C function*), 660
board_timing_freq_get_mhz (*C function*), 661
board_timing_init (*C function*), 659
board_timing_start (*C function*), 659
board_timing_stop (*C function*), 659
boot_erase_img_bank (*C function*), 823
boot_get_area_trailer_status_offset (*C function*), 824
boot_get_trailer_status_offset (*C function*), 824
BOOT_IMG_VER_STRLEN_MAX (*C macro*), 821
boot_is_img_confirmed (*C function*), 822
BOOT_MODE_TYPES (*C enum*), 1242

BOOT_MODE_TYPES.BOOT_MODE_TYPE_BOOTLOADER (*C enumerator*), 1242
BOOT_MODE_TYPES.BOOT_MODE_TYPE_NORMAL (*C enumerator*), 1242
boot_read_bank_header (*C function*), 822
boot_request_upgrade (*C function*), 823
boot_request_upgrade_multi (*C function*), 823
BOOT_SWAP_TYPE_FAIL (*C macro*), 821
BOOT_SWAP_TYPE_NONE (*C macro*), 821
BOOT_SWAP_TYPE_PERM (*C macro*), 821
BOOT_SWAP_TYPE_REVERT (*C macro*), 821
BOOT_SWAP_TYPE_TEST (*C macro*), 821
BOOT_UPGRADE_PERMANENT (*C macro*), 822
BOOT_UPGRADE_TEST (*C macro*), 822
boot_write_img_confirmed (*C function*), 822
boot_write_img_confirmed_multi (*C function*), 823
bootmode_check (*C function*), 1242
bootmode_clear (*C function*), 1242
bootmode_set (*C function*), 1242
BT_ADDR_ANY (*C macro*), 2001
bt_addr_any (*C var*), 2004
bt_addr_cmp (*C function*), 2002
bt_addr_copy (*C function*), 2003
bt_addr_eq (*C function*), 2002
bt_addr_from_str (*C function*), 2004
BT_ADDR_IS_NRPA (*C macro*), 2001
BT_ADDR_IS_RPA (*C macro*), 2001
BT_ADDR_IS_STATIC (*C macro*), 2001
BT_ADDR_LE_ANONYMOUS (*C macro*), 2001
BT_ADDR_LE_ANY (*C macro*), 2001
bt_addr_le_any (*C var*), 2004
bt_addr_le_cmp (*C function*), 2002
bt_addr_le_copy (*C function*), 2003
bt_addr_le_create_nrpa (*C function*), 2003
bt_addr_le_create_static (*C function*), 2003
bt_addr_le_eq (*C function*), 2002
bt_addr_le_from_str (*C function*), 2004
bt_addr_le_is_identity (*C function*), 2003
bt_addr_le_is_rpa (*C function*), 2003
BT_ADDR_LE_NONE (*C macro*), 2001
bt_addr_le_none (*C var*), 2004
BT_ADDR_LE_PUBLIC (*C macro*), 2001
BT_ADDR_LE_PUBLIC_ID (*C macro*), 2001
BT_ADDR_LE_RANDOM (*C macro*), 2001
BT_ADDR_LE_RANDOM_ID (*C macro*), 2001
BT_ADDR_LE_SIZE (*C macro*), 2001
BT_ADDR_LE_STR_LEN (*C macro*), 2002
bt_addr_le_t (*C struct*), 2004
bt_addr_le_to_str (*C function*), 2003
BT_ADDR_LE_UNRESOLVED (*C macro*), 2001
BT_ADDR_NONE (*C macro*), 2001
bt_addr_none (*C var*), 2004
BT_ADDR_SET_NRPA (*C macro*), 2001
BT_ADDR_SET_RPA (*C macro*), 2001
BT_ADDR_SET_STATIC (*C macro*), 2001
BT_ADDR_SIZE (*C macro*), 2001
BT_ADDR_STR_LEN (*C macro*), 2002
bt_addr_t (*C struct*), 2004
bt_addr_to_str (*C function*), 2003
BT_APPEARANCE_AIRCRAFT_LARGE_PASSENGER (*C macro*), 2024

BT_APPEARANCE_AIRCRAFT_LIGHT (*C macro*), 2024
BT_APPEARANCE_AIRCRAFT_MICROLIGHT (*C macro*), 2024
BT_APPEARANCE_AIRCRAFT_PARAGLIDER (*C macro*), 2024
BT_APPEARANCE_APPLIANCE_CLOTHES_IRON (*C macro*), 2023
BT_APPEARANCE_APPLIANCE_CLOTHES_STEAMER (*C macro*), 2023
BT_APPEARANCE_APPLIANCE_COFFEE_MAKER (*C macro*), 2023
BT_APPEARANCE_APPLIANCE_CURLING_IRON (*C macro*), 2023
BT_APPEARANCE_APPLIANCE_DRYER (*C macro*), 2023
BT_APPEARANCE_APPLIANCE_FREEZER (*C macro*), 2022
BT_APPEARANCE_APPLIANCE_HAIR_DRYER (*C macro*), 2023
BT_APPEARANCE_APPLIANCE_MICROWAVE (*C macro*), 2022
BT_APPEARANCE_APPLIANCE_OVEN (*C macro*), 2022
BT_APPEARANCE_APPLIANCE_REFRIGERATOR (*C macro*), 2022
BT_APPEARANCE_APPLIANCE_RICE_COOKER (*C macro*), 2023
BT_APPEARANCE_APPLIANCE_ROBOTIC_VACUUM_CLEANER (*C macro*), 2023
BT_APPEARANCE_APPLIANCE_TOASTER (*C macro*), 2022
BT_APPEARANCE_APPLIANCE_VACUUM_CLEANER (*C macro*), 2023
BT_APPEARANCE_APPLIANCE_WASHING_MACHINE (*C macro*), 2023
BT_APPEARANCE_AUDIO_SINK_BOOKSHELF_SPEAKER (*C macro*), 2020
BT_APPEARANCE_AUDIO_SINK_SOUNDBAR (*C macro*), 2020
BT_APPEARANCE_AUDIO_SINK_SPEAKERPHONE (*C macro*), 2020
BT_APPEARANCE_AUDIO_SINK_STANDALONE_SPEAKER (*C macro*), 2020
BT_APPEARANCE_AUDIO_SINK_STANDMOUNTED_SPEAKER (*C macro*), 2020
BT_APPEARANCE_AUDIO_SOURCE_ALARM (*C macro*), 2021
BT_APPEARANCE_AUDIO_SOURCE_AUDITORIUM (*C macro*), 2021
BT_APPEARANCE_AUDIO_SOURCE_BELL (*C macro*), 2021
BT_APPEARANCE_AUDIO_SOURCE_BROADCASTING_DEVICE (*C macro*), 2021
BT_APPEARANCE_AUDIO_SOURCE_BROADCASTING_ROOM (*C macro*), 2021
BT_APPEARANCE_AUDIO_SOURCE_HORN (*C macro*), 2021
BT_APPEARANCE_AUDIO_SOURCE_KIOSK (*C macro*), 2021
BT_APPEARANCE_AUDIO_SOURCE_MICROPHONE (*C macro*), 2021
BT_APPEARANCE_AUDIO_SOURCE_SERVICE_DESK (*C macro*), 2021
BT_APPEARANCE_AV_EQUIPMENT_AMPLIFIER (*C macro*), 2024
BT_APPEARANCE_AV_EQUIPMENT_BLURAY_PLAYER (*C macro*), 2024
BT_APPEARANCE_AV_EQUIPMENT_CD_PLAYER (*C macro*), 2024
BT_APPEARANCE_AV_EQUIPMENT_DVD_PLAYER (*C macro*), 2024
BT_APPEARANCE_AV_EQUIPMENT_OPTICAL_DISC_PLAYER (*C macro*), 2024
BT_APPEARANCE_AV_EQUIPMENT_RADIO (*C macro*), 2024
BT_APPEARANCE_AV_EQUIPMENT_RECEIVER (*C macro*), 2024
BT_APPEARANCE_AV_EQUIPMENT_SET_TOP_BOX (*C macro*), 2024
BT_APPEARANCE_AV_EQUIPMENT_TUNER (*C macro*), 2024
BT_APPEARANCE_AV_EQUIPMENT_TURNTABLE (*C macro*), 2024
BT_APPEARANCE_BLOOD_PRESSURE_ARM (*C macro*), 2010
BT_APPEARANCE_BLOOD_PRESSURE_WRIST (*C macro*), 2010
BT_APPEARANCE_COMPUTER_ALL_IN_ONE (*C macro*), 2008
BT_APPEARANCE_COMPUTER_BLADE_SERVER (*C macro*), 2009
BT_APPEARANCE_COMPUTER_CONVERTIBLE (*C macro*), 2009
BT_APPEARANCE_COMPUTER_DESKTOP_WORKSTATION (*C macro*), 2008
BT_APPEARANCE_COMPUTER_DETACHABLE (*C macro*), 2009
BT_APPEARANCE_COMPUTER_DOCKING_STATION (*C macro*), 2008
BT_APPEARANCE_COMPUTER_HANDHELD_PCPDA (*C macro*), 2008
BT_APPEARANCE_COMPUTER_IOT_GATEWAY (*C macro*), 2009
BT_APPEARANCE_COMPUTER_LAPTOP (*C macro*), 2008
BT_APPEARANCE_COMPUTER_MINI_PC (*C macro*), 2009
BT_APPEARANCE_COMPUTER_PALMSIZE_PCPDA (*C macro*), 2008
BT_APPEARANCE_COMPUTER_SERVER_CLASS (*C macro*), 2008
BT_APPEARANCE_COMPUTER_STICK_PC (*C macro*), 2009
BT_APPEARANCE_COMPUTER_TABLET (*C macro*), 2008

BT_APPEARANCE_COMPUTER_WEARABLE_COMPUTER (*C macro*), 2008
BT_APPEARANCE_CONTINUOUS_GLUCOSE_MONITOR (*C macro*), 2026
BT_APPEARANCE_CONTROL_ACCESS_DOOR (*C macro*), 2018
BT_APPEARANCE_CONTROL_ACCESS_LOCK (*C macro*), 2018
BT_APPEARANCE_CONTROL_BATTERY_SWITCH (*C macro*), 2012
BT_APPEARANCE_CONTROL_BUTTON (*C macro*), 2012
BT_APPEARANCE_CONTROL_DOOR_LOCK (*C macro*), 2018
BT_APPEARANCE_CONTROL_DOUBLE_SWITCH (*C macro*), 2012
BT_APPEARANCE_CONTROL_ELEVATOR (*C macro*), 2018
BT_APPEARANCE_CONTROL_EMERGENCY_EXIT_DOOR (*C macro*), 2018
BT_APPEARANCE_CONTROL_ENERGY_HARVESTING_SWITCH (*C macro*), 2012
BT_APPEARANCE_CONTROL_ENTRANCE_GATE (*C macro*), 2018
BT_APPEARANCE_CONTROL_GARAGE_DOOR (*C macro*), 2018
BT_APPEARANCE_CONTROL_LOCKER (*C macro*), 2018
BT_APPEARANCE_CONTROL_MULTI_SWITCH (*C macro*), 2012
BT_APPEARANCE_CONTROL_PUSH_BUTTON (*C macro*), 2012
BT_APPEARANCE_CONTROL_ROTARY_SWITCH (*C macro*), 2012
BT_APPEARANCE_CONTROL_SINGLE_SWITCH (*C macro*), 2012
BT_APPEARANCE_CONTROL_SLIDER (*C macro*), 2012
BT_APPEARANCE_CONTROL_SWITCH (*C macro*), 2011
BT_APPEARANCE_CONTROL_TOUCH_PANEL (*C macro*), 2012
BT_APPEARANCE_CONTROL_TRIPLE_SWITCH (*C macro*), 2012
BT_APPEARANCE_CONTROL_WINDOW (*C macro*), 2018
BT_APPEARANCE_CYCLING_CADENCE (*C macro*), 2011
BT_APPEARANCE_CYCLING_COMPUTER (*C macro*), 2011
BT_APPEARANCE_CYCLING_POWER (*C macro*), 2011
BT_APPEARANCE_CYCLING_SPEED (*C macro*), 2011
BT_APPEARANCE_CYCLING_SPEED_CADENCE (*C macro*), 2011
BT_APPEARANCE_DISPLAY_EQUIPMENT_MONITOR (*C macro*), 2025
BT_APPEARANCE_DISPLAY_EQUIPMENT_PROJECTOR (*C macro*), 2025
BT_APPEARANCE_DISPLAY_EQUIPMENT_TELEVISION (*C macro*), 2025
BT_APPEARANCE_FAN_AXIAL (*C macro*), 2016
BT_APPEARANCE_FAN_CEILING (*C macro*), 2016
BT_APPEARANCE_FAN_DESK (*C macro*), 2016
BT_APPEARANCE_FAN_EXHAUST (*C macro*), 2016
BT_APPEARANCE_FAN_PEDESTAL (*C macro*), 2016
BT_APPEARANCE_FAN_WALL (*C macro*), 2016
BT_APPEARANCE_GENERIC_ACCESS_CONTROL (*C macro*), 2018
BT_APPEARANCE_GENERIC_AIR_CONDITIONING (*C macro*), 2017
BT_APPEARANCE_GENERIC_AIRCRAFT (*C macro*), 2023
BT_APPEARANCE_GENERIC_AUDIO_SINK (*C macro*), 2020
BT_APPEARANCE_GENERIC_AUDIO_SOURCE (*C macro*), 2021
BT_APPEARANCE_GENERIC_AV_EQUIPMENT (*C macro*), 2024
BT_APPEARANCE_GENERIC_BARCODE_SCANNER (*C macro*), 2010
BT_APPEARANCE_GENERIC_BLOOD_PRESSURE (*C macro*), 2010
BT_APPEARANCE_GENERIC_CLOCK (*C macro*), 2009
BT_APPEARANCE_GENERIC_COMPUTER (*C macro*), 2008
BT_APPEARANCE_GENERIC_CONTROL_DEVICE (*C macro*), 2011
BT_APPEARANCE_GENERIC_CYCLING (*C macro*), 2011
BT_APPEARANCE_GENERIC_DISPLAY (*C macro*), 2009
BT_APPEARANCE_GENERIC_DISPLAY_EQUIPMENT (*C macro*), 2024
BT_APPEARANCE_GENERIC_DOMESTIC_APPLIANCE (*C macro*), 2022
BT_APPEARANCE_GENERIC_EYEGLASSES (*C macro*), 2009
BT_APPEARANCE_GENERIC_FAN (*C macro*), 2016
BT_APPEARANCE_GENERIC_GAMING (*C macro*), 2025
BT_APPEARANCE_GENERIC_GLUCOSE (*C macro*), 2011
BT_APPEARANCE_GENERIC_HEARING_AID (*C macro*), 2025
BT_APPEARANCE_GENERIC_HEART_RATE (*C macro*), 2010

BT_APPEARANCE_GENERIC_HEATING (*C macro*), 2017
BT_APPEARANCE_GENERIC_HID (*C macro*), 2010
BT_APPEARANCE_GENERIC_HUMIDIFIER (*C macro*), 2017
BT_APPEARANCE_GENERIC_HVAC (*C macro*), 2016
BT_APPEARANCE_GENERIC_INSULIN_PUMP (*C macro*), 2026
BT_APPEARANCE_GENERIC_KEYRING (*C macro*), 2009
BT_APPEARANCE_GENERIC_LIGHT_FIXTURES (*C macro*), 2014
BT_APPEARANCE_GENERIC_LIGHT_SOURCE (*C macro*), 2019
BT_APPEARANCE_GENERIC_MEDIA_PLAYER (*C macro*), 2009
BT_APPEARANCE_GENERIC_MEDICATION_DELIVERY (*C macro*), 2026
BT_APPEARANCE_GENERIC_MOTORIZED_DEVICE (*C macro*), 2018
BT_APPEARANCE_GENERIC_MOTORIZED_VEHICLE (*C macro*), 2021
BT_APPEARANCE_GENERIC_NETWORK_DEVICE (*C macro*), 2012
BT_APPEARANCE_GENERIC_OUTDOOR_SPORTS (*C macro*), 2026
BT_APPEARANCE_GENERIC_PERSONAL_MOBILITY_DEVICE (*C macro*), 2026
BT_APPEARANCE_GENERIC_PHONE (*C macro*), 2008
BT_APPEARANCE_GENERIC_POWER_DEVICE (*C macro*), 2019
BT_APPEARANCE_GENERIC_PULSE_OXIMETER (*C macro*), 2025
BT_APPEARANCE_GENERIC_REMOTE (*C macro*), 2009
BT_APPEARANCE_GENERIC_SENSOR (*C macro*), 2012
BT_APPEARANCE_GENERIC_SIGNAGE (*C macro*), 2025
BT_APPEARANCE_GENERIC_SPIROMETER (*C macro*), 2026
BT_APPEARANCE_GENERIC_TAG (*C macro*), 2009
BT_APPEARANCE_GENERIC_THERMOMETER (*C macro*), 2010
BT_APPEARANCE_GENERIC_WALKING (*C macro*), 2011
BT_APPEARANCE_GENERIC_WATCH (*C macro*), 2009
BT_APPEARANCE_GENERIC_WEARABLE_AUDIO_DEVICE (*C macro*), 2023
BT_APPEARANCE_GENERIC_WEIGHT_SCALE (*C macro*), 2026
BT_APPEARANCE_GENERIC_WINDOW_COVERING (*C macro*), 2020
BT_APPEARANCE_HEARING_AID_BEHIND_EAR (*C macro*), 2025
BT_APPEARANCE_HEARING_AID_COCHLEAR_IMPLANT (*C macro*), 2025
BT_APPEARANCE_HEARING_AID_IN_EAR (*C macro*), 2025
BT_APPEARANCE_HEART_RATE_BELT (*C macro*), 2010
BT_APPEARANCE_HEATING_AIR_CURTAIN (*C macro*), 2017
BT_APPEARANCE_HEATING_BOILER (*C macro*), 2017
BT_APPEARANCE_HEATING_FAN_HEATER (*C macro*), 2017
BT_APPEARANCE_HEATING_HEAT_PUMP (*C macro*), 2017
BT_APPEARANCE_HEATING_INFRARED_HEATER (*C macro*), 2017
BT_APPEARANCE_HEATING_RADIANT_PANEL_HEATER (*C macro*), 2017
BT_APPEARANCE_HEATING_RADIATOR (*C macro*), 2017
BT_APPEARANCE_HID_BARCODE_SCANNER (*C macro*), 2011
BT_APPEARANCE_HID_CARD_READER (*C macro*), 2010
BT_APPEARANCE_HID_DIGITAL_PEN (*C macro*), 2010
BT_APPEARANCE_HID_DIGITIZER_TABLET (*C macro*), 2010
BT_APPEARANCE_HID_GAMEPAD (*C macro*), 2010
BT_APPEARANCE_HID_JOYSTICK (*C macro*), 2010
BT_APPEARANCE_HID_KEYBOARD (*C macro*), 2010
BT_APPEARANCE_HID_MOUSE (*C macro*), 2010
BT_APPEARANCE_HID_PRESENTATION_REMOTE (*C macro*), 2011
BT_APPEARANCE_HID_TOUCHPAD (*C macro*), 2011
BT_APPEARANCE_HOME_VIDEO_GAME_CONSOLE (*C macro*), 2025
BT_APPEARANCE_HVAC_AIR_CURTAIN (*C macro*), 2017
BT_APPEARANCE_HVAC_BOILER (*C macro*), 2017
BT_APPEARANCE_HVAC_DEHUMIDIFIER (*C macro*), 2016
BT_APPEARANCE_HVAC_FAN_HEATER (*C macro*), 2017
BT_APPEARANCE_HVAC_HEAT_PUMP (*C macro*), 2017
BT_APPEARANCE_HVAC_HEATER (*C macro*), 2016
BT_APPEARANCE_HVAC_HUMIDIFIER (*C macro*), 2016

BT_APPEARANCE_HVAC_INFRARED_HEATER (*C macro*), 2017
BT_APPEARANCE_HVAC_RADIANT_PANEL_HEATER (*C macro*), 2017
BT_APPEARANCE_HVAC_RADIATOR (*C macro*), 2016
BT_APPEARANCE_HVAC_THERMOSTAT (*C macro*), 2016
BT_APPEARANCE_INSULIN_PEN (*C macro*), 2026
BT_APPEARANCE_INSULIN_PUMP_DURABLE (*C macro*), 2026
BT_APPEARANCE_INSULIN_PUMP_PATCH (*C macro*), 2026
BT_APPEARANCE_LIGHT_FIXTURES_BAY (*C macro*), 2015
BT_APPEARANCE_LIGHT_FIXTURES_BOLLARD_WITH (*C macro*), 2015
BT_APPEARANCE_LIGHT_FIXTURES_BULB (*C macro*), 2016
BT_APPEARANCE_LIGHT_FIXTURES_CABINET (*C macro*), 2014
BT_APPEARANCE_LIGHT_FIXTURES_CEILING (*C macro*), 2014
BT_APPEARANCE_LIGHT_FIXTURES_CONTROLLER (*C macro*), 2015
BT_APPEARANCE_LIGHT_FIXTURES_DESK (*C macro*), 2014
BT_APPEARANCE_LIGHT_FIXTURES_DRIVER (*C macro*), 2015
BT_APPEARANCE_LIGHT_FIXTURES_EMERGENCY_EXIT (*C macro*), 2015
BT_APPEARANCE_LIGHT_FIXTURES_FLOOD (*C macro*), 2015
BT_APPEARANCE_LIGHT_FIXTURES_FLOOR (*C macro*), 2014
BT_APPEARANCE_LIGHT_FIXTURES_GARDEN (*C macro*), 2015
BT_APPEARANCE_LIGHT_FIXTURES_HIGH_BAY (*C macro*), 2016
BT_APPEARANCE_LIGHT_FIXTURES_IN_GROUND (*C macro*), 2015
BT_APPEARANCE_LIGHT_FIXTURES_LINEAR (*C macro*), 2015
BT_APPEARANCE_LIGHT_FIXTURES_LOW_BAY (*C macro*), 2016
BT_APPEARANCE_LIGHT_FIXTURES_PATHWAY (*C macro*), 2015
BT_APPEARANCE_LIGHT_FIXTURES_PENDANT (*C macro*), 2015
BT_APPEARANCE_LIGHT_FIXTURES_POLE_TOP (*C macro*), 2015
BT_APPEARANCE_LIGHT_FIXTURES_SHELVES (*C macro*), 2015
BT_APPEARANCE_LIGHT_FIXTURES_STREET (*C macro*), 2015
BT_APPEARANCE_LIGHT_FIXTURES_TROFFER (*C macro*), 2014
BT_APPEARANCE_LIGHT_FIXTURES_UNDERWATER (*C macro*), 2015
BT_APPEARANCE_LIGHT_FIXTURES_WALL (*C macro*), 2014
BT_APPEARANCE_LIGHT_SOURCE_FLUORESCENT_LAMP (*C macro*), 2019
BT_APPEARANCE_LIGHT_SOURCE_HID_LAMP (*C macro*), 2019
BT_APPEARANCE_LIGHT_SOURCE_INCANDESCENT_BULB (*C macro*), 2019
BT_APPEARANCE_LIGHT_SOURCE_LED_ARRAY (*C macro*), 2019
BT_APPEARANCE_LIGHT_SOURCE_LED_LAMP (*C macro*), 2019
BT_APPEARANCE_LIGHT_SOURCE_LOW_VOLTAGE_HALOGEN (*C macro*), 2020
BT_APPEARANCE_LIGHT_SOURCE_MULTICOLOR_LED_ARRAY (*C macro*), 2020
BT_APPEARANCE_LIGHT_SOURCE_OLED (*C macro*), 2020
BT_APPEARANCE_MOBILITY_POWERED_WHEELCHAIR (*C macro*), 2026
BT_APPEARANCE_MOBILITY_SCOOTER (*C macro*), 2026
BT_APPEARANCE_MOTORIZED_AWNING (*C macro*), 2018
BT_APPEARANCE_MOTORIZED_BLINDS_OR_SHADES (*C macro*), 2018
BT_APPEARANCE_MOTORIZED_CURTAINS (*C macro*), 2018
BT_APPEARANCE_MOTORIZED_GATE (*C macro*), 2018
BT_APPEARANCE_MOTORIZED_SCREEN (*C macro*), 2018
BT_APPEARANCE_MULTISENSOR (*C macro*), 2014
BT_APPEARANCE_NETWORK_ACCESS_POINT (*C macro*), 2012
BT_APPEARANCE_NETWORK_MESH_DEVICE (*C macro*), 2012
BT_APPEARANCE_NETWORK_MESH_PROXY (*C macro*), 2012
BT_APPEARANCE_OUTDOOR_SPORTS_LOCATION (*C macro*), 2026
BT_APPEARANCE_OUTDOOR_SPORTS_LOCATION_AND_NAV (*C macro*), 2026
BT_APPEARANCE_OUTDOOR_SPORTS_LOCATION_POD (*C macro*), 2026
BT_APPEARANCE_OUTDOOR_SPORTS_LOCATION_POD_AND_NAV (*C macro*), 2027
BT_APPEARANCE_PORTABLE_HANDHELD_CONSOLE (*C macro*), 2025
BT_APPEARANCE_POWER_CHARGE_CASE (*C macro*), 2019
BT_APPEARANCE_POWER_FLUORESCENT_LAMP_GEAR (*C macro*), 2019
BT_APPEARANCE_POWER_HID_LAMP_GEAR (*C macro*), 2019

BT_APPEARANCE_POWER_LED_DRIVER (*C macro*), 2019
BT_APPEARANCE_POWER_OUTLET (*C macro*), 2019
BT_APPEARANCE_POWER_PLUG (*C macro*), 2019
BT_APPEARANCE_POWER_POWER_BANK (*C macro*), 2019
BT_APPEARANCE_POWER_STRIP (*C macro*), 2019
BT_APPEARANCE_POWER_SUPPLY (*C macro*), 2019
BT_APPEARANCE_PULSE_OXIMETER_FINGERTIP (*C macro*), 2025
BT_APPEARANCE_PULSE_OXIMETER_WRIST (*C macro*), 2025
BT_APPEARANCE_SENSOR_AIR_QUALITY (*C macro*), 2013
BT_APPEARANCE_SENSOR_AMBIENT_LIGHT (*C macro*), 2013
BT_APPEARANCE_SENSOR_CARBON_DIOXIDE (*C macro*), 2013
BT_APPEARANCE_SENSOR_CARBON_MONOXIDE (*C macro*), 2013
BT_APPEARANCE_SENSOR_CEILING_MOUNTED (*C macro*), 2014
BT_APPEARANCE_SENSOR_COLOR_LIGHT (*C macro*), 2013
BT_APPEARANCE_SENSOR_CONTACT (*C macro*), 2013
BT_APPEARANCE_SENSOR_ENERGY (*C macro*), 2013
BT_APPEARANCE_SENSOR_ENERGY_METER (*C macro*), 2014
BT_APPEARANCE_SENSOR_FIRE (*C macro*), 2013
BT_APPEARANCE_SENSOR_FLAME_DETECTOR (*C macro*), 2014
BT_APPEARANCE_SENSOR_FLUSH_MOUNTED (*C macro*), 2014
BT_APPEARANCE_SENSOR_HUMIDITY (*C macro*), 2013
BT_APPEARANCE_SENSOR_LEAK (*C macro*), 2013
BT_APPEARANCE_SENSOR_MOTION (*C macro*), 2013
BT_APPEARANCE_SENSOR_MULTI (*C macro*), 2014
BT_APPEARANCE_SENSOR_OCCUPANCY (*C macro*), 2013
BT_APPEARANCE_SENSOR_PROXIMITY (*C macro*), 2014
BT_APPEARANCE_SENSOR_RAIN (*C macro*), 2013
BT_APPEARANCE_SENSOR_SMOKE (*C macro*), 2013
BT_APPEARANCE_SENSOR_TEMPERATURE (*C macro*), 2013
BT_APPEARANCE_SENSOR_VEHICLE_TIRE_PRESSURE (*C macro*), 2014
BT_APPEARANCE_SENSOR_WALL_MOUNTED (*C macro*), 2014
BT_APPEARANCE_SENSOR_WIND (*C macro*), 2013
BT_APPEARANCE_SIGNAGE_DIGITAL (*C macro*), 2025
BT_APPEARANCE_SIGNAGE_ELECTRONIC_LABEL (*C macro*), 2025
BT_APPEARANCE_SMARTWATCH (*C macro*), 2009
BT_APPEARANCE_SPIROMETER_HANDHELD (*C macro*), 2026
BT_APPEARANCE_SPORTS_WATCH (*C macro*), 2009
BT_APPEARANCE_SPOT_LIGHT (*C macro*), 2015
BT_APPEARANCE_THERMOMETER_EAR (*C macro*), 2010
BT_APPEARANCE_UNKNOWN (*C macro*), 2008
BT_APPEARANCE_VEHICLE_AGRICULTURAL (*C macro*), 2022
BT_APPEARANCE_VEHICLE_BUS (*C macro*), 2022
BT_APPEARANCE_VEHICLE_CAMPER_OR_CARAVAN (*C macro*), 2022
BT_APPEARANCE_VEHICLE_CAR (*C macro*), 2021
BT_APPEARANCE_VEHICLE_LARGE_GOODS (*C macro*), 2021
BT_APPEARANCE_VEHICLE_LIGHT (*C macro*), 2022
BT_APPEARANCE_VEHICLE_MINIBUS (*C macro*), 2022
BT_APPEARANCE_VEHICLE_MOPED (*C macro*), 2022
BT_APPEARANCE_VEHICLE_MOTORBIKE (*C macro*), 2021
BT_APPEARANCE_VEHICLE_QUAD_BIKE (*C macro*), 2022
BT_APPEARANCE_VEHICLE_RECREATIONAL (*C macro*), 2022
BT_APPEARANCE_VEHICLE_SCOOTER (*C macro*), 2021
BT_APPEARANCE_VEHICLE_THREE_WHEELED (*C macro*), 2022
BT_APPEARANCE_VEHICLE_TROLLEY (*C macro*), 2022
BT_APPEARANCE_VEHICLE_TWO_WHEELED (*C macro*), 2021
BT_APPEARANCE_WALKING_IN_SHOE (*C macro*), 2011
BT_APPEARANCE_WALKING_ON_HIP (*C macro*), 2011
BT_APPEARANCE_WALKING_ON_SHOE (*C macro*), 2011

BT_APPEARANCE_WEARABLE_AUDIO_DEVICE_EARBUD (*C macro*), 2023
BT_APPEARANCE_WEARABLE_AUDIO_DEVICE_HEADPHONES (*C macro*), 2023
BT_APPEARANCE_WEARABLE_AUDIO_DEVICE_HEADSET (*C macro*), 2023
BT_APPEARANCE_WEARABLE_AUDIO_DEVICE_NECK_BAND (*C macro*), 2023
BT_APPEARANCE_WINDOW_AWNING (*C macro*), 2020
BT_APPEARANCE_WINDOW_BLINDS (*C macro*), 2020
BT_APPEARANCE_WINDOW_CURTAIN (*C macro*), 2020
BT_APPEARANCE_WINDOW_EXTERIOR_SCREEN (*C macro*), 2020
BT_APPEARANCE_WINDOW_EXTERIOR_SHUTTER (*C macro*), 2020
BT_APPEARANCE_WINDOW_SHADES (*C macro*), 2020
bt_att_chan_opt (*C enum*), 2072
bt_att_chan_opt.BT_ATT_CHAN_OPT_ENHANCED_ONLY (*C enumerator*), 2072
bt_att_chan_opt.BT_ATT_CHAN_OPT_NONE (*C enumerator*), 2072
bt_att_chan_opt.BT_ATT_CHAN_OPT_UNENHANCED_ONLY (*C enumerator*), 2072
BT_ATT_ERR_ATTRIBUTE_NOT_FOUND (*C macro*), 2071
BT_ATT_ERR_ATTRIBUTE_NOT_LONG (*C macro*), 2071
BT_ATT_ERR_AUTHENTICATION (*C macro*), 2071
BT_ATT_ERR_AUTHORIZATION (*C macro*), 2071
BT_ATT_ERR_CCC_IMPROPER_CONF (*C macro*), 2072
BT_ATT_ERR_DB_OUT_OF_SYNC (*C macro*), 2072
BT_ATT_ERR_ENCRYPTION_KEY_SIZE (*C macro*), 2071
BT_ATT_ERR_INSUFFICIENT_ENCRYPTION (*C macro*), 2071
BT_ATT_ERR_INSUFFICIENT_RESOURCES (*C macro*), 2071
BT_ATT_ERR_INVALID_ATTRIBUTE_LEN (*C macro*), 2071
BT_ATT_ERR_INVALID_HANDLE (*C macro*), 2070
BT_ATT_ERR_INVALID_OFFSET (*C macro*), 2071
BT_ATT_ERR_INVALID_PDU (*C macro*), 2071
BT_ATT_ERR_NOT_SUPPORTED (*C macro*), 2071
BT_ATT_ERR_OUT_OF_RANGE (*C macro*), 2072
BT_ATT_ERR_PREPARE_QUEUE_FULL (*C macro*), 2071
BT_ATT_ERR_PROCEDURE_IN_PROGRESS (*C macro*), 2072
BT_ATT_ERR_READ_NOT_PERMITTED (*C macro*), 2070
BT_ATT_ERR_SUCCESS (*C macro*), 2070
bt_att_err_to_str (*C function*), 2073
BT_ATT_ERR_UNLIKELY (*C macro*), 2071
BT_ATT_ERR_UNSUPPORTED_GROUP_TYPE (*C macro*), 2071
BT_ATT_ERR_VALUE_NOT_ALLOWED (*C macro*), 2072
BT_ATT_ERR_WRITE_NOT_PERMITTED (*C macro*), 2071
BT_ATT_ERR_WRITE_REQ_REJECTED (*C macro*), 2072
BT_ATT_FIRST_ATTRIBUTE_HANDLE (*C macro*), 2072
BT_ATT_LAST_ATTRIBUTE_HANDLE (*C macro*), 2072
BT_ATT_MAX_ATTRIBUTE_LEN (*C macro*), 2072
bt_audio_active_state (*C enum*), 1738
bt_audio_active_state.BT_AUDIO_ACTIVE_STATE_DISABLED (*C enumerator*), 1738
bt_audio_active_state.BT_AUDIO_ACTIVE_STATE_ENABLED (*C enumerator*), 1738
BT_AUDIO_BROADCAST_CODE_SIZE (*C macro*), 1729
BT_AUDIO_BROADCAST_ID_MAX (*C macro*), 1729
BT_AUDIO_BROADCAST_ID_SIZE (*C macro*), 1729
BT_AUDIO_BROADCAST_NAME_LEN_MAX (*C macro*), 1730
BT_AUDIO_BROADCAST_NAME_LEN_MIN (*C macro*), 1729
BT_AUDIO_CODEC_CAP (*C macro*), 1731
bt_audio_codec_cap (*C struct*), 1743
bt_audio_codec_cap_chan_count (*C enum*), 1734
BT_AUDIO_CODEC_CAP_CHAN_COUNT_MAX (*C macro*), 1730
BT_AUDIO_CODEC_CAP_CHAN_COUNT_MIN (*C macro*), 1730
BT_AUDIO_CODEC_CAP_CHAN_COUNT_SUPPORT (*C macro*), 1730
bt_audio_codec_cap_chan_count.BT_AUDIO_CODEC_CAP_CHAN_COUNT_1 (*C enumerator*), 1734
bt_audio_codec_cap_chan_count.BT_AUDIO_CODEC_CAP_CHAN_COUNT_2 (*C enumerator*), 1734

bt_audio_codec_cap_chan_count.BT_AUDIO_CODEC_CAP_CHAN_COUNT_3 (*C enumerator*), 1734
bt_audio_codec_cap_chan_count.BT_AUDIO_CODEC_CAP_CHAN_COUNT_4 (*C enumerator*), 1734
bt_audio_codec_cap_chan_count.BT_AUDIO_CODEC_CAP_CHAN_COUNT_5 (*C enumerator*), 1734
bt_audio_codec_cap_chan_count.BT_AUDIO_CODEC_CAP_CHAN_COUNT_6 (*C enumerator*), 1734
bt_audio_codec_cap_chan_count.BT_AUDIO_CODEC_CAP_CHAN_COUNT_7 (*C enumerator*), 1734
bt_audio_codec_cap_chan_count.BT_AUDIO_CODEC_CAP_CHAN_COUNT_8 (*C enumerator*), 1734
bt_audio_codec_cap_chan_count.BT_AUDIO_CODEC_CAP_CHAN_COUNT_ANY (*C enumerator*), 1734
bt_audio_codec_cap_frame_dur (*C enum*), 1733
bt_audio_codec_cap_frame_dur.BT_AUDIO_CODEC_CAP_DURATION_7_5 (*C enumerator*), 1733
bt_audio_codec_cap_frame_dur.BT_AUDIO_CODEC_CAP_DURATION_10 (*C enumerator*), 1733
bt_audio_codec_cap_frame_dur.BT_AUDIO_CODEC_CAP_DURATION_ANY (*C enumerator*), 1734
bt_audio_codec_cap_frame_dur.BT_AUDIO_CODEC_CAP_DURATION_PREFER_7_5 (*C enumerator*), 1734
bt_audio_codec_cap_frame_dur.BT_AUDIO_CODEC_CAP_DURATION_PREFER_10 (*C enumerator*), 1734
bt_audio_codec_cap_freq (*C enum*), 1732
bt_audio_codec_cap_freq.BT_AUDIO_CODEC_CAP_FREQ_8KHZ (*C enumerator*), 1732
bt_audio_codec_cap_freq.BT_AUDIO_CODEC_CAP_FREQ_11KHZ (*C enumerator*), 1732
bt_audio_codec_cap_freq.BT_AUDIO_CODEC_CAP_FREQ_16KHZ (*C enumerator*), 1732
bt_audio_codec_cap_freq.BT_AUDIO_CODEC_CAP_FREQ_22KHZ (*C enumerator*), 1733
bt_audio_codec_cap_freq.BT_AUDIO_CODEC_CAP_FREQ_24KHZ (*C enumerator*), 1733
bt_audio_codec_cap_freq.BT_AUDIO_CODEC_CAP_FREQ_32KHZ (*C enumerator*), 1733
bt_audio_codec_cap_freq.BT_AUDIO_CODEC_CAP_FREQ_44KHZ (*C enumerator*), 1733
bt_audio_codec_cap_freq.BT_AUDIO_CODEC_CAP_FREQ_48KHZ (*C enumerator*), 1733
bt_audio_codec_cap_freq.BT_AUDIO_CODEC_CAP_FREQ_88KHZ (*C enumerator*), 1733
bt_audio_codec_cap_freq.BT_AUDIO_CODEC_CAP_FREQ_96KHZ (*C enumerator*), 1733
bt_audio_codec_cap_freq.BT_AUDIO_CODEC_CAP_FREQ_176KHZ (*C enumerator*), 1733
bt_audio_codec_cap_freq.BT_AUDIO_CODEC_CAP_FREQ_192KHZ (*C enumerator*), 1733
bt_audio_codec_cap_freq.BT_AUDIO_CODEC_CAP_FREQ_384KHZ (*C enumerator*), 1733
bt_audio_codec_cap_freq.BT_AUDIO_CODEC_CAP_FREQ_ANY (*C enumerator*), 1733
bt_audio_codec_cap_type (*C enum*), 1732
bt_audio_codec_cap_type.BT_AUDIO_CODEC_CAP_TYPE_CHAN_COUNT (*C enumerator*), 1732
bt_audio_codec_cap_type.BT_AUDIO_CODEC_CAP_TYPE_DURATION (*C enumerator*), 1732
bt_audio_codec_cap_type.BT_AUDIO_CODEC_CAP_TYPE_FRAME_COUNT (*C enumerator*), 1732
bt_audio_codec_cap_type.BT_AUDIO_CODEC_CAP_TYPE_FRAME_LEN (*C enumerator*), 1732
bt_audio_codec_cap_type.BT_AUDIO_CODEC_CAP_TYPE_FREQ (*C enumerator*), 1732
bt_audio_codec_cap.cid (*C var*), 1743
bt_audio_codec_cap.ctlr_transcode (*C var*), 1743
bt_audio_codec_cap.data (*C var*), 1744
bt_audio_codec_cap.data_len (*C var*), 1744
bt_audio_codec_cap.id (*C var*), 1743
bt_audio_codec_cap.meta (*C var*), 1744
bt_audio_codec_cap.meta_len (*C var*), 1744
bt_audio_codec_cap.path_id (*C var*), 1743
bt_audio_codec_cap.vid (*C var*), 1744
BT_AUDIO_CODEC_CFG (*C macro*), 1730
bt_audio_codec_cfg (*C struct*), 1744
bt_audio_codec_cfg_frame_dur (*C enum*), 1736
bt_audio_codec_cfg_frame_dur_to_frame_dur_us (*C function*), 1748
bt_audio_codec_cfg_frame_dur_us_to_frame_dur (*C function*), 1748
bt_audio_codec_cfg_frame_dur.BT_AUDIO_CODEC_CFG_DURATION_7_5 (*C enumerator*), 1736
bt_audio_codec_cfg_frame_dur.BT_AUDIO_CODEC_CFG_DURATION_10 (*C enumerator*), 1736
bt_audio_codec_cfg_freq (*C enum*), 1735
bt_audio_codec_cfg_freq_hz_to_freq (*C function*), 1747
bt_audio_codec_cfg_freq_to_freq_hz (*C function*), 1747
bt_audio_codec_cfg_freq.BT_AUDIO_CODEC_CFG_FREQ_8KHZ (*C enumerator*), 1735
bt_audio_codec_cfg_freq.BT_AUDIO_CODEC_CFG_FREQ_11KHZ (*C enumerator*), 1735
bt_audio_codec_cfg_freq.BT_AUDIO_CODEC_CFG_FREQ_16KHZ (*C enumerator*), 1735

bt_audio_codec_cfg_freq.BT_AUDIO_CODEC_CFG_FREQ_22KHZ (*C enumerator*), 1735
bt_audio_codec_cfg_freq.BT_AUDIO_CODEC_CFG_FREQ_24KHZ (*C enumerator*), 1735
bt_audio_codec_cfg_freq.BT_AUDIO_CODEC_CFG_FREQ_32KHZ (*C enumerator*), 1735
bt_audio_codec_cfg_freq.BT_AUDIO_CODEC_CFG_FREQ_44KHZ (*C enumerator*), 1735
bt_audio_codec_cfg_freq.BT_AUDIO_CODEC_CFG_FREQ_48KHZ (*C enumerator*), 1735
bt_audio_codec_cfg_freq.BT_AUDIO_CODEC_CFG_FREQ_88KHZ (*C enumerator*), 1736
bt_audio_codec_cfg_freq.BT_AUDIO_CODEC_CFG_FREQ_96KHZ (*C enumerator*), 1736
bt_audio_codec_cfg_freq.BT_AUDIO_CODEC_CFG_FREQ_176KHZ (*C enumerator*), 1736
bt_audio_codec_cfg_freq.BT_AUDIO_CODEC_CFG_FREQ_192KHZ (*C enumerator*), 1736
bt_audio_codec_cfg_freq.BT_AUDIO_CODEC_CFG_FREQ_384KHZ (*C enumerator*), 1736
bt_audio_codec_cfg_get_chan_allocation (*C function*), 1749
bt_audio_codec_cfg_get_frame_blocks_per_sdu (*C function*), 1750
bt_audio_codec_cfg_get_frame_dur (*C function*), 1748
bt_audio_codec_cfg_get_freq (*C function*), 1747
bt_audio_codec_cfg_get_octets_per_frame (*C function*), 1749
bt_audio_codec_cfg_get_val (*C function*), 1751
bt_audio_codec_cfg_meta_get_audio_active_state (*C function*), 1757
bt_audio_codec_cfg_meta_get_bcast_audio_immediate_rend_flag (*C function*), 1757
bt_audio_codec_cfg_meta_get_ccid_list (*C function*), 1755
bt_audio_codec_cfg_meta_get_extended (*C function*), 1758
bt_audio_codec_cfg_meta_get_lang (*C function*), 1754
bt_audio_codec_cfg_meta_get_parental_rating (*C function*), 1755
bt_audio_codec_cfg_meta_get_pref_context (*C function*), 1752
bt_audio_codec_cfg_meta_get_program_info (*C function*), 1754
bt_audio_codec_cfg_meta_get_program_info_uri (*C function*), 1756
bt_audio_codec_cfg_meta_get_stream_context (*C function*), 1753
bt_audio_codec_cfg_meta_get_val (*C function*), 1752
bt_audio_codec_cfg_meta_get_vendor (*C function*), 1758
bt_audio_codec_cfg_meta_set_audio_active_state (*C function*), 1757
bt_audio_codec_cfg_meta_set_bcast_audio_immediate_rend_flag (*C function*), 1757
bt_audio_codec_cfg_meta_set_ccid_list (*C function*), 1755
bt_audio_codec_cfg_meta_set_extended (*C function*), 1758
bt_audio_codec_cfg_meta_set_lang (*C function*), 1755
bt_audio_codec_cfg_meta_set_parental_rating (*C function*), 1756
bt_audio_codec_cfg_meta_set_pref_context (*C function*), 1753
bt_audio_codec_cfg_meta_set_program_info (*C function*), 1754
bt_audio_codec_cfg_meta_set_program_info_uri (*C function*), 1756
bt_audio_codec_cfg_meta_set_stream_context (*C function*), 1753
bt_audio_codec_cfg_meta_set_val (*C function*), 1752
bt_audio_codec_cfg_meta_set_vendor (*C function*), 1758
bt_audio_codec_cfg_meta_unset_val (*C function*), 1752
bt_audio_codec_cfg_set_chan_allocation (*C function*), 1749
bt_audio_codec_cfg_set_frame_blocks_per_sdu (*C function*), 1751
bt_audio_codec_cfg_set_frame_dur (*C function*), 1748
bt_audio_codec_cfg_set_freq (*C function*), 1748
bt_audio_codec_cfg_set_octets_per_frame (*C function*), 1750
bt_audio_codec_cfg_set_val (*C function*), 1751
bt_audio_codec_cfg_type (*C enum*), 1735
bt_audio_codec_cfg_type.BT_AUDIO_CODEC_CFG_CHAN_ALLOC (*C enumerator*), 1735
bt_audio_codec_cfg_type.BT_AUDIO_CODEC_CFG_DURATION (*C enumerator*), 1735
bt_audio_codec_cfg_type.BT_AUDIO_CODEC_CFG_FRAME_BLKS_PER_SDU (*C enumerator*), 1735
bt_audio_codec_cfg_type.BT_AUDIO_CODEC_CFG_FRAME_LEN (*C enumerator*), 1735
bt_audio_codec_cfg_type.BT_AUDIO_CODEC_CFG_FREQ (*C enumerator*), 1735
bt_audio_codec_cfg_unset_val (*C function*), 1751
bt_audio_codec_cfg.cid (*C var*), 1744
bt_audio_codec_cfg.ctlr_transcode (*C var*), 1744
bt_audio_codec_cfg.data (*C var*), 1744
bt_audio_codec_cfg.data_len (*C var*), 1744

bt_audio_codec_cfg.id (*C var*), 1744
bt_audio_codec_cfg.meta (*C var*), 1745
bt_audio_codec_cfg.meta_len (*C var*), 1744
bt_audio_codec_cfg.path_id (*C var*), 1744
bt_audio_codec_cfg.vid (*C var*), 1744
BT_AUDIO_CODEC_DATA (*C macro*), 1730
bt_audio_codec_octets_per_codec_frame (*C struct*), 1743
bt_audio_codec_octets_per_codec_frame.max (*C var*), 1743
bt_audio_codec_octets_per_codec_frame.min (*C var*), 1743
BT_AUDIO_CODEC_QOS (*C macro*), 1731
bt_audio_codec_qos (*C struct*), 1745
BT_AUDIO_CODEC_QOS_FRAMED (*C macro*), 1731
bt_audio_codec_qos_framing (*C enum*), 1742
bt_audio_codec_qos_framing.BT_AUDIO_CODEC_QOS_FRAMING_FRAMED (*C enumerator*), 1742
bt_audio_codec_qos_framing.BT_AUDIO_CODEC_QOS_FRAMING_UNFRAMED (*C enumerator*), 1742
BT_AUDIO_CODEC_QOS_PREF (*C macro*), 1731
bt_audio_codec_qos_pref (*C struct*), 1746
bt_audio_codec_qos_pref.latency (*C var*), 1746
bt_audio_codec_qos_pref.pd_max (*C var*), 1747
bt_audio_codec_qos_pref.pd_min (*C var*), 1746
bt_audio_codec_qos_pref.phy (*C var*), 1746
bt_audio_codec_qos_pref.pref_pd_max (*C var*), 1747
bt_audio_codec_qos_pref.pref_pd_min (*C var*), 1747
bt_audio_codec_qos_pref.rtn (*C var*), 1746
bt_audio_codec_qos_pref.unframed_supported (*C var*), 1746
BT_AUDIO_CODEC_QOS_UNFRAMED (*C macro*), 1731
bt_audio_codec_qos.burst_number (*C var*), 1746
bt_audio_codec_qos.framing (*C var*), 1745
bt_audio_codec_qos.interval (*C var*), 1745
bt_audio_codec_qos.latency (*C var*), 1745
bt_audio_codec_qos.max_pdu (*C var*), 1745
bt_audio_codec_qos.num_subevents (*C var*), 1746
bt_audio_codec_qos.pd (*C var*), 1745
bt_audio_codec_qos.phy (*C var*), 1745
bt_audio_codec_qos.rtn (*C var*), 1745
bt_audio_codec_qos.sdu (*C var*), 1745
bt_audio_context (*C enum*), 1736
BT_AUDIO_CONTEXT_TYPE_ANY (*C macro*), 1730
bt_audio_context.BT_AUDIO_CONTEXT_TYPE_ALERTS (*C enumerator*), 1737
bt_audio_context.BT_AUDIO_CONTEXT_TYPE_CONVERSATIONAL (*C enumerator*), 1736
bt_audio_context.BT_AUDIO_CONTEXT_TYPE_EMERGENCY_ALARM (*C enumerator*), 1737
bt_audio_context.BT_AUDIO_CONTEXT_TYPE_GAME (*C enumerator*), 1736
bt_audio_context.BT_AUDIO_CONTEXT_TYPE_INSTRUCTIONAL (*C enumerator*), 1737
bt_audio_context.BT_AUDIO_CONTEXT_TYPE_LIVE (*C enumerator*), 1737
bt_audio_context.BT_AUDIO_CONTEXT_TYPE_MEDIA (*C enumerator*), 1736
bt_audio_context.BT_AUDIO_CONTEXT_TYPE_NOTIFICATIONS (*C enumerator*), 1737
bt_audio_context.BT_AUDIO_CONTEXT_TYPE_PROHIBITED (*C enumerator*), 1736
bt_audio_context.BT_AUDIO_CONTEXT_TYPE_RINGTONE (*C enumerator*), 1737
bt_audio_context.BT_AUDIO_CONTEXT_TYPE_SOUND_EFFECTS (*C enumerator*), 1737
bt_audio_context.BT_AUDIO_CONTEXT_TYPE_UNSPECIFIED (*C enumerator*), 1736
bt_audio_context.BT_AUDIO_CONTEXT_TYPE_VOICE_ASSISTANTS (*C enumerator*), 1737
bt_audio_data_parse (*C function*), 1742
bt_audio_dir (*C enum*), 1741
bt_audio_dir.BT_AUDIO_DIR_SINK (*C enumerator*), 1742
bt_audio_dir.BT_AUDIO_DIR_SOURCE (*C enumerator*), 1742
bt_audio_get_chan_count (*C function*), 1743
BT_AUDIO_LANG_SIZE (*C macro*), 1730
bt_audio_location (*C enum*), 1740

BT_AUDIO_LOCATION_ANY (*C macro*), 1731
bt_audio_location.BT_AUDIO_LOCATION_BACK_CENTER (*C enumerator*), 1740
bt_audio_location.BT_AUDIO_LOCATION_BACK_LEFT (*C enumerator*), 1740
bt_audio_location.BT_AUDIO_LOCATION_BACK_RIGHT (*C enumerator*), 1740
bt_audio_location.BT_AUDIO_LOCATION_BOTTOM_FRONT_CENTER (*C enumerator*), 1741
bt_audio_location.BT_AUDIO_LOCATION_BOTTOM_FRONT_LEFT (*C enumerator*), 1741
bt_audio_location.BT_AUDIO_LOCATION_BOTTOM_FRONT_RIGHT (*C enumerator*), 1741
bt_audio_location.BT_AUDIO_LOCATION_FRONT_CENTER (*C enumerator*), 1740
bt_audio_location.BT_AUDIO_LOCATION_FRONT_LEFT (*C enumerator*), 1740
bt_audio_location.BT_AUDIO_LOCATION_FRONT_LEFT_OF_CENTER (*C enumerator*), 1740
bt_audio_location.BT_AUDIO_LOCATION_FRONT_LEFT_WIDE (*C enumerator*), 1741
bt_audio_location.BT_AUDIO_LOCATION_FRONT_RIGHT (*C enumerator*), 1740
bt_audio_location.BT_AUDIO_LOCATION_FRONT_RIGHT_OF_CENTER (*C enumerator*), 1740
bt_audio_location.BT_AUDIO_LOCATION_FRONT_RIGHT_WIDE (*C enumerator*), 1741
bt_audio_location.BT_AUDIO_LOCATION_LEFT_SURROUND (*C enumerator*), 1741
bt_audio_location.BT_AUDIO_LOCATION_LOW_FREQ_EFFECTS_1 (*C enumerator*), 1740
bt_audio_location.BT_AUDIO_LOCATION_LOW_FREQ_EFFECTS_2 (*C enumerator*), 1740
bt_audio_location.BT_AUDIO_LOCATION_MONO_AUDIO (*C enumerator*), 1740
bt_audio_location.BT_AUDIO_LOCATION_RIGHT_SURROUND (*C enumerator*), 1741
bt_audio_location.BT_AUDIO_LOCATION_SIDE_LEFT (*C enumerator*), 1740
bt_audio_location.BT_AUDIO_LOCATION_SIDE_RIGHT (*C enumerator*), 1740
bt_audio_location.BT_AUDIO_LOCATION_TOP_BACK_CENTER (*C enumerator*), 1741
bt_audio_location.BT_AUDIO_LOCATION_TOP_BACK_LEFT (*C enumerator*), 1741
bt_audio_location.BT_AUDIO_LOCATION_TOP_BACK_RIGHT (*C enumerator*), 1741
bt_audio_location.BT_AUDIO_LOCATION_TOP_CENTER (*C enumerator*), 1741
bt_audio_location.BT_AUDIO_LOCATION_TOP_FRONT_CENTER (*C enumerator*), 1741
bt_audio_location.BT_AUDIO_LOCATION_TOP_FRONT_LEFT (*C enumerator*), 1740
bt_audio_location.BT_AUDIO_LOCATION_TOP_FRONT_RIGHT (*C enumerator*), 1741
bt_audio_location.BT_AUDIO_LOCATION_TOP_SIDE_LEFT (*C enumerator*), 1741
bt_audio_location.BT_AUDIO_LOCATION_TOP_SIDE_RIGHT (*C enumerator*), 1741
bt_audio_metadata_type (*C enum*), 1738
BT_AUDIO_METADATA_TYPE_IS_KNOWN (*C macro*), 1730
bt_audio_metadata_type.BT_AUDIO_METADATA_TYPE_AUDIO_STATE (*C enumerator*), 1739
bt_audio_metadata_type.BT_AUDIO_METADATA_TYPE_BROADCAST_IMMEDIATE (*C enumerator*), 1739
bt_audio_metadata_type.BT_AUDIO_METADATA_TYPE_CCID_LIST (*C enumerator*), 1739
bt_audio_metadata_type.BT_AUDIO_METADATA_TYPE_EXTENDED (*C enumerator*), 1739
bt_audio_metadata_type.BT_AUDIO_METADATA_TYPE_LANG (*C enumerator*), 1739
bt_audio_metadata_type.BT_AUDIO_METADATA_TYPE_PARENTAL_RATING (*C enumerator*), 1739
bt_audio_metadata_type.BT_AUDIO_METADATA_TYPE_PREF_CONTEXT (*C enumerator*), 1739
bt_audio_metadata_type.BT_AUDIO_METADATA_TYPE_PROGRAM_INFO (*C enumerator*), 1739
bt_audio_metadata_type.BT_AUDIO_METADATA_TYPE_PROGRAM_INFO_URI (*C enumerator*), 1739
bt_audio_metadata_type.BT_AUDIO_METADATA_TYPE_STREAM_CONTEXT (*C enumerator*), 1739
bt_audio_metadata_type.BT_AUDIO_METADATA_TYPE_VENDOR (*C enumerator*), 1739
bt_audio_parental_rating (*C enum*), 1737
bt_audio_parental_rating.BT_AUDIO_PARENTAL_RATING_AGE_5_OR_ABOVE (*C enumerator*), 1737
bt_audio_parental_rating.BT_AUDIO_PARENTAL_RATING_AGE_6_OR_ABOVE (*C enumerator*), 1737
bt_audio_parental_rating.BT_AUDIO_PARENTAL_RATING_AGE_7_OR_ABOVE (*C enumerator*), 1738
bt_audio_parental_rating.BT_AUDIO_PARENTAL_RATING_AGE_8_OR_ABOVE (*C enumerator*), 1738
bt_audio_parental_rating.BT_AUDIO_PARENTAL_RATING_AGE_9_OR_ABOVE (*C enumerator*), 1738
bt_audio_parental_rating.BT_AUDIO_PARENTAL_RATING_AGE_10_OR_ABOVE (*C enumerator*), 1738
bt_audio_parental_rating.BT_AUDIO_PARENTAL_RATING_AGE_11_OR_ABOVE (*C enumerator*), 1738
bt_audio_parental_rating.BT_AUDIO_PARENTAL_RATING_AGE_12_OR_ABOVE (*C enumerator*), 1738
bt_audio_parental_rating.BT_AUDIO_PARENTAL_RATING_AGE_13_OR_ABOVE (*C enumerator*), 1738
bt_audio_parental_rating.BT_AUDIO_PARENTAL_RATING_AGE_14_OR_ABOVE (*C enumerator*), 1738
bt_audio_parental_rating.BT_AUDIO_PARENTAL_RATING_AGE_15_OR_ABOVE (*C enumerator*), 1738
bt_audio_parental_rating.BT_AUDIO_PARENTAL_RATING_AGE_16_OR_ABOVE (*C enumerator*), 1738
bt_audio_parental_rating.BT_AUDIO_PARENTAL_RATING_AGE_17_OR_ABOVE (*C enumerator*), 1738
bt_audio_parental_rating.BT_AUDIO_PARENTAL_RATING_AGE_18_OR_ABOVE (*C enumerator*), 1738

bt_audio_parental_rating.BT_AUDIO_PARENTAL_RATING_AGE_ANY (*C enumerator*), 1737
bt_audio_parental_rating.BT_AUDIO_PARENTAL_RATING_NO_RATING (*C enumerator*), 1737
BT_AUDIO_PD_MAX (*C macro*), 1729
BT_AUDIO_PD_PREF_NONE (*C macro*), 1729
BT_AUDIO_UNICAST_ANNOUNCEMENT_GENERAL (*C macro*), 1729
BT_AUDIO_UNICAST_ANNOUNCEMENT_TARGETED (*C macro*), 1729
bt_bap_ascs_reason (*C enum*), 1763
bt_bap_ascs_reason.BT_BAP_ASCS_REASON_CIS (*C enumerator*), 1763
bt_bap_ascs_reason.BT_BAP_ASCS_REASON_CODEC (*C enumerator*), 1763
bt_bap_ascs_reason.BT_BAP_ASCS_REASON_CODEC_DATA (*C enumerator*), 1763
bt_bap_ascs_reason.BT_BAP_ASCS_REASON_FRAMING (*C enumerator*), 1763
bt_bap_ascs_reason.BT_BAP_ASCS_REASON_INTERVAL (*C enumerator*), 1763
bt_bap_ascs_reason.BT_BAP_ASCS_REASON_LATENCY (*C enumerator*), 1763
bt_bap_ascs_reason.BT_BAP_ASCS_REASON_NONE (*C enumerator*), 1763
bt_bap_ascs_reason.BT_BAP_ASCS_REASON_PD (*C enumerator*), 1763
bt_bap_ascs_reason.BT_BAP_ASCS_REASON_PHY (*C enumerator*), 1763
bt_bap_ascs_reason.BT_BAP_ASCS_REASON_RTN (*C enumerator*), 1763
bt_bap_ascs_reason.BT_BAP_ASCS_REASON_SDU (*C enumerator*), 1763
BT_BAP_ASCS_RSP (*C macro*), 1759
bt_bap_ascs_rsp (*C struct*), 1772
bt_bap_ascs_rsp_code (*C enum*), 1762
bt_bap_ascs_rsp_code.BT_BAP_ASCS_RSP_CODE_CAP_UNSUPPORTED (*C enumerator*), 1762
bt_bap_ascs_rsp_code.BT_BAP_ASCS_RSP_CODE_CONF_INVALID (*C enumerator*), 1762
bt_bap_ascs_rsp_code.BT_BAP_ASCS_RSP_CODE_CONF_REJECTED (*C enumerator*), 1762
bt_bap_ascs_rsp_code.BT_BAP_ASCS_RSP_CODE_CONF_UNSUPPORTED (*C enumerator*), 1762
bt_bap_ascs_rsp_code.BT_BAP_ASCS_RSP_CODE_INVALID_ASE (*C enumerator*), 1762
bt_bap_ascs_rsp_code.BT_BAP_ASCS_RSP_CODE_INVALID_ASE_STATE (*C enumerator*), 1762
bt_bap_ascs_rsp_code.BT_BAP_ASCS_RSP_CODE_INVALID_DIR (*C enumerator*), 1762
bt_bap_ascs_rsp_code.BT_BAP_ASCS_RSP_CODE_INVALID_LENGTH (*C enumerator*), 1762
bt_bap_ascs_rsp_code.BT_BAP_ASCS_RSP_CODE_METADATA_INVALID (*C enumerator*), 1763
bt_bap_ascs_rsp_code.BT_BAP_ASCS_RSP_CODE_METADATA_REJECTED (*C enumerator*), 1762
bt_bap_ascs_rsp_code.BT_BAP_ASCS_RSP_CODE_METADATA_UNSUPPORTED (*C enumerator*), 1762
bt_bap_ascs_rsp_code.BT_BAP_ASCS_RSP_CODE_NO_MEM (*C enumerator*), 1763
bt_bap_ascs_rsp_code.BT_BAP_ASCS_RSP_CODE_NOT_SUPPORTED (*C enumerator*), 1762
bt_bap_ascs_rsp_code.BT_BAP_ASCS_RSP_CODE_SUCCESS (*C enumerator*), 1762
bt_bap_ascs_rsp_code.BT_BAP_ASCS_RSP_CODE_UNSPECIFIED (*C enumerator*), 1763
bt_bap_ascs_rsp.code (*C var*), 1772
bt_bap_ascs_rsp.metadata_type (*C var*), 1772
bt_bap_ascs_rsp.reason (*C var*), 1772
bt_bap_base_codec_id (*C struct*), 1795
bt_bap_base_codec_id.cid (*C var*), 1795
bt_bap_base_codec_id.id (*C var*), 1795
bt_bap_base_codec_id.vid (*C var*), 1796
bt_bap_base_foreach_subgroup (*C function*), 1793
bt_bap_base_get_base_from_ad (*C function*), 1792
bt_bap_base_get_bis_indexes (*C function*), 1793
bt_bap_base_get_pres_delay (*C function*), 1792
bt_bap_base_get_size (*C function*), 1792
bt_bap_base_get_subgroup_bis_count (*C function*), 1794
bt_bap_base_get_subgroup_codec_data (*C function*), 1793
bt_bap_base_get_subgroup_codec_id (*C function*), 1793
bt_bap_base_get_subgroup_codec_meta (*C function*), 1794
bt_bap_base_get_subgroup_count (*C function*), 1793
bt_bap_base_subgroup_bis (*C struct*), 1796
bt_bap_base_subgroup_bis_codec_to_codec_cfg (*C function*), 1795
bt_bap_base_subgroup_bis.data (*C var*), 1796
bt_bap_base_subgroup_bis.data_len (*C var*), 1796
bt_bap_base_subgroup_bis.index (*C var*), 1796

bt_bap_base_subgroup_codec_to_codec_cfg (*C function*), 1794
bt_bap_base_subgroup_foreach_bis (*C function*), 1795
bt_bap_base_subgroup_get_bis_indexes (*C function*), 1794
bt_bap_bass_att_err (*C enum*), 1761
bt_bap_bass_att_err.BT_BAP_BASS_ERR_INVALID_SRC_ID (*C enumerator*), 1761
bt_bap_bass_att_err.BT_BAP_BASS_ERR_OPCODE_NOT_SUPPORTED (*C enumerator*), 1761
bt_bap_bass_subgroup (*C struct*), 1773
bt_bap_bass_subgroup.bis_sync (*C var*), 1773
bt_bap_bass_subgroup.metadata (*C var*), 1773
bt_bap_bass_subgroup.metadata_len (*C var*), 1773
bt_bap_big_enc_state (*C enum*), 1761
bt_bap_big_enc_state.BT_BAP_BIG_ENC_STATE_BAD_CODE (*C enumerator*), 1761
bt_bap_big_enc_state.BT_BAP_BIG_ENC_STATE_BCODE_REQ (*C enumerator*), 1761
bt_bap_big_enc_state.BT_BAP_BIG_ENC_STATE_DEC (*C enumerator*), 1761
bt_bap_big_enc_state.BT_BAP_BIG_ENC_STATE_NO_ENC (*C enumerator*), 1761
BT_BAP_BIS_SYNC_NO_PREF (*C macro*), 1759
bt_bap_broadcast_assistant_add_src (*C function*), 1771
bt_bap_broadcast_assistant_add_src_param (*C struct*), 1781
bt_bap_broadcast_assistant_add_src_param.addr (*C var*), 1781
bt_bap_broadcast_assistant_add_src_param.adv_sid (*C var*), 1781
bt_bap_broadcast_assistant_add_src_param.broadcast_id (*C var*), 1781
bt_bap_broadcast_assistant_add_src_param.num_subgroups (*C var*), 1781
bt_bap_broadcast_assistant_add_src_param.pa_interval (*C var*), 1781
bt_bap_broadcast_assistant_add_src_param.pa_sync (*C var*), 1781
bt_bap_broadcast_assistant_add_src_param.subgroups (*C var*), 1782
bt_bap_broadcast_assistant_cb (*C struct*), 1779
bt_bap_broadcast_assistant_cb.add_src (*C var*), 1780
bt_bap_broadcast_assistant_cb.broadcast_code (*C var*), 1781
bt_bap_broadcast_assistant_cb.discover (*C var*), 1780
bt_bap_broadcast_assistant_cb.mod_src (*C var*), 1781
bt_bap_broadcast_assistant_cb.recv_state (*C var*), 1780
bt_bap_broadcast_assistant_cb.recv_state_removed (*C var*), 1780
bt_bap_broadcast_assistant_cb.rem_src (*C var*), 1781
bt_bap_broadcast_assistant_cb.scan (*C var*), 1780
bt_bap_broadcast_assistant_cb.scan_start (*C var*), 1780
bt_bap_broadcast_assistant_cb.scan_stop (*C var*), 1780
bt_bap_broadcast_assistant_discover (*C function*), 1770
bt_bap_broadcast_assistant_mod_src (*C function*), 1771
bt_bap_broadcast_assistant_mod_src_param (*C struct*), 1782
bt_bap_broadcast_assistant_mod_src_param.num_subgroups (*C var*), 1782
bt_bap_broadcast_assistant_mod_src_param.pa_interval (*C var*), 1782
bt_bap_broadcast_assistant_mod_src_param.pa_sync (*C var*), 1782
bt_bap_broadcast_assistant_mod_src_param.src_id (*C var*), 1782
bt_bap_broadcast_assistant_mod_src_param.subgroups (*C var*), 1782
bt_bap_broadcast_assistant_read_recv_state (*C function*), 1772
bt_bap_broadcast_assistant_register_cb (*C function*), 1770
bt_bap_broadcast_assistant_rem_src (*C function*), 1771
bt_bap_broadcast_assistant_scan_start (*C function*), 1770
bt_bap_broadcast_assistant_scan_stop (*C function*), 1770
bt_bap_broadcast_assistant_set_broadcast_code (*C function*), 1771
bt_bap_broadcast_assistant_unregister_cb (*C function*), 1771
bt_bap_broadcast_assistant_write_cb (*C type*), 1760
bt_bap_broadcast_sink_cb (*C struct*), 1797
bt_bap_broadcast_sink_cb.base_recv (*C var*), 1798
bt_bap_broadcast_sink_cb.syncable (*C var*), 1798
bt_bap_broadcast_sink_create (*C function*), 1796
bt_bap_broadcast_sink_delete (*C function*), 1797
bt_bap_broadcast_sink_register_cb (*C function*), 1796

bt_bap_broadcast_sink_stop (*C function*), 1797
bt_bap_broadcast_sink_sync (*C function*), 1797
bt_bap_broadcast_source_create (*C function*), 1798
bt_bap_broadcast_source_delete (*C function*), 1800
bt_bap_broadcast_source_get_base (*C function*), 1800
bt_bap_broadcast_source_get_id (*C function*), 1800
bt_bap_broadcast_source_param (*C struct*), 1801
bt_bap_broadcast_source_param.broadcast_code (*C var*), 1801
bt_bap_broadcast_source_param.encryption (*C var*), 1801
bt_bap_broadcast_source_param.irc (*C var*), 1802
bt_bap_broadcast_source_param.iso_interval (*C var*), 1802
bt_bap_broadcast_source_param.packing (*C var*), 1801
bt_bap_broadcast_source_param.params (*C var*), 1801
bt_bap_broadcast_source_param.params_count (*C var*), 1801
bt_bap_broadcast_source_param.pto (*C var*), 1802
bt_bap_broadcast_source_param.qos (*C var*), 1801
bt_bap_broadcast_source_reconfig (*C function*), 1798
bt_bap_broadcast_source_start (*C function*), 1799
bt_bap_broadcast_source_stop (*C function*), 1799
bt_bap_broadcast_source_stream_param (*C struct*), 1800
bt_bap_broadcast_source_stream_param.data (*C var*), 1801
bt_bap_broadcast_source_stream_param.data_len (*C var*), 1800
bt_bap_broadcast_source_stream_param.stream (*C var*), 1800
bt_bap_broadcast_source_subgroup_param (*C struct*), 1801
bt_bap_broadcast_source_subgroup_param.codec_cfg (*C var*), 1801
bt_bap_broadcast_source_subgroup_param.params (*C var*), 1801
bt_bap_broadcast_source_subgroup_param.params_count (*C var*), 1801
bt_bap_broadcast_source_update_metadata (*C function*), 1799
bt_bap_ep_func_t (*C type*), 1788
bt_bap_ep_get_info (*C function*), 1764
bt_bap_ep_info (*C struct*), 1775
bt_bap_ep_info.can_recv (*C var*), 1776
bt_bap_ep_info.can_send (*C var*), 1775
bt_bap_ep_info.dir (*C var*), 1775
bt_bap_ep_info.id (*C var*), 1775
bt_bap_ep_info.iso_chan (*C var*), 1775
bt_bap_ep_info.paired_ep (*C var*), 1776
bt_bap_ep_info.qos_pref (*C var*), 1776
bt_bap_ep_info.state (*C var*), 1775
bt_bap_ep_state (*C enum*), 1761
bt_bap_ep_state.BT_BAP_EP_STATE_CODEC_CONFIGURED (*C enumerator*), 1761
bt_bap_ep_state.BT_BAP_EP_STATE_DISABLING (*C enumerator*), 1762
bt_bap_ep_state.BT_BAP_EP_STATE_ENABLING (*C enumerator*), 1761
bt_bap_ep_state.BT_BAP_EP_STATE_IDLE (*C enumerator*), 1761
bt_bap_ep_state.BT_BAP_EP_STATE_QOS_CONFIGURED (*C enumerator*), 1761
bt_bap_ep_state.BT_BAP_EP_STATE_RELEASING (*C enumerator*), 1762
bt_bap_ep_state.BT_BAP_EP_STATE_STREAMING (*C enumerator*), 1761
BT_BAP_PA_INTERVAL_UNKNOWN (*C macro*), 1759
bt_bap_pa_state (*C enum*), 1760
bt_bap_pa_state.BT_BAP_PA_STATE_FAILED (*C enumerator*), 1760
bt_bap_pa_state.BT_BAP_PA_STATE_INFO_REQ (*C enumerator*), 1760
bt_bap_pa_state.BT_BAP_PA_STATE_NO_PAST (*C enumerator*), 1761
bt_bap_pa_state.BT_BAP_PA_STATE_NOT_SYNCED (*C enumerator*), 1760
bt_bap_pa_state.BT_BAP_PA_STATE_SYNCED (*C enumerator*), 1760
bt_bap_scan_delegator_add_src (*C function*), 1769
bt_bap_scan_delegator_add_src_param (*C struct*), 1778
bt_bap_scan_delegator_add_src_param.broadcast_id (*C var*), 1779
bt_bap_scan_delegator_add_src_param.encrypt_state (*C var*), 1779

bt_bap_scan_delegator_add_src_param.num_subgroups (*C var*), 1779
bt_bap_scan_delegator_add_src_param.pa_sync (*C var*), 1779
bt_bap_scan_delegator_add_src_param.subgroups (*C var*), 1779
bt_bap_scan_delegator_cb (*C struct*), 1774
bt_bap_scan_delegator_cb.bis_sync_req (*C var*), 1775
bt_bap_scan_delegator_cb.broadcast_code (*C var*), 1775
bt_bap_scan_delegator_cb.pa_sync_req (*C var*), 1774
bt_bap_scan_delegator_cb.pa_sync_term_req (*C var*), 1774
bt_bap_scan_delegator_cb.recv_state_updated (*C var*), 1774
bt_bap_scan_delegator_find_state (*C function*), 1769
bt_bap_scan_delegator_foreach_state (*C function*), 1769
bt_bap_scan_delegator_mod_src (*C function*), 1769
bt_bap_scan_delegator_mod_src_param (*C struct*), 1779
bt_bap_scan_delegator_mod_src_param.broadcast_id (*C var*), 1779
bt_bap_scan_delegator_mod_src_param.encrypt_state (*C var*), 1779
bt_bap_scan_delegator_mod_src_param.num_subgroups (*C var*), 1779
bt_bap_scan_delegator_mod_src_param.src_id (*C var*), 1779
bt_bap_scan_delegator_mod_src_param.subgroups (*C var*), 1779
bt_bap_scan_delegator_recv_state (*C struct*), 1773
bt_bap_scan_delegator_recv_state.addr (*C var*), 1773
bt_bap_scan_delegator_recv_state.adv_sid (*C var*), 1773
bt_bap_scan_delegator_recv_state.bad_code (*C var*), 1773
bt_bap_scan_delegator_recv_state.broadcast_id (*C var*), 1773
bt_bap_scan_delegator_recv_state.encrypt_state (*C var*), 1773
bt_bap_scan_delegator_recv_state.num_subgroups (*C var*), 1773
bt_bap_scan_delegator_recv_state.pa_sync_state (*C var*), 1773
bt_bap_scan_delegator_recv_state.src_id (*C var*), 1773
bt_bap_scan_delegator_recv_state.subgroups (*C var*), 1774
bt_bap_scan_delegator_register_cb (*C function*), 1768
bt_bap_scan_delegator_rem_src (*C function*), 1769
bt_bap_scan_delegator_set_bis_sync_state (*C function*), 1768
bt_bap_scan_delegator_set_pa_state (*C function*), 1768
bt_bap_scan_delegator_state_func_t (*C type*), 1760
bt_bap_stream (*C struct*), 1776
bt_bap_stream_cb_register (*C function*), 1764
bt_bap_stream_config (*C function*), 1764
bt_bap_stream_connect (*C function*), 1766
bt_bap_stream_disable (*C function*), 1765
bt_bap_stream_enable (*C function*), 1765
bt_bap_stream_get_tx_sync (*C function*), 1768
bt_bap_stream_metadata (*C function*), 1765
bt_bap_stream_ops (*C struct*), 1776
bt_bap_stream_ops.configured (*C var*), 1777
bt_bap_stream_ops.connected (*C var*), 1778
bt_bap_stream_ops.disabled (*C var*), 1777
bt_bap_stream_ops.disconnected (*C var*), 1778
bt_bap_stream_ops.enabled (*C var*), 1777
bt_bap_stream_ops.metadata_updated (*C var*), 1777
bt_bap_stream_ops.qos_set (*C var*), 1777
bt_bap_stream_ops.recv (*C var*), 1778
bt_bap_stream_ops.released (*C var*), 1777
bt_bap_stream_ops.sent (*C var*), 1778
bt_bap_stream_ops.started (*C var*), 1777
bt_bap_stream_ops.stopped (*C var*), 1777
bt_bap_stream_qos (*C function*), 1765
bt_bap_stream_reconfig (*C function*), 1764
bt_bap_stream_release (*C function*), 1767
bt_bap_stream_send (*C function*), 1767

bt_bap_stream_send_ts (*C function*), 1767
bt_bap_stream_start (*C function*), 1766
bt_bap_stream_stop (*C function*), 1766
bt_bap_stream.codec_cfg (*C var*), 1776
bt_bap_stream.conn (*C var*), 1776
bt_bap_stream.ep (*C var*), 1776
bt_bap_stream.ops (*C var*), 1776
bt_bap_stream.qos (*C var*), 1776
bt_bap_stream.user_data (*C var*), 1776
bt_bap_unicast_client_cb (*C struct*), 1785
bt_bap_unicast_client_cb.available_contexts (*C var*), 1785
bt_bap_unicast_client_cb.config (*C var*), 1785
bt_bap_unicast_client_cb.disable (*C var*), 1787
bt_bap_unicast_client_cb.discover (*C var*), 1788
bt_bap_unicast_client_cb.enable (*C var*), 1786
bt_bap_unicast_client_cb.endpoint (*C var*), 1788
bt_bap_unicast_client_cb.location (*C var*), 1785
bt_bap_unicast_client_cb.metadata (*C var*), 1787
bt_bap_unicast_client_cb.pac_record (*C var*), 1787
bt_bap_unicast_client_cb.qos (*C var*), 1786
bt_bap_unicast_client_cb.release (*C var*), 1787
bt_bap_unicast_client_cb.start (*C var*), 1786
bt_bap_unicast_client_cb.stop (*C var*), 1786
bt_bap_unicast_client_discover (*C function*), 1783
bt_bap_unicast_client_register_cb (*C function*), 1783
bt_bap_unicast_group_add_streams (*C function*), 1782
bt_bap_unicast_group_create (*C function*), 1782
bt_bap_unicast_group_delete (*C function*), 1783
bt_bap_unicast_group_param (*C struct*), 1784
bt_bap_unicast_group_param.c_to_p_ft (*C var*), 1784
bt_bap_unicast_group_param.iso_interval (*C var*), 1785
bt_bap_unicast_group_param.p_to_c_ft (*C var*), 1785
bt_bap_unicast_group_param.packing (*C var*), 1784
bt_bap_unicast_group_param.params (*C var*), 1784
bt_bap_unicast_group_param.params_count (*C var*), 1784
bt_bap_unicast_group_stream_pair_param (*C struct*), 1784
bt_bap_unicast_group_stream_pair_param.rx_param (*C var*), 1784
bt_bap_unicast_group_stream_pair_param.tx_param (*C var*), 1784
bt_bap_unicast_group_stream_param (*C struct*), 1783
bt_bap_unicast_group_stream_param.qos (*C var*), 1784
bt_bap_unicast_group_stream_param.stream (*C var*), 1784
bt_bap_unicast_server_cb (*C struct*), 1789
bt_bap_unicast_server_cb.config (*C var*), 1789
bt_bap_unicast_server_cb.disable (*C var*), 1791
bt_bap_unicast_server_cb.enable (*C var*), 1790
bt_bap_unicast_server_cb.metadata (*C var*), 1791
bt_bap_unicast_server_cb.qos (*C var*), 1790
bt_bap_unicast_server_cb.reconfig (*C var*), 1790
bt_bap_unicast_server_cb.release (*C var*), 1792
bt_bap_unicast_server_cb.start (*C var*), 1791
bt_bap_unicast_server_cb.stop (*C var*), 1791
bt_bap_unicast_server_config_ase (*C function*), 1789
bt_bap_unicast_server_foreach_ep (*C function*), 1789
bt_bap_unicast_server_register_cb (*C function*), 1788
bt_bap_unicast_server_unregister_cb (*C function*), 1789
bt_bas_get_battery_level (*C function*), 1935
bt_bas_set_battery_level (*C function*), 1935
bt_bond_info (*C struct*), 1999

bt_bond_info.addr (*C var*), 2000
BT_BR_CONN_PARAM (*C macro*), 2312
bt_br_conn_param (*C struct*), 2343
BT_BR_CONN_PARAM_DEFAULT (*C macro*), 2312
BT_BR_CONN_PARAM_INIT (*C macro*), 2312
bt_br_discovery_cb_t (*C type*), 1962
bt_br_discovery_param (*C struct*), 1999
bt_br_discovery_param.length (*C var*), 1999
bt_br_discovery_param.limited (*C var*), 1999
bt_br_discovery_result (*C struct*), 1999
bt_br_discovery_result.addr (*C var*), 1999
bt_br_discovery_result.cod (*C var*), 1999
bt_br_discovery_result.eir (*C var*), 1999
bt_br_discovery_result.rssi (*C var*), 1999
bt_br_discovery_start (*C function*), 1985
bt_br_discovery_stop (*C function*), 1985
bt_br_oob (*C struct*), 1999
bt_br_oob_get_local (*C function*), 1986
bt_br_oob.addr (*C var*), 1999
bt_br_set_connectable (*C function*), 1986
bt_br_set_discoverable (*C function*), 1986
BT_BUF_ACL_RX_SIZE (*C macro*), 2343
BT_BUF_ACL_SIZE (*C macro*), 2343
BT_BUF_CMD_SIZE (*C macro*), 2343
BT_BUF_CMD_TX_SIZE (*C macro*), 2344
bt_buf_data (*C struct*), 2346
BT_BUF_EVT_RX_SIZE (*C macro*), 2343
BT_BUF_EVT_SIZE (*C macro*), 2343
bt_buf_get_evt (*C function*), 2345
bt_buf_get_rx (*C function*), 2344
bt_buf_get_tx (*C function*), 2345
bt_buf_get_type (*C function*), 2345
BT_BUF_ISO_RX_COUNT (*C macro*), 2344
BT_BUF_ISO_RX_SIZE (*C macro*), 2344
BT_BUF_ISO_SIZE (*C macro*), 2343
BT_BUF_RESERVE (*C macro*), 2343
BT_BUF_RX_COUNT (*C macro*), 2344
BT_BUF_RX_SIZE (*C macro*), 2344
bt_buf_set_type (*C function*), 2345
BT_BUF_SIZE (*C macro*), 2343
bt_buf_type (*C enum*), 2344
bt_buf_type.BT_BUF_ACL_IN (*C enumerator*), 2344
bt_buf_type.BT_BUF_ACL_OUT (*C enumerator*), 2344
bt_buf_type.BT_BUF_CMD (*C enumerator*), 2344
bt_buf_type.BT_BUF_EVT (*C enumerator*), 2344
bt_buf_type.BT_BUF_H4 (*C enumerator*), 2344
bt_buf_type.BT_BUF_ISO_IN (*C enumerator*), 2344
bt_buf_type.BT_BUF_ISO_OUT (*C enumerator*), 2344
bt_cap_acceptor_register (*C function*), 1803
bt_cap_broadcast_to_unicast_param (*C struct*), 1818
bt_cap_broadcast_to_unicast_param.broadcast_source (*C var*), 1818
bt_cap_broadcast_to_unicast_param.count (*C var*), 1818
bt_cap_broadcast_to_unicast_param.members (*C var*), 1818
bt_cap_broadcast_to_unicast_param.type (*C var*), 1818
bt_cap_commander_broadcast_reception_start (*C function*), 1811
bt_cap_commander_broadcast_reception_start_member_param (*C struct*), 1820
bt_cap_commander_broadcast_reception_start_member_param.addr (*C var*), 1820
bt_cap_commander_broadcast_reception_start_member_param.adv_sid (*C var*), 1820

bt_cap_commander_broadcast_reception_start_member_param.broadcast_id (*C var*), 1820
bt_cap_commander_broadcast_reception_start_member_param.member (*C var*), 1820
bt_cap_commander_broadcast_reception_start_member_param.num_subgroups (*C var*), 1820
bt_cap_commander_broadcast_reception_start_member_param.pa_interval (*C var*), 1820
bt_cap_commander_broadcast_reception_start_member_param.subgroups (*C var*), 1820
bt_cap_commander_broadcast_reception_start_param (*C struct*), 1820
bt_cap_commander_broadcast_reception_start_param.count (*C var*), 1821
bt_cap_commander_broadcast_reception_start_param.param (*C var*), 1820
bt_cap_commander_broadcast_reception_start_param.type (*C var*), 1820
bt_cap_commander_broadcast_reception_stop (*C function*), 1811
bt_cap_commander_broadcast_reception_stop_param (*C struct*), 1821
bt_cap_commander_broadcast_reception_stop_param.count (*C var*), 1821
bt_cap_commander_broadcast_reception_stop_param.members (*C var*), 1821
bt_cap_commander_broadcast_reception_stop_param.type (*C var*), 1821
bt_cap_commander_cancel (*C function*), 1811
bt_cap_commander_cb (*C struct*), 1818
bt_cap_commander_cb.broadcast_reception_start (*C var*), 1820
bt_cap_commander_cb.discovery_complete (*C var*), 1818
bt_cap_commander_cb.microphone_gain_changed (*C var*), 1819
bt_cap_commander_cb.microphone_mute_changed (*C var*), 1819
bt_cap_commander_cb.volume_changed (*C var*), 1819
bt_cap_commander_cb.volume_mute_changed (*C var*), 1819
bt_cap_commander_cb.volume_offset_changed (*C var*), 1819
bt_cap_commander_change_microphone_gain_setting (*C function*), 1812
bt_cap_commander_change_microphone_gain_setting_member_param (*C struct*), 1823
bt_cap_commander_change_microphone_gain_setting_member_param.gain (*C var*), 1823
bt_cap_commander_change_microphone_gain_setting_member_param.member (*C var*), 1823
bt_cap_commander_change_microphone_gain_setting_param (*C struct*), 1823
bt_cap_commander_change_microphone_gain_setting_param.count (*C var*), 1823
bt_cap_commander_change_microphone_gain_setting_param.param (*C var*), 1823
bt_cap_commander_change_microphone_gain_setting_param.type (*C var*), 1823
bt_cap_commander_change_microphone_mute_state (*C function*), 1812
bt_cap_commander_change_microphone_mute_state_param (*C struct*), 1822
bt_cap_commander_change_microphone_mute_state_param.count (*C var*), 1823
bt_cap_commander_change_microphone_mute_state_param.members (*C var*), 1822
bt_cap_commander_change_microphone_mute_state_param.mute (*C var*), 1823
bt_cap_commander_change_microphone_mute_state_param.type (*C var*), 1822
bt_cap_commander_change_volume (*C function*), 1811
bt_cap_commander_change_volume_mute_state (*C function*), 1812
bt_cap_commander_change_volume_mute_state_param (*C struct*), 1822
bt_cap_commander_change_volume_mute_state_param.count (*C var*), 1822
bt_cap_commander_change_volume_mute_state_param.members (*C var*), 1822
bt_cap_commander_change_volume_mute_state_param.mute (*C var*), 1822
bt_cap_commander_change_volume_mute_state_param.type (*C var*), 1822
bt_cap_commander_change_volume_offset (*C function*), 1812
bt_cap_commander_change_volume_offset_member_param (*C struct*), 1821
bt_cap_commander_change_volume_offset_member_param.member (*C var*), 1821
bt_cap_commander_change_volume_offset_member_param.offset (*C var*), 1821
bt_cap_commander_change_volume_offset_param (*C struct*), 1822
bt_cap_commander_change_volume_offset_param.count (*C var*), 1822
bt_cap_commander_change_volume_offset_param.param (*C var*), 1822
bt_cap_commander_change_volume_offset_param.type (*C var*), 1822
bt_cap_commander_change_volume_param (*C struct*), 1821
bt_cap_commander_change_volume_param.count (*C var*), 1821
bt_cap_commander_change_volume_param.members (*C var*), 1821
bt_cap_commander_change_volume_param.type (*C var*), 1821
bt_cap_commander_change_volume_param.volume (*C var*), 1821
bt_cap_commander_discover (*C function*), 1810

bt_cap_commander_register_cb (*C function*), 1810
bt_cap_commander_unregister_cb (*C function*), 1810
bt_cap_initiator_broadcast_audio_create (*C function*), 1807
bt_cap_initiator_broadcast_audio_delete (*C function*), 1808
bt_cap_initiator_broadcast_audio_start (*C function*), 1807
bt_cap_initiator_broadcast_audio_stop (*C function*), 1808
bt_cap_initiator_broadcast_audio_update (*C function*), 1807
bt_cap_initiator_broadcast_create_param (*C struct*), 1816
bt_cap_initiator_broadcast_create_param.broadcast_code (*C var*), 1817
bt_cap_initiator_broadcast_create_param.encryption (*C var*), 1817
bt_cap_initiator_broadcast_create_param.irc (*C var*), 1817
bt_cap_initiator_broadcast_create_param.iso_interval (*C var*), 1817
bt_cap_initiator_broadcast_create_param.packing (*C var*), 1817
bt_cap_initiator_broadcast_create_param.pto (*C var*), 1817
bt_cap_initiator_broadcast_create_param.qos (*C var*), 1817
bt_cap_initiator_broadcast_create_param.subgroup_count (*C var*), 1816
bt_cap_initiator_broadcast_create_param.subgroup_params (*C var*), 1817
bt_cap_initiator_broadcast_get_base (*C function*), 1809
bt_cap_initiator_broadcast_get_id (*C function*), 1808
bt_cap_initiator_broadcast_stream_param (*C struct*), 1816
bt_cap_initiator_broadcast_stream_param.data (*C var*), 1816
bt_cap_initiator_broadcast_stream_param.data_len (*C var*), 1816
bt_cap_initiator_broadcast_stream_param.stream (*C var*), 1816
bt_cap_initiator_broadcast_subgroup_param (*C struct*), 1816
bt_cap_initiator_broadcast_subgroup_param.codec_cfg (*C var*), 1816
bt_cap_initiator_broadcast_subgroup_param.stream_count (*C var*), 1816
bt_cap_initiator_broadcast_subgroup_param.stream_params (*C var*), 1816
bt_cap_initiator_broadcast_to_unicast (*C function*), 1809
bt_cap_initiator_cb (*C struct*), 1812
bt_cap_initiator_cb.unicast_discovery_complete (*C var*), 1812
bt_cap_initiator_cb.unicast_start_complete (*C var*), 1813
bt_cap_initiator_cb.unicast_stop_complete (*C var*), 1813
bt_cap_initiator_cb.unicast_update_complete (*C var*), 1813
bt_cap_initiator_register_cb (*C function*), 1805
bt_cap_initiator_unicast_audio_cancel (*C function*), 1806
bt_cap_initiator_unicast_audio_start (*C function*), 1805
bt_cap_initiator_unicast_audio_stop (*C function*), 1806
bt_cap_initiator_unicast_audio_update (*C function*), 1806
bt_cap_initiator_unicast_discover (*C function*), 1803
bt_cap_initiator_unicast_to_broadcast (*C function*), 1809
bt_cap_initiator_unregister_cb (*C function*), 1805
bt_cap_set_member (*C union*), 1813
bt_cap_set_member.csip (*C var*), 1813
bt_cap_set_member.member (*C var*), 1813
bt_cap_set_type (*C enum*), 1803
bt_cap_set_type.BT_CAP_SET_TYPE_AD_HOC (*C enumerator*), 1803
bt_cap_set_type.BT_CAP_SET_TYPE_CSIP (*C enumerator*), 1803
bt_cap_stream (*C struct*), 1814
bt_cap_stream_get_tx_sync (*C function*), 1805
bt_cap_stream_ops_register (*C function*), 1803
bt_cap_stream_send (*C function*), 1804
bt_cap_stream_send_ts (*C function*), 1804
bt_cap_stream.bap_stream (*C var*), 1814
bt_cap_stream.ops (*C var*), 1814
bt_cap_unicast_audio_start_param (*C struct*), 1814
bt_cap_unicast_audio_start_param.count (*C var*), 1815
bt_cap_unicast_audio_start_param.stream_params (*C var*), 1815
bt_cap_unicast_audio_start_param.type (*C var*), 1814

bt_cap_unicast_audio_start_stream_param (*C struct*), 1814
bt_cap_unicast_audio_start_stream_param.codec_cfg (*C var*), 1814
bt_cap_unicast_audio_start_stream_param.ep (*C var*), 1814
bt_cap_unicast_audio_start_stream_param.member (*C var*), 1814
bt_cap_unicast_audio_start_stream_param.stream (*C var*), 1814
bt_cap_unicast_audio_stop_param (*C struct*), 1815
bt_cap_unicast_audio_stop_param.count (*C var*), 1816
bt_cap_unicast_audio_stop_param.streams (*C var*), 1816
bt_cap_unicast_audio_stop_param.type (*C var*), 1815
bt_cap_unicast_audio_update_param (*C struct*), 1815
bt_cap_unicast_audio_update_param.count (*C var*), 1815
bt_cap_unicast_audio_update_param.stream_params (*C var*), 1815
bt_cap_unicast_audio_update_param.type (*C var*), 1815
bt_cap_unicast_audio_update_stream_param (*C struct*), 1815
bt_cap_unicast_audio_update_stream_param.meta (*C var*), 1815
bt_cap_unicast_audio_update_stream_param.meta_len (*C var*), 1815
bt_cap_unicast_audio_update_stream_param.stream (*C var*), 1815
bt_cap_unicast_to_broadcast_param (*C struct*), 1817
bt_cap_unicast_to_broadcast_param.broadcast_code (*C var*), 1818
bt_cap_unicast_to_broadcast_param.encrypt (*C var*), 1818
bt_cap_unicast_to_broadcast_param.unicast_group (*C var*), 1818
bt_ccm_decrypt (*C function*), 2353
bt_ccm_encrypt (*C function*), 2354
BT_COMP_ID_LF (*C macro*), 2005
bt_configure_data_path (*C function*), 1986
bt_conn_auth_cancel (*C function*), 2328
bt_conn_auth_cb (*C struct*), 2340
bt_conn_auth_cb_overlay (*C function*), 2327
bt_conn_auth_cb_register (*C function*), 2326
bt_conn_auth_cb.cancel (*C var*), 2341
bt_conn_auth_cb.oob_data_request (*C var*), 2341
bt_conn_auth_cb.pairing_accept (*C var*), 2340
bt_conn_auth_cb.pairing_confirm (*C var*), 2342
bt_conn_auth_cb.passkey_confirm (*C var*), 2341
bt_conn_auth_cb.passkey_display (*C var*), 2340
bt_conn_auth_cb.passkey_entry (*C var*), 2341
bt_conn_auth_cb.pincode_entry (*C var*), 2342
bt_conn_auth_info_cb (*C struct*), 2342
bt_conn_auth_info_cb_register (*C function*), 2327
bt_conn_auth_info_cb_unregister (*C function*), 2327
bt_conn_auth_info_cb.bond_deleted (*C var*), 2343
bt_conn_auth_info_cb.node (*C var*), 2343
bt_conn_auth_info_cb.pairing_complete (*C var*), 2342
bt_conn_auth_info_cb.pairing_failed (*C var*), 2342
bt_conn_auth_keypress (*C enum*), 2315
bt_conn_auth_keypress_notify (*C function*), 2328
bt_conn_auth_keypress.BT_CONN_AUTH_KEYPRESS_CLEARED (*C enumerator*), 2315
bt_conn_auth_keypress.BT_CONN_AUTH_KEYPRESS_DIGIT_ENTERED (*C enumerator*), 2315
bt_conn_auth_keypress.BT_CONN_AUTH_KEYPRESS_DIGIT_ERASED (*C enumerator*), 2315
bt_conn_auth_keypress.BT_CONN_AUTH_KEYPRESS_ENTRY_COMPLETED (*C enumerator*), 2315
bt_conn_auth_keypress.BT_CONN_AUTH_KEYPRESS_ENTRY_STARTED (*C enumerator*), 2315
bt_conn_auth_pairing_confirm (*C function*), 2328
bt_conn_auth_passkey_confirm (*C function*), 2328
bt_conn_auth_passkey_entry (*C function*), 2327
bt_conn_auth_pincode_entry (*C function*), 2329
bt_conn_br_info (*C struct*), 2331
bt_conn_br_info.dst (*C var*), 2331
bt_conn_br_remote_info (*C struct*), 2332

bt_conn_br_remote_info.features (*C var*), 2332
bt_conn_br_remote_info.num_pages (*C var*), 2332
bt_conn_cb (*C struct*), 2336
BT_CONN_CB_DEFINE (*C macro*), 2312
bt_conn_cb_register (*C function*), 2324
bt_conn_cb_unregister (*C function*), 2324
bt_conn_cb.connected (*C var*), 2336
bt_conn_cb.disconnected (*C var*), 2336
bt_conn_cb.identity_resolved (*C var*), 2337
bt_conn_cb.le_data_len_updated (*C var*), 2338
bt_conn_cb.le_param_req (*C var*), 2337
bt_conn_cb.le_param_updated (*C var*), 2337
bt_conn_cb.le_phy_updated (*C var*), 2338
bt_conn_cb.recycled (*C var*), 2337
bt_conn_cb.remote_info_available (*C var*), 2338
bt_conn_cb.security_changed (*C var*), 2338
bt_conn_create_auto_stop (*C function*), 2322
bt_conn_create_br (*C function*), 2329
bt_conn_disconnect (*C function*), 2321
bt_conn_enc_key_size (*C function*), 2324
bt_conn_foreach (*C function*), 2317
bt_conn_get_dst (*C function*), 2317
bt_conn_get_info (*C function*), 2318
bt_conn_get_remote_info (*C function*), 2318
bt_conn_get_security (*C function*), 2324
bt_conn_index (*C function*), 2318
bt_conn_info (*C struct*), 2331
bt_conn_info.br (*C var*), 2332
bt_conn_info.id (*C var*), 2331
bt_conn_info.le (*C var*), 2332
bt_conn_info.role (*C var*), 2331
bt_conn_info.security (*C var*), 2332
bt_conn_info.state (*C var*), 2332
bt_conn_info.type (*C var*), 2331
BT_CONN_INTERVAL_TO_MS (*C macro*), 2311
BT_CONN_INTERVAL_TO_US (*C macro*), 2311
bt_conn_le_create (*C function*), 2321
bt_conn_le_create_auto (*C function*), 2322
BT_CONN_LE_CREATE_CONN (*C macro*), 2312
BT_CONN_LE_CREATE_CONN_AUTO (*C macro*), 2312
BT_CONN_LE_CREATE_PARAM (*C macro*), 2311
bt_conn_le_create_param (*C struct*), 2335
BT_CONN_LE_CREATE_PARAM_INIT (*C macro*), 2311
bt_conn_le_create_param.interval (*C var*), 2335
bt_conn_le_create_param.interval_coded (*C var*), 2335
bt_conn_le_create_param.options (*C var*), 2335
bt_conn_le_create_param.timeout (*C var*), 2335
bt_conn_le_create_param.window (*C var*), 2335
bt_conn_le_create_param.window_coded (*C var*), 2335
bt_conn_le_create_synced (*C function*), 2322
bt_conn_le_create_synced_param (*C struct*), 2335
bt_conn_le_create_synced_param.peer (*C var*), 2335
bt_conn_le_create_synced_param.subevent (*C var*), 2336
bt_conn_le_data_len_info (*C struct*), 2330
bt_conn_le_data_len_info.rx_max_len (*C var*), 2330
bt_conn_le_data_len_info.rx_max_time (*C var*), 2330
bt_conn_le_data_len_info.tx_max_len (*C var*), 2330
bt_conn_le_data_len_info.tx_max_time (*C var*), 2330

BT_CONN_LE_DATA_LEN_PARAM (*C macro*), 2311
bt_conn_le_data_len_param (*C struct*), 2330
BT_CONN_LE_DATA_LEN_PARAM_INIT (*C macro*), 2311
bt_conn_le_data_len_param.tx_max_len (*C var*), 2330
bt_conn_le_data_len_param.tx_max_time (*C var*), 2330
bt_conn_le_data_len_update (*C function*), 2320
bt_conn_le_enhanced_get_tx_power_level (*C function*), 2319
bt_conn_le_get_remote_tx_power_level (*C function*), 2319
bt_conn_le_get_tx_power_level (*C function*), 2318
bt_conn_le_info (*C struct*), 2330
bt_conn_le_info.dst (*C var*), 2330
bt_conn_le_info.interval (*C var*), 2331
bt_conn_le_info.latency (*C var*), 2331
bt_conn_le_info.local (*C var*), 2330
bt_conn_le_info.remote (*C var*), 2330
bt_conn_le_info.src (*C var*), 2330
bt_conn_le_info.timeout (*C var*), 2331
bt_conn_le_param_update (*C function*), 2320
bt_conn_le_path_loss_reporting_param (*C struct*), 2334
bt_conn_le_path_loss_reporting_param.high_hysteresis (*C var*), 2334
bt_conn_le_path_loss_reporting_param.high_threshold (*C var*), 2334
bt_conn_le_path_loss_reporting_param.low_hysteresis (*C var*), 2335
bt_conn_le_path_loss_reporting_param.low_threshold (*C var*), 2334
bt_conn_le_path_loss_reporting_param.min_time_spent (*C var*), 2335
bt_conn_le_path_loss_threshold_report (*C struct*), 2334
bt_conn_le_path_loss_threshold_report.path_loss (*C var*), 2334
bt_conn_le_path_loss_threshold_report.zone (*C var*), 2334
bt_conn_le_path_loss_zone (*C enum*), 2314
bt_conn_le_path_loss_zone.BT_CONN_LE_PATH_LOSS_ZONE_ENTERED_HIGH (*C enumerator*), 2315
bt_conn_le_path_loss_zone.BT_CONN_LE_PATH_LOSS_ZONE_ENTERED_LOW (*C enumerator*), 2315
bt_conn_le_path_loss_zone.BT_CONN_LE_PATH_LOSS_ZONE_ENTERED_MIDDLE (*C enumerator*), 2315
bt_conn_le_path_loss_zone.BT_CONN_LE_PATH_LOSS_ZONE_UNAVAILABLE (*C enumerator*), 2315
bt_conn_le_phy_info (*C struct*), 2329
bt_conn_le_phy_info.rx_phy (*C var*), 2329
BT_CONN_LE_PHY_PARAM (*C macro*), 2310
bt_conn_le_phy_param (*C struct*), 2329
BT_CONN_LE_PHY_PARAM_1M (*C macro*), 2310
BT_CONN_LE_PHY_PARAM_2M (*C macro*), 2311
BT_CONN_LE_PHY_PARAM_ALL (*C macro*), 2311
BT_CONN_LE_PHY_PARAM_CODED (*C macro*), 2311
BT_CONN_LE_PHY_PARAM_INIT (*C macro*), 2310
bt_conn_le_phy_param.options (*C var*), 2329
bt_conn_le_phy_param.pref_rx_phy (*C var*), 2329
bt_conn_le_phy_param.pref_tx_phy (*C var*), 2329
bt_conn_le_phy_update (*C function*), 2321
bt_conn_le_remote_info (*C struct*), 2332
bt_conn_le_remote_info.features (*C var*), 2332
bt_conn_le_set_path_loss_mon_enable (*C function*), 2320
bt_conn_le_set_path_loss_mon_param (*C function*), 2319
bt_conn_le_set_tx_power_report_enable (*C function*), 2319
bt_conn_le_tx_power (*C struct*), 2333
bt_conn_le_tx_power_phy (*C enum*), 2314
bt_conn_le_tx_power_phy.BT_CONN_LE_TX_POWER_PHY_1M (*C enumerator*), 2314
bt_conn_le_tx_power_phy.BT_CONN_LE_TX_POWER_PHY_2M (*C enumerator*), 2314
bt_conn_le_tx_power_phy.BT_CONN_LE_TX_POWER_PHY_CODED_S2 (*C enumerator*), 2314
bt_conn_le_tx_power_phy.BT_CONN_LE_TX_POWER_PHY_CODED_S8 (*C enumerator*), 2314
bt_conn_le_tx_power_phy.BT_CONN_LE_TX_POWER_PHY_NONE (*C enumerator*), 2314

bt_conn_le_tx_power_report (*C struct*), 2333
bt_conn_le_tx_power_report.delta (*C var*), 2334
bt_conn_le_tx_power_report.phy (*C var*), 2333
bt_conn_le_tx_power_report.reason (*C var*), 2333
bt_conn_le_tx_power_report.tx_power_level (*C var*), 2333
bt_conn_le_tx_power_report.tx_power_level_flag (*C var*), 2334
bt_conn_le_tx_power.current_level (*C var*), 2333
bt_conn_le_tx_power.max_level (*C var*), 2333
bt_conn_le_tx_power.phy (*C var*), 2333
bt_conn_lookup_addr_le (*C function*), 2317
bt_conn_oob_info (*C struct*), 2338
bt_conn_oob_info.lesc (*C var*), 2339
bt_conn_oob_info.oob_config (*C var*), 2339
bt_conn_oob_info.PhonyNameDueToError.BT_CONN_OOB_LE_LEGACY (*C enumerator*), 2339
bt_conn_oob_info.PhonyNameDueToError.BT_CONN_OOB_LE_SC (*C enumerator*), 2339
bt_conn_oob_info.type (*C var*), 2339
bt_conn_pairing_feat (*C struct*), 2339
bt_conn_pairing_feat.auth_req (*C var*), 2339
bt_conn_pairing_feat.init_key_dist (*C var*), 2340
bt_conn_pairing_feat.io_capability (*C var*), 2339
bt_conn_pairing_feat.max_enc_key_size (*C var*), 2339
bt_conn_pairing_feat.oob_data_flag (*C var*), 2339
bt_conn_pairing_feat.resp_key_dist (*C var*), 2340
bt_conn_ref (*C function*), 2316
bt_conn_remote_info (*C struct*), 2332
bt_conn_remote_info.br (*C var*), 2333
bt_conn_remote_info.le (*C var*), 2333
bt_conn_remote_info.manufacturer (*C var*), 2333
bt_conn_remote_info.subversion (*C var*), 2333
bt_conn_remote_info.type (*C var*), 2333
bt_conn_remote_info.version (*C var*), 2333
bt_conn_set_bondable (*C function*), 2325
bt_conn_set_security (*C function*), 2323
bt_conn_state (*C enum*), 2313
bt_conn_state.BT_CONN_STATE_CONNECTED (*C enumerator*), 2313
bt_conn_state.BT_CONN_STATE_CONNECTING (*C enumerator*), 2313
bt_conn_state.BT_CONN_STATE_DISCONNECTED (*C enumerator*), 2313
bt_conn_state.BT_CONN_STATE_DISCONNECTING (*C enumerator*), 2313
bt_conn_type (*C enum*), 2313
bt_conn_type.BT_CONN_TYPE_ALL (*C enumerator*), 2313
bt_conn_type.BT_CONN_TYPE_BR (*C enumerator*), 2313
bt_conn_type.BT_CONN_TYPE_ISO (*C enumerator*), 2313
bt_conn_type.BT_CONN_TYPE_LE (*C enumerator*), 2313
bt_conn_type.BT_CONN_TYPE_SCO (*C enumerator*), 2313
bt_conn_unref (*C function*), 2317
BT_CSIP_DATA_RSI (*C macro*), 1825
BT_CSIP_ERROR_LOCK_ALREADY_GRANTED (*C macro*), 1825
BT_CSIP_ERROR_LOCK_DENIED (*C macro*), 1824
BT_CSIP_ERROR_LOCK_INVAL_VALUE (*C macro*), 1824
BT_CSIP_ERROR_LOCK_RELEASE_DENIED (*C macro*), 1824
BT_CSIP_ERROR_SIRK_OOB_ONLY (*C macro*), 1824
BT_CSIP_READ_SIRK_REQ_RSP_ACCEPT (*C macro*), 1824
BT_CSIP_READ_SIRK_REQ_RSP_ACCEPT_ENC (*C macro*), 1824
BT_CSIP_READ_SIRK_REQ_RSP_OOB_ONLY (*C macro*), 1824
BT_CSIP_READ_SIRK_REQ_RSP_REJECT (*C macro*), 1824
BT_CSIP_RSI_SIZE (*C macro*), 1824
bt_csip_set_coordinator_cb (*C struct*), 1834
bt_csip_set_coordinator_cb.discover (*C var*), 1834

bt_csip_set_coordinator_cb.lock_changed (*C var*), 1834
bt_csip_set_coordinator_cb.lock_set (*C var*), 1834
bt_csip_set_coordinator_cb.ordered_access (*C var*), 1834
bt_csip_set_coordinator_cb.release_set (*C var*), 1834
bt_csip_set_coordinator_cb.sirk_changed (*C var*), 1834
bt_csip_set_coordinator_csis_inst (*C struct*), 1833
bt_csip_set_coordinator_csis_inst.info (*C var*), 1834
bt_csip_set_coordinator_csis_inst.svc_inst (*C var*), 1834
bt_csip_set_coordinator_discover (*C function*), 1828
bt_csip_set_coordinator_discover_cb (*C type*), 1825
BT_CSIP_SET_COORDINATOR_DISCOVER_TIMER_VALUE (*C macro*), 1824
bt_csip_set_coordinator_is_set_member (*C function*), 1828
bt_csip_set_coordinator_lock (*C function*), 1829
bt_csip_set_coordinator_lock_changed_cb (*C type*), 1825
bt_csip_set_coordinator_lock_set_cb (*C type*), 1825
BT_CSIP_SET_COORDINATOR_MAX_CSIS_INSTANCES (*C macro*), 1824
bt_csip_set_coordinator_ordered_access (*C function*), 1829
bt_csip_set_coordinator_ordered_access_cb_t (*C type*), 1826
bt_csip_set_coordinator_ordered_access_t (*C type*), 1826
bt_csip_set_coordinator_register_cb (*C function*), 1829
bt_csip_set_coordinator_release (*C function*), 1829
bt_csip_set_coordinator_set_info (*C struct*), 1833
bt_csip_set_coordinator_set_info.lockable (*C var*), 1833
bt_csip_set_coordinator_set_info.rank (*C var*), 1833
bt_csip_set_coordinator_set_info.set_size (*C var*), 1833
bt_csip_set_coordinator_set_info.sirk (*C var*), 1833
bt_csip_set_coordinator_set_member (*C struct*), 1834
bt_csip_set_coordinator_set_member_by_conn (*C function*), 1828
bt_csip_set_coordinator_set_member.insts (*C var*), 1834
bt_csip_set_coordinator_sirk_changed_cb (*C type*), 1825
bt_csip_set_member_cb (*C struct*), 1832
bt_csip_set_member_cb.lock_changed (*C var*), 1832
bt_csip_set_member_cb.sirk_read_req (*C var*), 1832
bt_csip_set_member_generate_rsi (*C function*), 1827
bt_csip_set_member_get_sirk (*C function*), 1827
bt_csip_set_member_lock (*C function*), 1828
bt_csip_set_member_register (*C function*), 1827
bt_csip_set_member_register_param (*C struct*), 1832
bt_csip_set_member_register_param.cb (*C var*), 1833
bt_csip_set_member_register_param.lockable (*C var*), 1833
bt_csip_set_member_register_param.parent (*C var*), 1833
bt_csip_set_member_register_param.rank (*C var*), 1833
bt_csip_set_member_register_param.set_size (*C var*), 1832
bt_csip_set_member_register_param.sirk (*C var*), 1832
bt_csip_set_member_sirk (*C function*), 1827
bt_csip_set_member_svc_decl_get (*C function*), 1826
bt_csip_set_member_unregister (*C function*), 1827
BT_CSIP_SIRK_SIZE (*C macro*), 1824
bt_ctlr_set_public_addr (*C function*), 2354
BT_DATA (*C macro*), 1958
bt_data (*C struct*), 1989
BT_DATA_3D_INFO (*C macro*), 2007
BT_DATA_ADV_INT (*C macro*), 2006
BT_DATA_ADV_INT_LONG (*C macro*), 2007
BT_DATA_BIG_INFO (*C macro*), 2007
BT_DATA_BROADCAST_CODE (*C macro*), 2007
BT_DATA_BROADCAST_NAME (*C macro*), 2007
BT_DATA_BYTES (*C macro*), 1958

BT_DATA_CHANNEL_MAP_UPDATE_IND (*C macro*), 2007
BT_DATA_CSIS_RSI (*C macro*), 2007
BT_DATA_ENCRYPTED_AD_DATA (*C macro*), 2007
BT_DATA_FLAGS (*C macro*), 2005
BT_DATA_GAP_APPEARANCE (*C macro*), 2006
bt_data_get_len (*C function*), 1972
BT_DATA_INDOOR_POS (*C macro*), 2007
BT_DATA_LE_BT_DEVICE_ADDRESS (*C macro*), 2006
BT_DATA_LE_ROLE (*C macro*), 2006
BT_DATA_LE_SC_CONFIRM_VALUE (*C macro*), 2006
BT_DATA_LE_SC_RANDOM_VALUE (*C macro*), 2006
BT_DATA_LE_SUPPORTED_FEATURES (*C macro*), 2007
BT_DATA_MANUFACTURER_DATA (*C macro*), 2007
BT_DATA_MESH_BEACON (*C macro*), 2007
BT_DATA_MESH_MESSAGE (*C macro*), 2007
BT_DATA_MESH_PROV (*C macro*), 2007
BT_DATA_NAME_COMPLETE (*C macro*), 2005
BT_DATA_NAME_SHORTENED (*C macro*), 2005
bt_data_parse (*C function*), 1984
BT_DATA_PERIPHERAL_INT_RANGE (*C macro*), 2005
BT_DATA_PUB_TARGET_ADDR (*C macro*), 2006
BT_DATA_RAND_TARGET_ADDR (*C macro*), 2006
bt_data_serialize (*C function*), 1973
BT_DATA_SERIALIZED_SIZE (*C macro*), 1957
BT_DATA_SIMPLE_PAIRING_HASH (*C macro*), 2006
BT_DATA_SIMPLE_PAIRING_RAND (*C macro*), 2006
BT_DATA_SM_OOB_FLAGS (*C macro*), 2005
BT_DATA_SM_TK_VALUE (*C macro*), 2005
BT_DATA_SOLICIT16 (*C macro*), 2006
BT_DATA_SOLICIT32 (*C macro*), 2006
BT_DATA_SOLICIT128 (*C macro*), 2006
BT_DATA_SVC_DATA16 (*C macro*), 2006
BT_DATA_SVC_DATA32 (*C macro*), 2006
BT_DATA_SVC_DATA128 (*C macro*), 2006
BT_DATA_TRANS_DISCOVER_DATA (*C macro*), 2007
BT_DATA_TX_POWER (*C macro*), 2005
BT_DATA_URI (*C macro*), 2007
BT_DATA_UUID16_ALL (*C macro*), 2005
BT_DATA_UUID16_SOME (*C macro*), 2005
BT_DATA_UUID32_ALL (*C macro*), 2005
BT_DATA_UUID32_SOME (*C macro*), 2005
BT_DATA_UUID128_ALL (*C macro*), 2005
BT_DATA_UUID128_SOME (*C macro*), 2005
bt_disable (*C function*), 1969
bt_eatt_connect (*C function*), 2073
bt_eatt_count (*C function*), 2073
bt_enable (*C function*), 1969
bt_enable_raw (*C function*), 2351
bt_encrypt_be (*C function*), 2353
bt_encrypt_le (*C function*), 2353
bt_foreach_bond (*C function*), 1986
BT_GAP_ADV_FAST_INT_MAX_1 (*C macro*), 2027
BT_GAP_ADV_FAST_INT_MAX_2 (*C macro*), 2027
BT_GAP_ADV_FAST_INT_MIN_1 (*C macro*), 2027
BT_GAP_ADV_FAST_INT_MIN_2 (*C macro*), 2027
BT_GAP_ADV_HIGH_DUTY_CYCLE_MAX_TIMEOUT (*C macro*), 2028
BT_GAP_ADV_MAX_ADV_DATA_LEN (*C macro*), 2028
BT_GAP_ADV_MAX_EXT_ADV_DATA_LEN (*C macro*), 2028

BT_GAP_ADV_SLOW_INT_MAX (*C macro*), 2027
BT_GAP_ADV_SLOW_INT_MIN (*C macro*), 2027
BT_GAP_DATA_LEN_DEFAULT (*C macro*), 2028
BT_GAP_DATA_LEN_MAX (*C macro*), 2028
BT_GAP_DATA_TIME_DEFAULT (*C macro*), 2028
BT_GAP_DATA_TIME_MAX (*C macro*), 2028
BT_GAP_INIT_CONN_INT_MAX (*C macro*), 2028
BT_GAP_INIT_CONN_INT_MIN (*C macro*), 2027
BT_GAP_NO_TIMEOUT (*C macro*), 2028
BT_GAP_PER_ADV_FAST_INT_MAX_1 (*C macro*), 2027
BT_GAP_PER_ADV_FAST_INT_MAX_2 (*C macro*), 2027
BT_GAP_PER_ADV_FAST_INT_MIN_1 (*C macro*), 2027
BT_GAP_PER_ADV_FAST_INT_MIN_2 (*C macro*), 2027
BT_GAP_PER_ADV_INTERVAL_TO_MS (*C macro*), 2029
BT_GAP_PER_ADV_MAX_INTERVAL (*C macro*), 2029
BT_GAP_PER_ADV_MAX_SKIP (*C macro*), 2028
BT_GAP_PER_ADV_MAX_TIMEOUT (*C macro*), 2029
BT_GAP_PER_ADV_MIN_INTERVAL (*C macro*), 2029
BT_GAP_PER_ADV_MIN_TIMEOUT (*C macro*), 2028
BT_GAP_PER_ADV_SLOW_INT_MAX (*C macro*), 2027
BT_GAP_PER_ADV_SLOW_INT_MIN (*C macro*), 2027
BT_GAP_RSSI_INVALID (*C macro*), 2028
BT_GAP_SCAN_FAST_INTERVAL (*C macro*), 2027
BT_GAP_SCAN_FAST_INTERVAL_MIN (*C macro*), 2027
BT_GAP_SCAN_FAST_WINDOW (*C macro*), 2027
BT_GAP_SCAN_SLOW_INTERVAL_1 (*C macro*), 2027
BT_GAP_SCAN_SLOW_INTERVAL_2 (*C macro*), 2027
BT_GAP_SCAN_SLOW_WINDOW_1 (*C macro*), 2027
BT_GAP_SCAN_SLOW_WINDOW_2 (*C macro*), 2027
BT_GAP_SID_INVALID (*C macro*), 2028
BT_GAP_SID_MAX (*C macro*), 2028
BT_GAP_TX_POWER_INVALID (*C macro*), 2028
bt_gatt_attr (*C struct*), 2039
bt_gatt_attr_func_t (*C type*), 2046
bt_gatt_attr_get_handle (*C function*), 2049
bt_gatt_attr_next (*C function*), 2049
bt_gatt_attr_read (*C function*), 2050
bt_gatt_attr_read_ccc (*C function*), 2051
bt_gatt_attr_read_cep (*C function*), 2052
bt_gatt_attr_read_chrc (*C function*), 2051
bt_gatt_attr_read_cpf (*C function*), 2053
bt_gatt_attr_read_cud (*C function*), 2053
bt_gatt_attr_read_func_t (*C type*), 2037
bt_gatt_attr_read_included (*C function*), 2050
bt_gatt_attr_read_service (*C function*), 2050
bt_gatt_attr_value_handle (*C function*), 2049
bt_gatt_attr_write_ccc (*C function*), 2052
bt_gatt_attr_write_func_t (*C type*), 2037
bt_gatt_attr.handle (*C var*), 2039
BT_GATT_ATTRIBUTE (*C macro*), 2046
bt_gatt_attr.perm (*C var*), 2040
bt_gatt_attr.read (*C var*), 2039
bt_gatt_attr.user_data (*C var*), 2039
bt_gatt_attr.uuid (*C var*), 2039
bt_gatt_attr.write (*C var*), 2039
bt_gatt_authorization_cb (*C struct*), 2041
bt_gatt_authorization_cb_register (*C function*), 2047
bt_gatt_authorization_cb.read_authorize (*C var*), 2041

bt_gatt_authorization_cb.write_authorize (*C var*), 2041
bt_gatt_cancel (*C function*), 2067
bt_gatt_cb (*C struct*), 2041
bt_gatt_cb_register (*C function*), 2047
bt_gatt_cb.att_mtu_updated (*C var*), 2041
BT_GATT_CCC (*C macro*), 2045
bt_gatt_ccc (*C struct*), 2042
bt_gatt_ccc_cfg (*C struct*), 2057
bt_gatt_ccc_cfg.id (*C var*), 2057
bt_gatt_ccc_cfg.peer (*C var*), 2057
bt_gatt_ccc_cfg.value (*C var*), 2057
BT_GATT_CCC_INDICATE (*C macro*), 2036
BT_GATT_CCC_INITIALIZER (*C macro*), 2044
BT_GATT_CCC_MANAGED (*C macro*), 2045
BT_GATT_CCC_MAX (*C macro*), 2044
BT_GATT_CCC_NOTIFY (*C macro*), 2036
bt_gatt_ccc.flags (*C var*), 2042
BT_GATT_CEP (*C macro*), 2045
bt_gatt_cep (*C struct*), 2042
BT_GATT_CEP_RELIABLE_WRITE (*C macro*), 2036
BT_GATT_CEP_WRITABLE_AUX (*C macro*), 2036
bt_gatt_cep.properties (*C var*), 2042
BT_GATT_CHARACTERISTIC (*C macro*), 2044
bt_gatt_chrc (*C struct*), 2042
BT_GATT_CHRC_AUTH (*C macro*), 2036
BT_GATT_CHRC_BROADCAST (*C macro*), 2036
BT_GATT_CHRC_EXT_PROP (*C macro*), 2036
BT_GATT_CHRC_INDICATE (*C macro*), 2036
BT_GATT_CHRC_INIT (*C macro*), 2044
BT_GATT_CHRC_NOTIFY (*C macro*), 2036
BT_GATT_CHRC_READ (*C macro*), 2036
BT_GATT_CHRC_WRITE (*C macro*), 2036
BT_GATT_CHRC_WRITE_WITHOUT_RESP (*C macro*), 2036
bt_gatt_chrc.properties (*C var*), 2042
bt_gatt_chrc.uuid (*C var*), 2042
bt_gatt_chrc.value_handle (*C var*), 2042
bt_gatt_complete_func_t (*C type*), 2046
BT_GATT_CPF (*C macro*), 2045
bt_gatt_cpf (*C struct*), 2042
bt_gatt_cpf.description (*C var*), 2043
bt_gatt_cpf.exponent (*C var*), 2043
bt_gatt_cpf.format (*C var*), 2043
bt_gatt_cpf.name_space (*C var*), 2043
bt_gatt_cpf.unit (*C var*), 2043
BT_GATT_CUD (*C macro*), 2045
BT_GATT_DESCRIPTOR (*C macro*), 2045
bt_gatt_discover (*C function*), 2063
bt_gatt_discover_func_t (*C type*), 2058
bt_gatt_discover_params (*C struct*), 2068
bt_gatt_discover_params.attr_handle (*C var*), 2068
bt_gatt_discover_params.end_handle (*C var*), 2068
bt_gatt_discover_params.func (*C var*), 2068
bt_gatt_discover_params.start_handle (*C var*), 2068
bt_gatt_discover_params.sub_params (*C var*), 2068
bt_gatt_discover_params.type (*C var*), 2068
bt_gatt_discover_params.uuid (*C var*), 2068
BT_GATT_ERR (*C macro*), 2035
bt_gatt_err_to_str (*C function*), 2047

bt_gatt_exchange_mtu (*C function*), 2062
bt_gatt_exchange_params (*C struct*), 2067
bt_gatt_exchange_params.func (*C var*), 2068
bt_gatt_find_by_uuid (*C function*), 2049
bt_gatt_foreach_attr (*C function*), 2049
bt_gatt_foreach_attr_type (*C function*), 2048
bt_gatt_get_mtu (*C function*), 2056
bt_gatt_include (*C struct*), 2040
BT_GATT_INCLUDE_SERVICE (*C macro*), 2044
bt_gatt_include.end_handle (*C var*), 2041
bt_gatt_include.start_handle (*C var*), 2041
bt_gatt_include.uuid (*C var*), 2041
bt_gatt_indicate (*C function*), 2055
bt_gatt_indicate_func_t (*C type*), 2046
bt_gatt_indicate_params (*C struct*), 2057
bt_gatt_indicate_params_destroy_t (*C type*), 2047
bt_gatt_indicate_params.attr (*C var*), 2058
bt_gatt_indicate_params.data (*C var*), 2058
bt_gatt_indicate_params.destroy (*C var*), 2058
bt_gatt_indicate_params.func (*C var*), 2058
bt_gatt_indicate_params.len (*C var*), 2058
bt_gatt_indicate_params.uuid (*C var*), 2058
bt_gatt_is_subscribed (*C function*), 2056
bt_gatt_notify (*C function*), 2055
bt_gatt_notify_cb (*C function*), 2053
bt_gatt_notify_func_t (*C type*), 2060
bt_gatt_notify_multiple (*C function*), 2054
bt_gatt_notify_params (*C struct*), 2057
bt_gatt_notify_params.attr (*C var*), 2057
bt_gatt_notify_params.data (*C var*), 2057
bt_gatt_notify_params.func (*C var*), 2057
bt_gatt_notify_params.len (*C var*), 2057
bt_gatt_notify_params.user_data (*C var*), 2057
bt_gatt_notify_params.uuid (*C var*), 2057
bt_gatt_notify_uuid (*C function*), 2055
bt_gatt_perm (*C enum*), 2038
bt_gatt_perm.BT_GATT_PERM_NONE (*C enumerator*), 2038
bt_gatt_perm.BT_GATT_PERM_PREPARE_WRITE (*C enumerator*), 2038
bt_gatt_perm.BT_GATT_PERM_READ (*C enumerator*), 2038
bt_gatt_perm.BT_GATT_PERM_READ_AUTHEN (*C enumerator*), 2038
bt_gatt_perm.BT_GATT_PERM_READ_ENCRYPT (*C enumerator*), 2038
bt_gatt_perm.BT_GATT_PERM_READ_LESC (*C enumerator*), 2038
bt_gatt_perm.BT_GATT_PERM_WRITE (*C enumerator*), 2038
bt_gatt_perm.BT_GATT_PERM_WRITE_AUTHEN (*C enumerator*), 2038
bt_gatt_perm.BT_GATT_PERM_WRITE_ENCRYPT (*C enumerator*), 2038
bt_gatt_perm.BT_GATT_PERM_WRITE_LESC (*C enumerator*), 2039
BT_GATT_PRIMARY_SERVICE (*C macro*), 2044
bt_gatt_read (*C function*), 2063
bt_gatt_read_func_t (*C type*), 2059
bt_gatt_read_params (*C struct*), 2068
bt_gatt_read_params.end_handle (*C var*), 2069
bt_gatt_read_params.func (*C var*), 2068
bt_gatt_read_params.handle (*C var*), 2068
bt_gatt_read_params.handle_count (*C var*), 2068
bt_gatt_read_params.handles (*C var*), 2069
bt_gatt_read_params.offset (*C var*), 2069
bt_gatt_read_params.start_handle (*C var*), 2069
bt_gatt_read_params.uuid (*C var*), 2069

bt_gatt_read_params.variable (*C var*), 2069
bt_gatt_resubscribe (*C function*), 2066
bt_gatt_scc (*C struct*), 2042
BT_GATT_SCC_BROADCAST (*C macro*), 2037
bt_gatt_scc.flags (*C var*), 2042
BT_GATT_SECONDARY_SERVICE (*C macro*), 2044
BT_GATT_SERVICE (*C macro*), 2043
bt_gatt_service (*C struct*), 2040
BT_GATT_SERVICE_DEFINE (*C macro*), 2043
BT_GATT_SERVICE_INSTANCE_DEFINE (*C macro*), 2043
bt_gatt_service_is_registered (*C function*), 2048
bt_gatt_service_register (*C function*), 2048
bt_gatt_service_static (*C struct*), 2040
bt_gatt_service_static.attr_count (*C var*), 2040
bt_gatt_service_static.attrs (*C var*), 2040
bt_gatt_service_unregister (*C function*), 2048
bt_gatt_service_val (*C struct*), 2040
bt_gatt_service_val.end_handle (*C var*), 2040
bt_gatt_service_val.uuid (*C var*), 2040
bt_gatt_service.attr_count (*C var*), 2040
bt_gatt_service.attrs (*C var*), 2040
bt_gatt_subscribe (*C function*), 2065
bt_gatt_subscribe_func_t (*C type*), 2060
bt_gatt_subscribe_params (*C struct*), 2069
bt_gatt_subscribe_params.ccc_handle (*C var*), 2070
bt_gatt_subscribe_params.disc_params (*C var*), 2070
bt_gatt_subscribe_params.end_handle (*C var*), 2070
bt_gatt_subscribe_params.flags (*C var*), 2070
bt_gatt_subscribe_params.min_security (*C var*), 2070
bt_gatt_subscribe_params.notify (*C var*), 2070
bt_gatt_subscribe_params.subscribe (*C var*), 2070
bt_gatt_subscribe_params.value (*C var*), 2070
bt_gatt_subscribe_params.value_handle (*C var*), 2070
bt_gatt_unsubscribe (*C function*), 2067
bt_gatt_write (*C function*), 2064
bt_gatt_write_func_t (*C type*), 2059
bt_gatt_write_params (*C struct*), 2069
bt_gatt_write_params.data (*C var*), 2069
bt_gatt_write_params.func (*C var*), 2069
bt_gatt_write_params.handle (*C var*), 2069
bt_gatt_write_params.length (*C var*), 2069
bt_gatt_write_params.offset (*C var*), 2069
bt_gatt_write_without_response (*C function*), 2065
bt_gatt_write_without_response_cb (*C function*), 2064
bt_get_appearance (*C function*), 1970
bt_get_name (*C function*), 1970
bt_hci_cmd_complete_create (*C function*), 2348
bt_hci_cmd_status_create (*C function*), 2348
bt_hci_driver (*C struct*), 2349
bt_hci_driver_bus (*C enum*), 2346
bt_hci_driver_bus.BT_HCI_DRIVER_BUS_I2C (*C enumerator*), 2346
bt_hci_driver_bus.BT_HCI_DRIVER_BUS_IPM (*C enumerator*), 2346
bt_hci_driver_bus.BT_HCI_DRIVER_BUS_PCCARD (*C enumerator*), 2346
bt_hci_driver_bus.BT_HCI_DRIVER_BUS_PCI (*C enumerator*), 2346
bt_hci_driver_bus.BT_HCI_DRIVER_BUS_RS232 (*C enumerator*), 2346
bt_hci_driver_bus.BT_HCI_DRIVER_BUS_SDIO (*C enumerator*), 2346
bt_hci_driver_bus.BT_HCI_DRIVER_BUS_SPI (*C enumerator*), 2346
bt_hci_driver_bus.BT_HCI_DRIVER_BUS_UART (*C enumerator*), 2346

bt_hci_driver_bus.BT_HCI_DRIVER_BUS_USB (*C enumerator*), 2346
bt_hci_driver_bus.BT_HCI_DRIVER_BUS_VIRTUAL (*C enumerator*), 2346
bt_hci_driver_register (*C function*), 2347
bt_hci_driver.bus (*C var*), 2349
bt_hci_driver.close (*C var*), 2349
bt_hci_driver.name (*C var*), 2349
bt_hci_driver.open (*C var*), 2349
bt_hci_driver.quirks (*C var*), 2349
bt_hci_driver.send (*C var*), 2349
bt_hci_driver.setup (*C var*), 2350
BT_HCI_ERR_EXT_HANDLED (*C macro*), 2350
bt_hci_evt_create (*C function*), 2348
BT_HCI_RAW_CMD_EXT (*C macro*), 2350
bt_hci_raw_cmd_ext (*C struct*), 2352
bt_hci_raw_cmd_ext_register (*C function*), 2351
bt_hci_raw_cmd_ext.func (*C var*), 2352
bt_hci_raw_cmd_ext.min_len (*C var*), 2352
bt_hci_raw_cmd_ext.op (*C var*), 2352
bt_hci_raw_get_mode (*C function*), 2351
bt_hci_raw_set_mode (*C function*), 2351
bt_hci_setup_params (*C struct*), 2348
bt_hci_setup_params.public_addr (*C var*), 2349
bt_hci_transport_setup (*C function*), 2347
bt_hci_transport_teardown (*C function*), 2347
bt_hfp_hf_at_cmd (*C enum*), 1705
bt_hfp_hf_at_cmd.BT_HFP_HF_AT_CHUP (*C enumerator*), 1705
bt_hfp_hf_at_cmd.BT_HFP_HF_ATA (*C enumerator*), 1705
bt_hfp_hf_cb (*C struct*), 1706
bt_hfp_hf_cb.battery (*C var*), 1707
bt_hfp_hf_cb.call (*C var*), 1707
bt_hfp_hf_cb.call_held (*C var*), 1707
bt_hfp_hf_cb.call_setup (*C var*), 1707
bt_hfp_hf_cb.cmd_complete_cb (*C var*), 1708
bt_hfp_hf_cb.connected (*C var*), 1706
bt_hfp_hf_cb.disconnected (*C var*), 1706
bt_hfp_hf_cb.ring_indication (*C var*), 1707
bt_hfp_hf_cb.roam (*C var*), 1707
bt_hfp_hf_cb.sco_connected (*C var*), 1706
bt_hfp_hf_cb.sco_disconnected (*C var*), 1706
bt_hfp_hf_cb.service (*C var*), 1706
bt_hfp_hf_cb.signal (*C var*), 1707
bt_hfp_hf_cmd_complete (*C struct*), 1706
bt_hfp_hf_register (*C function*), 1705
bt_hfp_hf_send_cmd (*C function*), 1705
bt_hrs_cb (*C struct*), 1936
bt_hrs_cb_register (*C function*), 1936
bt_hrs_cb_unregister (*C function*), 1936
bt_hrs_cb.ntf_changed (*C var*), 1936
bt_hrs_notify (*C function*), 1936
bt_ias_alert_lvl (*C enum*), 1937
bt_ias_alert_lvl.BT_IAS_ALERT_LVL_HIGH_ALERT (*C enumerator*), 1937
bt_ias_alert_lvl.BT_IAS_ALERT_LVL_MILD_ALERT (*C enumerator*), 1937
bt_ias_alert_lvl.BT_IAS_ALERT_LVL_NO_ALERT (*C enumerator*), 1937
bt_ias_cb (*C struct*), 1938
BT_IAS_CB_DEFINE (*C macro*), 1937
bt_ias_cb.high_alert (*C var*), 1938
bt_ias_cb.mild_alert (*C var*), 1938
bt_ias_cb.no_alert (*C var*), 1938

bt_ias_client_alert_write (*C function*), 1937
bt_ias_client_cb (*C struct*), 1938
bt_ias_client_cb_register (*C function*), 1938
bt_ias_client_cb.discover (*C var*), 1938
bt_ias_discover (*C function*), 1937
bt_ias_local_alert_stop (*C function*), 1937
bt_id_create (*C function*), 1971
BT_ID_DEFAULT (*C macro*), 1957
bt_id_delete (*C function*), 1972
bt_id_get (*C function*), 1970
bt_id_reset (*C function*), 1972
bt_is_ready (*C function*), 1969
bt_l2cap_br_chan (*C struct*), 2306
bt_l2cap_br_chan.chan (*C var*), 2306
bt_l2cap_br_chan.ident (*C var*), 2306
bt_l2cap_br_chan.psm (*C var*), 2306
bt_l2cap_br_chan.rx (*C var*), 2306
bt_l2cap_br_chan.tx (*C var*), 2306
bt_l2cap_br_endpoint (*C struct*), 2306
bt_l2cap_br_endpoint.cid (*C var*), 2306
bt_l2cap_br_endpoint.mtu (*C var*), 2306
bt_l2cap_br_server_register (*C function*), 2301
BT_L2CAP_BUF_SIZE (*C macro*), 2299
bt_l2cap_chan (*C struct*), 2304
bt_l2cap_chan_connect (*C function*), 2302
bt_l2cap_chan_destroy_t (*C type*), 2300
bt_l2cap_chan_disconnect (*C function*), 2303
bt_l2cap_chan_give_credits (*C function*), 2304
bt_l2cap_chan_ops (*C struct*), 2306
bt_l2cap_chan_ops.alloc_buf (*C var*), 2307
bt_l2cap_chan_ops.alloc_seg (*C var*), 2307
bt_l2cap_chan_ops.connected (*C var*), 2306
bt_l2cap_chan_ops.disconnected (*C var*), 2306
bt_l2cap_chan_ops.encrypt_change (*C var*), 2307
bt_l2cap_chan_ops.reconfigured (*C var*), 2308
bt_l2cap_chan_ops.recv (*C var*), 2307
bt_l2cap_chan_ops.seg_recv (*C var*), 2308
bt_l2cap_chan_ops.sent (*C var*), 2308
bt_l2cap_chan_ops.status (*C var*), 2308
bt_l2cap_chan_recv_complete (*C function*), 2304
bt_l2cap_chan_send (*C function*), 2303
BT_L2CAP_CHAN_SEND_RESERVE (*C macro*), 2300
bt_l2cap_chan_state (*C enum*), 2300
bt_l2cap_chan_state_t (*C type*), 2300
bt_l2cap_chan_state.BT_L2CAP_CONFIG (*C enumerator*), 2301
bt_l2cap_chan_state.BT_L2CAP_CONNECTED (*C enumerator*), 2301
bt_l2cap_chan_state.BT_L2CAP_CONNECTING (*C enumerator*), 2300
bt_l2cap_chan_state.BT_L2CAP_DISCONNECTED (*C enumerator*), 2300
bt_l2cap_chan_state.BT_L2CAP_DISCONNECTING (*C enumerator*), 2301
bt_l2cap_chan_status (*C enum*), 2301
bt_l2cap_chan_status_t (*C type*), 2300
bt_l2cap_chan_status.BT_L2CAP_NUM_STATUS (*C enumerator*), 2301
bt_l2cap_chan_status.BT_L2CAP_STATUS_ENCRYPT_PENDING (*C enumerator*), 2301
bt_l2cap_chan_status.BT_L2CAP_STATUS_OUT (*C enumerator*), 2301
bt_l2cap_chan_status.BT_L2CAP_STATUS_SHUTDOWN (*C enumerator*), 2301
bt_l2cap_chan.conn (*C var*), 2304
bt_l2cap_chan.ops (*C var*), 2304
bt_l2cap_ecred_chan_connect (*C function*), 2302

bt_l2cap_ecred_chan_reconfigure (*C function*), 2302
BT_L2CAP_HDR_SIZE (*C macro*), 2299
BT_L2CAP_LE_CHAN (*C macro*), 2299
bt_l2cap_le_chan (*C struct*), 2305
bt_l2cap_le_chan.chan (*C var*), 2305
bt_l2cap_le_chan.pending_rx_mtu (*C var*), 2305
bt_l2cap_le_chan.rx (*C var*), 2305
bt_l2cap_le_chan.tx (*C var*), 2305
bt_l2cap_le_chan.tx_queue (*C var*), 2305
bt_l2cap_le_endpoint (*C struct*), 2304
bt_l2cap_le_endpoint.cid (*C var*), 2305
bt_l2cap_le_endpoint.credits (*C var*), 2305
bt_l2cap_le_endpoint.mps (*C var*), 2305
bt_l2cap_le_endpoint.mtu (*C var*), 2305
BT_L2CAP_RX_MTU (*C macro*), 2299
BT_L2CAP_SDU_BUF_SIZE (*C macro*), 2299
BT_L2CAP_SDU_CHAN_SEND_RESERVE (*C macro*), 2300
BT_L2CAP_SDU_HDR_SIZE (*C macro*), 2299
BT_L2CAP_SDU_RX_MTU (*C macro*), 2299
BT_L2CAP_SDU_TX_MTU (*C macro*), 2299
bt_l2cap_server (*C struct*), 2309
bt_l2cap_server_register (*C function*), 2301
bt_l2cap_server.accept (*C var*), 2309
bt_l2cap_server.psm (*C var*), 2309
bt_l2cap_server.sec_level (*C var*), 2309
BT_L2CAP_TX_MTU (*C macro*), 2299
BT_LE_AD_GENERAL (*C macro*), 2008
BT_LE_AD_LIMITED (*C macro*), 2008
BT_LE_AD_NO_BREDR (*C macro*), 2008
BT_LE_ADV_CONN (*C macro*), 1958
BT_LE_ADV_CONN_DIR (*C macro*), 1958
BT_LE_ADV_CONN_DIR_LOW_DUTY (*C macro*), 1959
BT_LE_ADV_CONN_NAME (*C macro*), 1958
BT_LE_ADV_CONN_NAME_AD (*C macro*), 1959
BT_LE_ADV_CONN_ONE_TIME (*C macro*), 1958
BT_LE_ADV_NCONN (*C macro*), 1959
BT_LE_ADV_NCONN_IDENTITY (*C macro*), 1959
BT_LE_ADV_NCONN_NAME (*C macro*), 1959
BT_LE_ADV_PARAM (*C macro*), 1958
bt_le_adv_param (*C struct*), 1989
BT_LE_ADV_PARAM_INIT (*C macro*), 1958
bt_le_adv_param.id (*C var*), 1990
bt_le_adv_param.interval_max (*C var*), 1990
bt_le_adv_param.interval_min (*C var*), 1990
bt_le_adv_param.options (*C var*), 1990
bt_le_adv_param.peer (*C var*), 1990
bt_le_adv_param.secondary_max_skip (*C var*), 1990
bt_le_adv_param.sid (*C var*), 1990
bt_le_adv_start (*C function*), 1973
bt_le_adv_stop (*C function*), 1974
bt_le_adv_update_data (*C function*), 1973
BT_LE_CONN_PARAM (*C macro*), 2310
bt_le_conn_param (*C struct*), 2329
BT_LE_CONN_PARAM_DEFAULT (*C macro*), 2310
BT_LE_CONN_PARAM_INIT (*C macro*), 2310
BT_LE_DATA_LEN_PARAM_DEFAULT (*C macro*), 2311
BT_LE_DATA_LEN_PARAM_MAX (*C macro*), 2311
bt_le_ext_adv_cb (*C struct*), 1989

bt_le_ext_adv_cb.connected (*C var*), 1989
bt_le_ext_adv_cb.scanned (*C var*), 1989
bt_le_ext_adv_cb.sent (*C var*), 1989
BT_LE_EXT_ADV_CODED_NCONN (*C macro*), 1960
BT_LE_EXT_ADV_CODED_NCONN_IDENTITY (*C macro*), 1960
BT_LE_EXT_ADV_CODED_NCONN_NAME (*C macro*), 1960
BT_LE_EXT_ADV_CONN (*C macro*), 1959
BT_LE_EXT_ADV_CONN_NAME (*C macro*), 1959
bt_le_ext_adv_connected_info (*C struct*), 1987
bt_le_ext_adv_connected_info.conn (*C var*), 1988
bt_le_ext_adv_create (*C function*), 1974
bt_le_ext_adv_delete (*C function*), 1976
bt_le_ext_adv_get_index (*C function*), 1976
bt_le_ext_adv_get_info (*C function*), 1976
bt_le_ext_adv_info (*C struct*), 1991
bt_le_ext_adv_info.addr (*C var*), 1992
bt_le_ext_adv_info.tx_power (*C var*), 1992
BT_LE_EXT_ADV_NCONN (*C macro*), 1959
BT_LE_EXT_ADV_NCONN_IDENTITY (*C macro*), 1960
BT_LE_EXT_ADV_NCONN_NAME (*C macro*), 1959
bt_le_ext_adv_oob_get_local (*C function*), 1984
BT_LE_EXT_ADV_SCAN (*C macro*), 1959
BT_LE_EXT_ADV_SCAN_NAME (*C macro*), 1959
bt_le_ext_adv_scanned_info (*C struct*), 1988
bt_le_ext_adv_scanned_info.addr (*C var*), 1988
bt_le_ext_adv_sent_info (*C struct*), 1987
bt_le_ext_adv_sent_info.num_sent (*C var*), 1987
bt_le_ext_adv_set_data (*C function*), 1975
bt_le_ext_adv_start (*C function*), 1974
BT_LE_EXT_ADV_START_DEFAULT (*C macro*), 1960
BT_LE_EXT_ADV_START_PARAM (*C macro*), 1960
bt_le_ext_adv_start_param (*C struct*), 1991
BT_LE_EXT_ADV_START_PARAM_INIT (*C macro*), 1960
bt_le_ext_adv_start_param.num_events (*C var*), 1991
bt_le_ext_adv_start_param.timeout (*C var*), 1991
bt_le_ext_adv_stop (*C function*), 1974
bt_le_ext_adv_update_param (*C function*), 1975
bt_le_filter_accept_list_add (*C function*), 1982
bt_le_filter_accept_list_clear (*C function*), 1983
bt_le_filter_accept_list_remove (*C function*), 1982
bt_le_oob (*C struct*), 1998
bt_le_oob_get_local (*C function*), 1984
bt_le_oob_get_sc_data (*C function*), 2326
bt_le_oob_sc_data (*C struct*), 1998
bt_le_oob_sc_data.c (*C var*), 1998
bt_le_oob_sc_data.r (*C var*), 1998
bt_le_oob_set_legacy_flag (*C function*), 2325
bt_le_oob_set_legacy_tk (*C function*), 2325
bt_le_oob_set_sc_data (*C function*), 2325
bt_le_oob_set_sc_flag (*C function*), 2325
bt_le_oob.addr (*C var*), 1998
bt_le_oob.le_sc_data (*C var*), 1998
bt_le_per_adv_data_request (*C struct*), 1988
bt_le_per_adv_data_request.count (*C var*), 1988
bt_le_per_adv_data_request.start (*C var*), 1988
BT_LE_PER_ADV_DEFAULT (*C macro*), 1961
bt_le_per_adv_list_add (*C function*), 1980
bt_le_per_adv_list_clear (*C function*), 1981

bt_le_per_adv_list_remove (*C function*), 1981
BT_LE_PER_ADV_PARAM (*C macro*), 1960
bt_le_per_adv_param (*C struct*), 1991
BT_LE_PER_ADV_PARAM_INIT (*C macro*), 1960
bt_le_per_adv_param.interval_max (*C var*), 1991
bt_le_per_adv_param.interval_min (*C var*), 1991
bt_le_per_adv_param.options (*C var*), 1991
bt_le_per_adv_response_info (*C struct*), 1988
bt_le_per_adv_response_info.cte_type (*C var*), 1989
bt_le_per_adv_response_info.response_slot (*C var*), 1989
bt_le_per_adv_response_info.rssi (*C var*), 1988
bt_le_per_adv_response_info.subevent (*C var*), 1988
bt_le_per_adv_response_info.tx_power (*C var*), 1988
bt_le_per_adv_response_info.tx_status (*C var*), 1988
bt_le_per_adv_response_params (*C struct*), 2000
bt_le_per_adv_response_params.request_event (*C var*), 2000
bt_le_per_adv_response_params.request_subevent (*C var*), 2000
bt_le_per_adv_response_params.response_slot (*C var*), 2000
bt_le_per_adv_response_params.response_subevent (*C var*), 2000
bt_le_per_adv_set_data (*C function*), 1976
bt_le_per_adv_set_info_transfer (*C function*), 1980
bt_le_per_adv_set_param (*C function*), 1976
bt_le_per_adv_set_response_data (*C function*), 1987
bt_le_per_adv_set_subevent_data (*C function*), 1977
bt_le_per_adv_start (*C function*), 1977
bt_le_per_adv_stop (*C function*), 1977
bt_le_per_adv_subevent_data_params (*C struct*), 1992
bt_le_per_adv_subevent_data_params.data (*C var*), 1992
bt_le_per_adv_subevent_data_params.response_slot_count (*C var*), 1992
bt_le_per_adv_subevent_data_params.response_slot_start (*C var*), 1992
bt_le_per_adv_subevent_data_params.subevent (*C var*), 1992
bt_le_per_adv_sync_cb (*C struct*), 1994
bt_le_per_adv_sync_cb_register (*C function*), 1979
bt_le_per_adv_sync_cb.biginfo (*C var*), 1995
bt_le_per_adv_sync_cb.cte_report_cb (*C var*), 1995
bt_le_per_adv_sync_cb.recv (*C var*), 1994
bt_le_per_adv_sync_cb.state_changed (*C var*), 1994
bt_le_per_adv_sync_cb.synced (*C var*), 1994
bt_le_per_adv_sync_cb.term (*C var*), 1994
bt_le_per_adv_sync_create (*C function*), 1978
bt_le_per_adv_sync_delete (*C function*), 1979
bt_le_per_adv_sync_get_index (*C function*), 1977
bt_le_per_adv_sync_get_info (*C function*), 1978
bt_le_per_adv_sync_info (*C struct*), 1995
bt_le_per_adv_sync_info.addr (*C var*), 1996
bt_le_per_adv_sync_info.interval (*C var*), 1996
bt_le_per_adv_sync_info.phy (*C var*), 1996
bt_le_per_adv_sync_info.sid (*C var*), 1996
bt_le_per_adv_sync_lookup_addr (*C function*), 1978
bt_le_per_adv_sync_lookup_index (*C function*), 1978
bt_le_per_adv_sync_param (*C struct*), 1995
bt_le_per_adv_sync_param.addr (*C var*), 1995
bt_le_per_adv_sync_param.options (*C var*), 1995
bt_le_per_adv_sync_param.sid (*C var*), 1995
bt_le_per_adv_sync_param.skip (*C var*), 1995
bt_le_per_adv_sync_param.timeout (*C var*), 1995
bt_le_per_adv_sync_recv_disable (*C function*), 1979
bt_le_per_adv_sync_recv_enable (*C function*), 1979

bt_le_per_adv_sync_recv_info (*C struct*), 1993
bt_le_per_adv_sync_recv_info.addr (*C var*), 1993
bt_le_per_adv_sync_recv_info.cte_type (*C var*), 1993
bt_le_per_adv_sync_recv_info.rssi (*C var*), 1993
bt_le_per_adv_sync_recv_info.sid (*C var*), 1993
bt_le_per_adv_sync_recv_info.tx_power (*C var*), 1993
bt_le_per_adv_sync_state_info (*C struct*), 1993
bt_le_per_adv_sync_state_info.recv_enabled (*C var*), 1994
bt_le_per_adv_sync_subevent (*C function*), 1987
bt_le_per_adv_sync_subevent_params (*C struct*), 2000
bt_le_per_adv_sync_subevent_params.num_subevents (*C var*), 2000
bt_le_per_adv_sync_subevent_params.properties (*C var*), 2000
bt_le_per_adv_sync_subevent_params.subevents (*C var*), 2000
bt_le_per_adv_sync_synced_info (*C struct*), 1992
bt_le_per_adv_sync_synced_info.addr (*C var*), 1992
bt_le_per_adv_sync_synced_info.conn (*C var*), 1993
bt_le_per_adv_sync_synced_info.interval (*C var*), 1992
bt_le_per_adv_sync_synced_info.phy (*C var*), 1992
bt_le_per_adv_sync_synced_info.recv_enabled (*C var*), 1992
bt_le_per_adv_sync_synced_info.service_data (*C var*), 1992
bt_le_per_adv_sync_synced_info.sid (*C var*), 1992
bt_le_per_adv_sync_term_info (*C struct*), 1993
bt_le_per_adv_sync_term_info.addr (*C var*), 1993
bt_le_per_adv_sync_term_info.reason (*C var*), 1993
bt_le_per_adv_sync_term_info.sid (*C var*), 1993
bt_le_per_adv_sync_transfer (*C function*), 1979
bt_le_per_adv_sync_transfer_param (*C struct*), 1996
bt_le_per_adv_sync_transfer_param.options (*C var*), 1996
bt_le_per_adv_sync_transfer_param.skip (*C var*), 1996
bt_le_per_adv_sync_transfer_param.timeout (*C var*), 1996
bt_le_per_adv_sync_transfer_subscribe (*C function*), 1980
bt_le_per_adv_sync_transfer_unsubscribe (*C function*), 1980
BT_LE_SCAN_ACTIVE (*C macro*), 1961
BT_LE_SCAN_ACTIVE_CONTINUOUS (*C macro*), 1961
bt_le_scan_cb (*C struct*), 1998
bt_le_scan_cb_register (*C function*), 1982
bt_le_scan_cb_t (*C type*), 1962
bt_le_scan_cb_unregister (*C function*), 1982
bt_le_scan_cb.recv (*C var*), 1998
bt_le_scan_cb.timeout (*C var*), 1998
BT_LE_SCAN_CODED_ACTIVE (*C macro*), 1961
BT_LE_SCAN_CODED_PASSIVE (*C macro*), 1961
BT_LE_SCAN_OPT_FILTER_WHITELIST (*C macro*), 1961
BT_LE_SCAN_PARAM (*C macro*), 1961
bt_le_scan_param (*C struct*), 1996
BT_LE_SCAN_PARAM_INIT (*C macro*), 1961
bt_le_scan_param.interval (*C var*), 1996
bt_le_scan_param.interval_coded (*C var*), 1997
bt_le_scan_param.options (*C var*), 1996
bt_le_scan_param.timeout (*C var*), 1997
bt_le_scan_param.type (*C var*), 1996
bt_le_scan_param.window (*C var*), 1997
bt_le_scan_param.window_coded (*C var*), 1997
BT_LE_SCAN_PASSIVE (*C macro*), 1961
BT_LE_SCAN_PASSIVE_CONTINUOUS (*C macro*), 1961
bt_le_scan_recv_info (*C struct*), 1997
bt_le_scan_recv_info.addr (*C var*), 1997
bt_le_scan_recv_info.adv_props (*C var*), 1997

bt_le_scan_recv_info.adv_type (*C var*), 1997
bt_le_scan_recv_info.interval (*C var*), 1998
bt_le_scan_recv_info.primary_phy (*C var*), 1998
bt_le_scan_recv_info.rssi (*C var*), 1997
bt_le_scan_recv_info.secondary_phy (*C var*), 1998
bt_le_scan_recv_info.sid (*C var*), 1997
bt_le_scan_recv_info.tx_power (*C var*), 1997
bt_le_scan_start (*C function*), 1981
bt_le_scan_stop (*C function*), 1982
bt_le_set_auto_conn (*C function*), 2323
bt_le_set_chan_map (*C function*), 1983
bt_le_set_rpa_timeout (*C function*), 1983
BT_LE_SUPP_FEAT_8_ENCODE (*C macro*), 2030
BT_LE_SUPP_FEAT_16_ENCODE (*C macro*), 2030
BT_LE_SUPP_FEAT_24_ENCODE (*C macro*), 2029
BT_LE_SUPP_FEAT_32_ENCODE (*C macro*), 2029
BT_LE_SUPP_FEAT_40_ENCODE (*C macro*), 2029
BT_LE_SUPP_FEAT_VALIDATE (*C macro*), 2030
bt_mcc_cb (*C struct*), 1887
bt_mcc_cb.cmd_ntf (*C var*), 1888
bt_mcc_cb.discover_mcs (*C var*), 1887
bt_mcc_cb.otc_current_group_object (*C var*), 1889
bt_mcc_cb.otc_current_track_object (*C var*), 1889
bt_mcc_cb.otc_icon_object (*C var*), 1889
bt_mcc_cb.otc_next_track_object (*C var*), 1889
bt_mcc_cb.otc_obj_metadata (*C var*), 1889
bt_mcc_cb.otc_obj_selected (*C var*), 1889
bt_mcc_cb.otc_parent_group_object (*C var*), 1889
bt_mcc_cb.otc_track_segments_object (*C var*), 1889
bt_mcc_cb.read_content_control_id (*C var*), 1889
bt_mcc_cb.read_current_group_obj_id (*C var*), 1888
bt_mcc_cb.read_current_track_obj_id (*C var*), 1888
bt_mcc_cb.read_icon_obj_id (*C var*), 1887
bt_mcc_cb.read_icon_url (*C var*), 1887
bt_mcc_cb.read_media_state (*C var*), 1888
bt_mcc_cb.read_next_track_obj_id (*C var*), 1888
bt_mcc_cb.read_opcodes_supported (*C var*), 1888
bt_mcc_cb.read_parent_group_obj_id (*C var*), 1888
bt_mcc_cb.read_playback_speed (*C var*), 1887
bt_mcc_cb.read_player_name (*C var*), 1887
bt_mcc_cb.read_playing_order (*C var*), 1888
bt_mcc_cb.read_playing_orders_supported (*C var*), 1888
bt_mcc_cb.read_search_results_obj_id (*C var*), 1889
bt_mcc_cb.read_seeking_speed (*C var*), 1888
bt_mcc_cb.read_segments_obj_id (*C var*), 1888
bt_mcc_cb.read_track_duration (*C var*), 1887
bt_mcc_cb.read_track_position (*C var*), 1887
bt_mcc_cb.read_track_title (*C var*), 1887
bt_mcc_cb.search_ntf (*C var*), 1889
bt_mcc_cb.send_cmd (*C var*), 1888
bt_mcc_cb.send_search (*C var*), 1889
bt_mcc_cb.set_current_group_obj_id (*C var*), 1888
bt_mcc_cb.set_current_track_obj_id (*C var*), 1888
bt_mcc_cb.set_next_track_obj_id (*C var*), 1888
bt_mcc_cb.set_playback_speed (*C var*), 1887
bt_mcc_cb.set_playing_order (*C var*), 1888
bt_mcc_cb.set_track_position (*C var*), 1887
bt_mcc_cb.track_changed_ntf (*C var*), 1887

bt_mcc_cmd_ntf_cb (*C type*), 1877
bt_mcc_discover_mcs (*C function*), 1881
bt_mcc_discover_mcs_cb (*C type*), 1872
bt_mcc_init (*C function*), 1881
bt_mcc_otc_inst (*C function*), 1887
bt_mcc_otc_obj_metadata_cb (*C type*), 1879
bt_mcc_otc_obj_selected_cb (*C type*), 1879
bt_mcc_otc_read_current_group_object (*C function*), 1886
bt_mcc_otc_read_current_group_object_cb (*C type*), 1881
bt_mcc_otc_read_current_track_object (*C function*), 1886
bt_mcc_otc_read_current_track_object_cb (*C type*), 1880
bt_mcc_otc_read_icon_object (*C function*), 1886
bt_mcc_otc_read_icon_object_cb (*C type*), 1879
bt_mcc_otc_read_next_track_object (*C function*), 1886
bt_mcc_otc_read_next_track_object_cb (*C type*), 1880
bt_mcc_otc_read_object_metadata (*C function*), 1886
bt_mcc_otc_read_parent_group_object (*C function*), 1886
bt_mcc_otc_read_parent_group_object_cb (*C type*), 1880
bt_mcc_otc_read_track_segments_object (*C function*), 1886
bt_mcc_otc_read_track_segments_object_cb (*C type*), 1879
bt_mcc_read_content_control_id (*C function*), 1885
bt_mcc_read_content_control_id_cb (*C type*), 1879
bt_mcc_read_current_group_obj_id (*C function*), 1884
bt_mcc_read_current_group_obj_id_cb (*C type*), 1876
bt_mcc_read_current_track_obj_id (*C function*), 1883
bt_mcc_read_current_track_obj_id_cb (*C type*), 1874
bt_mcc_read_icon_obj_id (*C function*), 1881
bt_mcc_read_icon_obj_id_cb (*C type*), 1872
bt_mcc_read_icon_url (*C function*), 1882
bt_mcc_read_icon_url_cb (*C type*), 1872
bt_mcc_read_media_state (*C function*), 1885
bt_mcc_read_media_state_cb (*C type*), 1877
bt_mcc_read_next_track_obj_id (*C function*), 1883
bt_mcc_read_next_track_obj_id_cb (*C type*), 1875
bt_mcc_read_opcodes_supported (*C function*), 1885
bt_mcc_read_opcodes_supported_cb (*C type*), 1878
bt_mcc_read_parent_group_obj_id (*C function*), 1884
bt_mcc_read_parent_group_obj_id_cb (*C type*), 1875
bt_mcc_read_playback_speed (*C function*), 1882
bt_mcc_read_playback_speed_cb (*C type*), 1874
bt_mcc_read_player_name (*C function*), 1881
bt_mcc_read_player_name_cb (*C type*), 1872
bt_mcc_read_playing_order (*C function*), 1884
bt_mcc_read_playing_order_cb (*C type*), 1876
bt_mcc_read_playing_orders_supported (*C function*), 1885
bt_mcc_read_playing_orders_supported_cb (*C type*), 1877
bt_mcc_read_search_results_obj_id (*C function*), 1885
bt_mcc_read_search_results_obj_id_cb (*C type*), 1878
bt_mcc_read_seeking_speed (*C function*), 1883
bt_mcc_read_seeking_speed_cb (*C type*), 1874
bt_mcc_read_segments_obj_id (*C function*), 1883
bt_mcc_read_segments_obj_id_cb (*C type*), 1874
bt_mcc_read_track_duration (*C function*), 1882
bt_mcc_read_track_duration_cb (*C type*), 1873
bt_mcc_read_track_position (*C function*), 1882
bt_mcc_read_track_position_cb (*C type*), 1873
bt_mcc_read_track_title (*C function*), 1882
bt_mcc_read_track_title_cb (*C type*), 1873

bt_mcc_search_ntf_cb (*C type*), 1878
bt_mcc_send_cmd (*C function*), 1885
bt_mcc_send_cmd_cb (*C type*), 1877
bt_mcc_send_search (*C function*), 1885
bt_mcc_send_search_cb (*C type*), 1878
bt_mcc_set_current_group_obj_id (*C function*), 1884
bt_mcc_set_current_group_obj_id_cb (*C type*), 1876
bt_mcc_set_current_track_obj_id (*C function*), 1883
bt_mcc_set_current_track_obj_id_cb (*C type*), 1875
bt_mcc_set_next_track_obj_id (*C function*), 1883
bt_mcc_set_next_track_obj_id_cb (*C type*), 1875
bt_mcc_set_playback_speed (*C function*), 1883
bt_mcc_set_playback_speed_cb (*C type*), 1874
bt_mcc_set_playing_order (*C function*), 1884
bt_mcc_set_playing_order_cb (*C type*), 1876
bt_mcc_set_track_position (*C function*), 1882
bt_mcc_set_track_position_cb (*C type*), 1873
bt_mcc_track_changed_ntf_cb (*C type*), 1872
BT_MCS_ERR_LONG_VAL_CHANGED (*C macro*), 1842
bt_mcs_get_ots (*C function*), 1856
BT_MCS_GROUP_OBJECT_GROUP_TYPE (*C macro*), 1842
BT_MCS_GROUP_OBJECT_TRACK_TYPE (*C macro*), 1842
BT_MCS_MEDIA_STATE_INACTIVE (*C macro*), 1837
BT_MCS_MEDIA_STATE_PAUSED (*C macro*), 1837
BT_MCS_MEDIA_STATE_PLAYING (*C macro*), 1837
BT_MCS_MEDIA_STATE_SEEKING (*C macro*), 1837
BT_MCS_OPC_FAST_FORWARD (*C macro*), 1838
BT_MCS_OPC_FAST_REWIND (*C macro*), 1838
BT_MCS_OPC_FIRST_GROUP (*C macro*), 1839
BT_MCS_OPC_FIRST_SEGMENT (*C macro*), 1838
BT_MCS_OPC_FIRST_TRACK (*C macro*), 1838
BT_MCS_OPC_GOTO_GROUP (*C macro*), 1839
BT_MCS_OPC_GOTO_SEGMENT (*C macro*), 1838
BT_MCS_OPC_GOTO_TRACK (*C macro*), 1839
BT_MCS_OPC_LAST_GROUP (*C macro*), 1839
BT_MCS_OPC_LAST_SEGMENT (*C macro*), 1838
BT_MCS_OPC_LAST_TRACK (*C macro*), 1838
BT_MCS_OPC_MOVE_RELATIVE (*C macro*), 1838
BT_MCS_OPC_NEXT_GROUP (*C macro*), 1839
BT_MCS_OPC_NEXT_SEGMENT (*C macro*), 1838
BT_MCS_OPC_NEXT_TRACK (*C macro*), 1838
BT_MCS_OPC_NTF_CANNOT_BE_COMPLETED (*C macro*), 1841
BT_MCS_OPC_NTF_NOT_SUPPORTED (*C macro*), 1840
BT_MCS_OPC_NTF_PLAYER_INACTIVE (*C macro*), 1840
BT_MCS_OPC_NTF_SUCCESS (*C macro*), 1840
BT_MCS_OPC_PAUSE (*C macro*), 1838
BT_MCS_OPC_PLAY (*C macro*), 1838
BT_MCS_OPC_PREV_GROUP (*C macro*), 1839
BT_MCS_OPC_PREV_SEGMENT (*C macro*), 1838
BT_MCS_OPC_PREV_TRACK (*C macro*), 1838
BT_MCS_OPC_STOP (*C macro*), 1838
BT_MCS_OPC_SUP_FAST_FORWARD (*C macro*), 1839
BT_MCS_OPC_SUP_FAST_REWIND (*C macro*), 1839
BT_MCS_OPC_SUP_FIRST_GROUP (*C macro*), 1840
BT_MCS_OPC_SUP_FIRST_SEGMENT (*C macro*), 1839
BT_MCS_OPC_SUP_FIRST_TRACK (*C macro*), 1840
BT_MCS_OPC_SUP_GOTO_GROUP (*C macro*), 1840
BT_MCS_OPC_SUP_GOTO_SEGMENT (*C macro*), 1840

BT_MCS_OPC_SUP_GOTO_TRACK (*C macro*), 1840
BT_MCS_OPC_SUP_LAST_GROUP (*C macro*), 1840
BT_MCS_OPC_SUP_LAST_SEGMENT (*C macro*), 1840
BT_MCS_OPC_SUP_LAST_TRACK (*C macro*), 1840
BT_MCS_OPC_SUP_MOVE_RELATIVE (*C macro*), 1839
BT_MCS_OPC_SUP_NEXT_GROUP (*C macro*), 1840
BT_MCS_OPC_SUP_NEXT_SEGMENT (*C macro*), 1839
BT_MCS_OPC_SUP_NEXT_TRACK (*C macro*), 1840
BT_MCS_OPC_SUP_PAUSE (*C macro*), 1839
BT_MCS_OPC_SUP_PLAY (*C macro*), 1839
BT_MCS_OPC_SUP_PREV_GROUP (*C macro*), 1840
BT_MCS_OPC_SUP_PREV_SEGMENT (*C macro*), 1839
BT_MCS_OPC_SUP_PREV_TRACK (*C macro*), 1840
BT_MCS_OPC_SUP_STOP (*C macro*), 1839
BT_MCS_OPCODES_SUPPORTED_LEN (*C macro*), 1842
BT_MCS_PLAYBACK_SPEED_DOUBLE (*C macro*), 1835
BT_MCS_PLAYBACK_SPEED_HALF (*C macro*), 1835
BT_MCS_PLAYBACK_SPEED_MAX (*C macro*), 1835
BT_MCS_PLAYBACK_SPEED_MIN (*C macro*), 1835
BT_MCS_PLAYBACK_SPEED_QUARTER (*C macro*), 1835
BT_MCS_PLAYBACK_SPEED_UNITY (*C macro*), 1835
BT_MCS_PLAYING_ORDER_INORDER_ONCE (*C macro*), 1836
BT_MCS_PLAYING_ORDER_INORDER_REPEAT (*C macro*), 1836
BT_MCS_PLAYING_ORDER_NEWEST_ONCE (*C macro*), 1836
BT_MCS_PLAYING_ORDER_NEWEST_REPEAT (*C macro*), 1836
BT_MCS_PLAYING_ORDER_OLDEST_ONCE (*C macro*), 1836
BT_MCS_PLAYING_ORDER_OLDEST_REPEAT (*C macro*), 1836
BT_MCS_PLAYING_ORDER_SHUFFLE_ONCE (*C macro*), 1836
BT_MCS_PLAYING_ORDER_SHUFFLE_REPEAT (*C macro*), 1836
BT_MCS_PLAYING_ORDER_SINGLE_ONCE (*C macro*), 1836
BT_MCS_PLAYING_ORDER_SINGLE_REPEAT (*C macro*), 1836
BT_MCS_PLAYING_ORDERS_SUPPORTED_INORDER_ONCE (*C macro*), 1837
BT_MCS_PLAYING_ORDERS_SUPPORTED_INORDER_REPEAT (*C macro*), 1837
BT_MCS_PLAYING_ORDERS_SUPPORTED_NEWEST_ONCE (*C macro*), 1837
BT_MCS_PLAYING_ORDERS_SUPPORTED_NEWEST_REPEAT (*C macro*), 1837
BT_MCS_PLAYING_ORDERS_SUPPORTED_OLDEST_ONCE (*C macro*), 1837
BT_MCS_PLAYING_ORDERS_SUPPORTED_OLDEST_REPEAT (*C macro*), 1837
BT_MCS_PLAYING_ORDERS_SUPPORTED_SHUFFLE_ONCE (*C macro*), 1837
BT_MCS_PLAYING_ORDERS_SUPPORTED_SHUFFLE_REPEAT (*C macro*), 1837
BT_MCS_PLAYING_ORDERS_SUPPORTED_SINGLE_ONCE (*C macro*), 1837
BT_MCS_PLAYING_ORDERS_SUPPORTED_SINGLE_REPEAT (*C macro*), 1837
BT_MCS_SCP_NTF_FAILURE (*C macro*), 1841
BT_MCS_SCP_NTF_SUCCESS (*C macro*), 1841
BT_MCS_SEARCH_TYPE_ALBUM_NAME (*C macro*), 1841
BT_MCS_SEARCH_TYPE_ARTIST_NAME (*C macro*), 1841
BT_MCS_SEARCH_TYPE_EARLIEST_YEAR (*C macro*), 1841
BT_MCS_SEARCH_TYPE_GENRE (*C macro*), 1841
BT_MCS_SEARCH_TYPE_GROUP_NAME (*C macro*), 1841
BT_MCS_SEARCH_TYPE_LATEST_YEAR (*C macro*), 1841
BT_MCS_SEARCH_TYPE_ONLY_GROUPS (*C macro*), 1841
BT_MCS_SEARCH_TYPE_ONLY_TRACKS (*C macro*), 1841
BT_MCS_SEARCH_TYPE_TRACK_NAME (*C macro*), 1841
BT_MCS_SEEKING_SPEED_FACTOR_MAX (*C macro*), 1836
BT_MCS_SEEKING_SPEED_FACTOR_MIN (*C macro*), 1836
BT_MCS_SEEKING_SPEED_FACTOR_ZERO (*C macro*), 1836
BT_MESH_ADDR_ALL_NODES (*C macro*), 2083
BT_MESH_ADDR_DFW_NODES (*C macro*), 2083
BT_MESH_ADDR_FRIENDS (*C macro*), 2083

BT_MESH_ADDR_IP_BR_ROUTERS (*C macro*), 2083
BT_MESH_ADDR_IP_NODES (*C macro*), 2083
BT_MESH_ADDR_IS_FIXED_GROUP (*C macro*), 2089
BT_MESH_ADDR_IS_GROUP (*C macro*), 2089
BT_MESH_ADDR_IS_RFU (*C macro*), 2089
BT_MESH_ADDR_IS_UNICAST (*C macro*), 2089
BT_MESH_ADDR_IS_VIRTUAL (*C macro*), 2089
BT_MESH_ADDR_PROXIES (*C macro*), 2083
BT_MESH_ADDR_RELAYS (*C macro*), 2083
BT_MESH_ADDR_UNASSIGNED (*C macro*), 2083
BT_MESH_APP_SEG_SDU_MAX (*C macro*), 2089
BT_MESH_APP_UNSEG_SDU_MAX (*C macro*), 2089
bt_mesh_auth_method_set_input (*C function*), 2245
bt_mesh_auth_method_set_none (*C function*), 2246
bt_mesh_auth_method_set_output (*C function*), 2245
bt_mesh_auth_method_set_static (*C function*), 2245
BT_MESH_BEACON_DISABLED (*C macro*), 2259
bt_mesh_beacon_enabled (*C function*), 2260
BT_MESH_BEACON_ENABLED (*C macro*), 2259
bt_mesh_beacon_set (*C function*), 2260
bt_mesh_blob_block (*C struct*), 2198
bt_mesh_blob_block.chunk_count (*C var*), 2198
bt_mesh_blob_block.missing (*C var*), 2198
bt_mesh_blob_block.number (*C var*), 2198
bt_mesh_blob_block.offset (*C var*), 2198
BT_MESH_BLOB_BLOCKS_MAX (*C macro*), 2181
bt_mesh_blob_block.size (*C var*), 2198
bt_mesh_blob_chunk (*C struct*), 2198
bt_mesh_blob_chunk.data (*C var*), 2198
bt_mesh_blob_chunk.offset (*C var*), 2198
bt_mesh_blob_chunk.size (*C var*), 2198
bt_mesh_blob_cli (*C struct*), 2192
bt_mesh_blob_cli_cancel (*C function*), 2188
bt_mesh_blob_cli_caps (*C struct*), 2190
bt_mesh_blob_cli_caps_get (*C function*), 2187
bt_mesh_blob_cli_caps.max_block_size_log (*C var*), 2191
bt_mesh_blob_cli_caps.max_chunk_size (*C var*), 2191
bt_mesh_blob_cli_caps.max_chunks (*C var*), 2191
bt_mesh_blob_cli_caps.max_size (*C var*), 2191
bt_mesh_blob_cli_caps.min_block_size_log (*C var*), 2191
bt_mesh_blob_cli_caps.modes (*C var*), 2191
bt_mesh_blob_cli_caps.mtu_size (*C var*), 2191
bt_mesh_blob_cli_cb (*C struct*), 2191
bt_mesh_blob_cli_cb.caps (*C var*), 2191
bt_mesh_blob_cli_cb.end (*C var*), 2192
bt_mesh_blob_cli_cb.lost_target (*C var*), 2191
bt_mesh_blob_cli_cb.suspended (*C var*), 2192
bt_mesh_blob_cli_cb.xfer_progress (*C var*), 2192
bt_mesh_blob_cli_cb.xfer_progress_complete (*C var*), 2192
bt_mesh_blob_cli_inputs (*C struct*), 2190
bt_mesh_blob_cli_inputs.app_idx (*C var*), 2190
bt_mesh_blob_cli_inputs.group (*C var*), 2190
bt_mesh_blob_cli_inputs.targets (*C var*), 2190
bt_mesh_blob_cli_inputs.timeout_base (*C var*), 2190
bt_mesh_blob_cli_inputs.ttl (*C var*), 2190
bt_mesh_blob_cli_is_busy (*C function*), 2188
bt_mesh_blob_cli_resume (*C function*), 2188
bt_mesh_blob_cli_send (*C function*), 2187

bt_mesh_blob_cli_state (*C enum*), 2186
bt_mesh_blob_cli_state.BT_MESH_BLOB_CLI_STATE_BLOCK_CHECK (*C enumerator*), 2186
bt_mesh_blob_cli_state.BT_MESH_BLOB_CLI_STATE_BLOCK_SEND (*C enumerator*), 2186
bt_mesh_blob_cli_state.BT_MESH_BLOB_CLI_STATE_BLOCK_START (*C enumerator*), 2186
bt_mesh_blob_cli_state.BT_MESH_BLOB_CLI_STATE_CANCEL (*C enumerator*), 2186
bt_mesh_blob_cli_state.BT_MESH_BLOB_CLI_STATE_CAPS_GET (*C enumerator*), 2186
bt_mesh_blob_cli_state.BT_MESH_BLOB_CLI_STATE_NONE (*C enumerator*), 2186
bt_mesh_blob_cli_state.BT_MESH_BLOB_CLI_STATE_START (*C enumerator*), 2186
bt_mesh_blob_cli_state.BT_MESH_BLOB_CLI_STATE_SUSPENDED (*C enumerator*), 2187
bt_mesh_blob_cli_state.BT_MESH_BLOB_CLI_STATE_XFER_CHECK (*C enumerator*), 2186
bt_mesh_blob_cli_state.BT_MESH_BLOB_CLI_STATE_XFER_PROGRESS_GET (*C enumerator*), 2187
bt_mesh_blob_cli_suspend (*C function*), 2188
bt_mesh_blob_cli_xfer_progress_active_get (*C function*), 2188
bt_mesh_blob_cli_xfer_progress_get (*C function*), 2188
bt_mesh_blob_cli.cb (*C var*), 2193
bt_mesh_blob_io (*C struct*), 2199
bt_mesh_blob_io_flash (*C struct*), 2194
bt_mesh_blob_io_flash_init (*C function*), 2194
bt_mesh_blob_io_flash.area_id (*C var*), 2195
bt_mesh_blob_io_flash.mode (*C var*), 2195
bt_mesh_blob_io_flash.offset (*C var*), 2195
bt_mesh_blob_io_mode (*C enum*), 2197
bt_mesh_blob_io_mode.BT_MESH_BLOB_READ (*C enumerator*), 2197
bt_mesh_blob_io_mode.BT_MESH_BLOB_WRITE (*C enumerator*), 2197
bt_mesh_blob_io.block_end (*C var*), 2199
bt_mesh_blob_io.block_start (*C var*), 2199
bt_mesh_blob_io.close (*C var*), 2199
bt_mesh_blob_io.open (*C var*), 2199
bt_mesh_blob_io.rd (*C var*), 2200
bt_mesh_blob_io.wr (*C var*), 2200
bt_mesh_blob_srv (*C struct*), 2184
bt_mesh_blob_srv_cancel (*C function*), 2182
bt_mesh_blob_srv_cb (*C struct*), 2182
bt_mesh_blob_srv_cb.end (*C var*), 2183
bt_mesh_blob_srv_cb.recover (*C var*), 2183
bt_mesh_blob_srv_cb.resume (*C var*), 2183
bt_mesh_blob_srv_cb.start (*C var*), 2182
bt_mesh_blob_srv_cb.suspended (*C var*), 2183
bt_mesh_blob_srv_is_busy (*C function*), 2182
bt_mesh_blob_srv_progress (*C function*), 2182
bt_mesh_blob_srv_recv (*C function*), 2181
bt_mesh_blob_srv.bt_mesh_blob_srv_state (*C struct*), 2184
bt_mesh_blob_srv.cb (*C var*), 2184
bt_mesh_blob_status (*C enum*), 2197
bt_mesh_blob_status.BT_MESH_BLOB_ERR_BLOB_TOO_LARGE (*C enumerator*), 2197
bt_mesh_blob_status.BT_MESH_BLOB_ERR_INFO_UNAVAILABLE (*C enumerator*), 2197
bt_mesh_blob_status.BT_MESH_BLOB_ERR_INTERNAL (*C enumerator*), 2197
bt_mesh_blob_status.BT_MESH_BLOB_ERR_INVALID_BLOCK_NUM (*C enumerator*), 2197
bt_mesh_blob_status.BT_MESH_BLOB_ERR_INVALID_BLOCK_SIZE (*C enumerator*), 2197
bt_mesh_blob_status.BT_MESH_BLOB_ERR_INVALID_CHUNK_SIZE (*C enumerator*), 2197
bt_mesh_blob_status.BT_MESH_BLOB_ERR_INVALID_PARAM (*C enumerator*), 2197
bt_mesh_blob_status.BT_MESH_BLOB_ERR_UNSUPPORTED_MODE (*C enumerator*), 2197
bt_mesh_blob_status.BT_MESH_BLOB_ERR_WRONG_BLOB_ID (*C enumerator*), 2197
bt_mesh_blob_status.BT_MESH_BLOB_ERR_WRONG_PHASE (*C enumerator*), 2197
bt_mesh_blob_status.BT_MESH_BLOB_SUCCESS (*C enumerator*), 2197
bt_mesh_blob_target (*C struct*), 2189
bt_mesh_blob_target_pull (*C struct*), 2188
bt_mesh_blob_target_pull.block_report_timestamp (*C var*), 2189

bt_mesh_blob_target_pull.missing (*C var*), 2189
bt_mesh_blob_target.addr (*C var*), 2189
bt_mesh_blob_target.n (*C var*), 2189
bt_mesh_blob_target.pull (*C var*), 2189
bt_mesh_blob_target.status (*C var*), 2189
bt_mesh_blob_xfer (*C struct*), 2198
bt_mesh_blob_xfer_info (*C struct*), 2189
bt_mesh_blob_xfer_info.block_size_log (*C var*), 2190
bt_mesh_blob_xfer_info.id (*C var*), 2190
bt_mesh_blob_xfer_info.missing_blocks (*C var*), 2190
bt_mesh_blob_xfer_info.mode (*C var*), 2189
bt_mesh_blob_xfer_info.mtu_size (*C var*), 2190
bt_mesh_blob_xfer_info.phase (*C var*), 2189
bt_mesh_blob_xfer_info.size (*C var*), 2190
bt_mesh_blob_xfer_info.status (*C var*), 2189
bt_mesh_blob_xfer_mode (*C enum*), 2196
bt_mesh_blob_xfer_mode.BT_MESH_BLOB_XFER_MODE_ALL (*C enumerator*), 2196
bt_mesh_blob_xfer_mode.BT_MESH_BLOB_XFER_MODE_NONE (*C enumerator*), 2196
bt_mesh_blob_xfer_mode.BT_MESH_BLOB_XFER_MODE_PULL (*C enumerator*), 2196
bt_mesh_blob_xfer_mode.BT_MESH_BLOB_XFER_MODE_PUSH (*C enumerator*), 2196
bt_mesh_blob_xfer_phase (*C enum*), 2196
bt_mesh_blob_xfer_phase.BT_MESH_BLOB_XFER_PHASE_COMPLETE (*C enumerator*), 2196
bt_mesh_blob_xfer_phase.BT_MESH_BLOB_XFER_PHASE_INACTIVE (*C enumerator*), 2196
bt_mesh_blob_xfer_phase.BT_MESH_BLOB_XFER_PHASE_SUSPENDED (*C enumerator*), 2196
bt_mesh_blob_xfer_phase.BT_MESH_BLOB_XFER_PHASE_WAITING_FOR_BLOCK (*C enumerator*), 2196
bt_mesh_blob_xfer_phase.BT_MESH_BLOB_XFER_PHASE_WAITING_FOR_CHUNK (*C enumerator*), 2196
bt_mesh_blob_xfer_phase.BT_MESH_BLOB_XFER_PHASE_WAITING_FOR_START (*C enumerator*), 2196
bt_mesh_blob_xfer.chunk_size (*C var*), 2199
bt_mesh_blob_xfer.id (*C var*), 2198
bt_mesh_blob_xfer.mode (*C var*), 2199
bt_mesh_blob_xfer.size (*C var*), 2198
bt_mesh_cfg_cli (*C struct*), 2136
bt_mesh_cfg_cli_app_key_add (*C function*), 2112
bt_mesh_cfg_cli_app_key_del (*C function*), 2113
bt_mesh_cfg_cli_app_key_get (*C function*), 2113
bt_mesh_cfg_cli_app_key_update (*C function*), 2126
bt_mesh_cfg_cli_beacon_get (*C function*), 2106
bt_mesh_cfg_cli_beacon_set (*C function*), 2107
bt_mesh_cfg_cli_cb (*C struct*), 2130
bt_mesh_cfg_cli_cb.app_key_list (*C var*), 2134
bt_mesh_cfg_cli_cb.app_key_status (*C var*), 2134
bt_mesh_cfg_cli_cb.beacon_status (*C var*), 2132
bt_mesh_cfg_cli_cb.comp_data (*C var*), 2130
bt_mesh_cfg_cli_cb.friend_status (*C var*), 2132
bt_mesh_cfg_cli_cb.gatt_proxy_status (*C var*), 2132
bt_mesh_cfg_cli_cb.hb_pub_status (*C var*), 2136
bt_mesh_cfg_cli_cb.hb_sub_status (*C var*), 2136
bt_mesh_cfg_cli_cb.krp_status (*C var*), 2136
bt_mesh_cfg_cli_cb.lpn_timeout_status (*C var*), 2135
bt_mesh_cfg_cli_cb.mod_app_list (*C var*), 2135
bt_mesh_cfg_cli_cb.mod_app_status (*C var*), 2134
bt_mesh_cfg_cli_cb.mod_pub_status (*C var*), 2130
bt_mesh_cfg_cli_cb.mod_sub_list (*C var*), 2131
bt_mesh_cfg_cli_cb.mod_sub_status (*C var*), 2131
bt_mesh_cfg_cli_cb.net_key_list (*C var*), 2133
bt_mesh_cfg_cli_cb.net_key_status (*C var*), 2133
bt_mesh_cfg_cli_cb.network_transmit_status (*C var*), 2133
bt_mesh_cfg_cli_cb.node_identity_status (*C var*), 2135

bt_mesh_cfg_cli_cb.node_reset_status (*C var*), 2132
bt_mesh_cfg_cli_cb.relay_status (*C var*), 2133
bt_mesh_cfg_cli_cb.ttl_status (*C var*), 2132
bt_mesh_cfg_cli_comp_data_get (*C function*), 2106
bt_mesh_cfg_cli_friend_get (*C function*), 2108
bt_mesh_cfg_cli_friend_set (*C function*), 2109
bt_mesh_cfg_cli_gatt_proxy_get (*C function*), 2109
bt_mesh_cfg_cli_gatt_proxy_set (*C function*), 2109
bt_mesh_cfg_cli_hb_pub (*C struct*), 2138
bt_mesh_cfg_cli_hb_pub_get (*C function*), 2124
bt_mesh_cfg_cli_hb_pub_set (*C function*), 2124
bt_mesh_cfg_cli_hb_pub.count (*C var*), 2138
bt_mesh_cfg_cli_hb_pub.dst (*C var*), 2138
bt_mesh_cfg_cli_hb_pub.feat (*C var*), 2139
bt_mesh_cfg_cli_hb_pub.net_idx (*C var*), 2139
bt_mesh_cfg_cli_hb_pub.period (*C var*), 2138
bt_mesh_cfg_cli_hb_pub.ttl (*C var*), 2139
bt_mesh_cfg_cli_hb_sub (*C struct*), 2137
bt_mesh_cfg_cli_hb_sub_get (*C function*), 2124
bt_mesh_cfg_cli_hb_sub_set (*C function*), 2123
bt_mesh_cfg_cli_hb_sub.count (*C var*), 2138
bt_mesh_cfg_cli_hb_sub.dst (*C var*), 2137
bt_mesh_cfg_cli_hb_sub.max (*C var*), 2138
bt_mesh_cfg_cli_hb_sub.min (*C var*), 2138
bt_mesh_cfg_cli_hb_sub.period (*C var*), 2138
bt_mesh_cfg_cli_hb_sub.src (*C var*), 2137
bt_mesh_cfg_cli_krp_get (*C function*), 2107
bt_mesh_cfg_cli_krp_set (*C function*), 2107
bt_mesh_cfg_cli_lpn_timeout_get (*C function*), 2127
bt_mesh_cfg_cli_mod_app_bind (*C function*), 2113
bt_mesh_cfg_cli_mod_app_bind_vnd (*C function*), 2114
bt_mesh_cfg_cli_mod_app_get (*C function*), 2115
bt_mesh_cfg_cli_mod_app_get_vnd (*C function*), 2115
bt_mesh_cfg_cli_mod_app_unbind (*C function*), 2114
bt_mesh_cfg_cli_mod_app_unbind_vnd (*C function*), 2115
bt_mesh_cfg_cli_mod_pub (*C struct*), 2136
bt_mesh_cfg_cli_mod_pub_get (*C function*), 2116
bt_mesh_cfg_cli_mod_pub_get_vnd (*C function*), 2116
bt_mesh_cfg_cli_mod_pub_set (*C function*), 2117
bt_mesh_cfg_cli_mod_pub_set_vnd (*C function*), 2117
bt_mesh_cfg_cli_mod_pub.addr (*C var*), 2137
bt_mesh_cfg_cli_mod_pub.app_idx (*C var*), 2137
bt_mesh_cfg_cli_mod_pub.cred_flag (*C var*), 2137
bt_mesh_cfg_cli_mod_pub.period (*C var*), 2137
bt_mesh_cfg_cli_mod_pub.transmit (*C var*), 2137
bt_mesh_cfg_cli_mod_pub.ttl (*C var*), 2137
bt_mesh_cfg_cli_mod_pub.uuid (*C var*), 2137
bt_mesh_cfg_cli_mod_sub_add (*C function*), 2118
bt_mesh_cfg_cli_mod_sub_add_vnd (*C function*), 2118
bt_mesh_cfg_cli_mod_sub_del (*C function*), 2118
bt_mesh_cfg_cli_mod_sub_del_all (*C function*), 2125
bt_mesh_cfg_cli_mod_sub_del_all_vnd (*C function*), 2125
bt_mesh_cfg_cli_mod_sub_del_vnd (*C function*), 2119
bt_mesh_cfg_cli_mod_sub_get (*C function*), 2122
bt_mesh_cfg_cli_mod_sub_get_vnd (*C function*), 2123
bt_mesh_cfg_cli_mod_sub_overwrite (*C function*), 2119
bt_mesh_cfg_cli_mod_sub_overwrite_vnd (*C function*), 2119
bt_mesh_cfg_cli_mod_sub_va_add (*C function*), 2120

bt_mesh_cfg_cli_mod_sub_va_add_vnd (*C function*), 2120
bt_mesh_cfg_cli_mod_sub_va_del (*C function*), 2121
bt_mesh_cfg_cli_mod_sub_va_del_vnd (*C function*), 2121
bt_mesh_cfg_cli_mod_sub_va_overwrite (*C function*), 2122
bt_mesh_cfg_cli_mod_sub_va_overwrite_vnd (*C function*), 2122
bt_mesh_cfg_cli_net_key_add (*C function*), 2111
bt_mesh_cfg_cli_net_key_del (*C function*), 2112
bt_mesh_cfg_cli_net_key_get (*C function*), 2112
bt_mesh_cfg_cli_net_key_update (*C function*), 2125
bt_mesh_cfg_cli_net_transmit_get (*C function*), 2110
bt_mesh_cfg_cli_net_transmit_set (*C function*), 2110
bt_mesh_cfg_cli_node_identity_get (*C function*), 2127
bt_mesh_cfg_cli_node_identity_set (*C function*), 2126
bt_mesh_cfg_cli_node_reset (*C function*), 2106
bt_mesh_cfg_cli_relay_get (*C function*), 2110
bt_mesh_cfg_cli_relay_set (*C function*), 2111
bt_mesh_cfg_cli_timeout_get (*C function*), 2127
bt_mesh_cfg_cli_timeout_set (*C function*), 2127
bt_mesh_cfg_cli_ttl_get (*C function*), 2108
bt_mesh_cfg_cli_ttl_set (*C function*), 2108
bt_mesh_cfg_cli.cb (*C var*), 2136
bt_mesh_cfg_cli.model (*C var*), 2136
bt_mesh_comp (*C struct*), 2103
bt_mesh_comp2 (*C struct*), 2104
bt_mesh_comp2_record (*C struct*), 2104
bt_mesh_comp2_record.data (*C var*), 2104
bt_mesh_comp2_record.data_len (*C var*), 2104
bt_mesh_comp2_record.elem_offset (*C var*), 2104
bt_mesh_comp2_record.elem_offset_cnt (*C var*), 2104
bt_mesh_comp2_record.id (*C var*), 2104
bt_mesh_comp2_record.version (*C var*), 2104
bt_mesh_comp2_record.x (*C var*), 2104
bt_mesh_comp2_record.y (*C var*), 2104
bt_mesh_comp2_record.z (*C var*), 2104
bt_mesh_comp2_register (*C function*), 2097
bt_mesh_comp2.record (*C var*), 2105
bt_mesh_comp2.record_cnt (*C var*), 2104
bt_mesh_comp_change_prepare (*C function*), 2097
bt_mesh_comp_p0 (*C struct*), 2139
bt_mesh_comp_p0_elem (*C struct*), 2139
bt_mesh_comp_p0_elem_mod (*C function*), 2128
bt_mesh_comp_p0_elem_mod_vnd (*C function*), 2128
bt_mesh_comp_p0_elem_pull (*C function*), 2128
bt_mesh_comp_p0_elem.loc (*C var*), 2139
bt_mesh_comp_p0_elem.nsig (*C var*), 2139
bt_mesh_comp_p0_elem.nvnd (*C var*), 2139
bt_mesh_comp_p0_get (*C function*), 2127
bt_mesh_comp_p0.cid (*C var*), 2139
bt_mesh_comp_p0.crpl (*C var*), 2139
bt_mesh_comp_p0.feat (*C var*), 2139
bt_mesh_comp_p0.pid (*C var*), 2139
bt_mesh_comp_p0.vid (*C var*), 2139
bt_mesh_comp_p1_elem (*C struct*), 2140
bt_mesh_comp_p1_elem_pull (*C function*), 2129
bt_mesh_comp_p1_elem.nsig (*C var*), 2140
bt_mesh_comp_p1_elem.nvnd (*C var*), 2140
bt_mesh_comp_p1_ext_item (*C struct*), 2141
bt_mesh_comp_p1_ext_item.long_item (*C var*), 2141

bt_mesh_comp_p1_ext_item.short_item (*C var*), 2141
bt_mesh_comp_p1_item_long (*C struct*), 2140
bt_mesh_comp_p1_item_long.elem_offset (*C var*), 2141
bt_mesh_comp_p1_item_long.mod_item_idx (*C var*), 2141
bt_mesh_comp_p1_item_pull (*C function*), 2129
bt_mesh_comp_p1_item_short (*C struct*), 2140
bt_mesh_comp_p1_item_short.elem_offset (*C var*), 2140
bt_mesh_comp_p1_item_short.mod_item_idx (*C var*), 2140
bt_mesh_comp_p1_model_item (*C struct*), 2140
bt_mesh_comp_p1_model_item.cor_id (*C var*), 2140
bt_mesh_comp_p1_model_item.cor_present (*C var*), 2140
bt_mesh_comp_p1_model_item.ext_item_cnt (*C var*), 2140
bt_mesh_comp_p1_model_item.format (*C var*), 2140
bt_mesh_comp_p1_pull_ext_item (*C function*), 2129
bt_mesh_comp_p2_record (*C struct*), 2141
bt_mesh_comp_p2_record_pull (*C function*), 2129
bt_mesh_comp_p2_record.data_buf (*C var*), 2141
bt_mesh_comp_p2_record.elem_buf (*C var*), 2141
bt_mesh_comp_p2_record.id (*C var*), 2141
bt_mesh_comp_p2_record.version (*C var*), 2141
bt_mesh_comp_p2_record.x (*C var*), 2141
bt_mesh_comp_p2_record.y (*C var*), 2141
bt_mesh_comp_p2_record.z (*C var*), 2141
bt_mesh_comp.cid (*C var*), 2103
bt_mesh_comp.elem (*C var*), 2104
bt_mesh_comp.elem_count (*C var*), 2104
bt_mesh_comp.pid (*C var*), 2103
bt_mesh_comp.vid (*C var*), 2103
bt_mesh_default_ttl_get (*C function*), 2261
bt_mesh_default_ttl_set (*C function*), 2261
bt_mesh_dev_capabilities (*C struct*), 2248
bt_mesh_dev_capabilities.algorithms (*C var*), 2248
bt_mesh_dev_capabilities.elem_count (*C var*), 2248
bt_mesh_dev_capabilities.input_actions (*C var*), 2249
bt_mesh_dev_capabilities.input_size (*C var*), 2249
bt_mesh_dev_capabilities.oob_type (*C var*), 2249
bt_mesh_dev_capabilities.output_actions (*C var*), 2249
bt_mesh_dev_capabilities.output_size (*C var*), 2249
bt_mesh_dev_capabilities.pub_key_type (*C var*), 2248
bt_mesh_dfd_phase (*C enum*), 2224
bt_mesh_dfd_phase.BT_MESH_DFD_PHASE_APPLYING_UPDATE (*C enumerator*), 2224
bt_mesh_dfd_phase.BT_MESH_DFD_PHASE_CANCELING_UPDATE (*C enumerator*), 2225
bt_mesh_dfd_phase.BT_MESH_DFD_PHASE_COMPLETED (*C enumerator*), 2225
bt_mesh_dfd_phase.BT_MESH_DFD_PHASE_FAILED (*C enumerator*), 2225
bt_mesh_dfd_phase.BT_MESH_DFD_PHASE_IDLE (*C enumerator*), 2224
bt_mesh_dfd_phase.BT_MESH_DFD_PHASE_TRANSFER_ACTIVE (*C enumerator*), 2224
bt_mesh_dfd_phase.BT_MESH_DFD_PHASE_TRANSFER_SUCCESS (*C enumerator*), 2224
bt_mesh_dfd_phase.BT_MESH_DFD_PHASE_TRANSFER_SUSPENDED (*C enumerator*), 2225
bt_mesh_dfd_srv (*C struct*), 2216
bt_mesh_dfd_srv_cb (*C struct*), 2215
bt_mesh_dfd_srv_cb.del (*C var*), 2216
bt_mesh_dfd_srv_cb.phase (*C var*), 2216
bt_mesh_dfd_srv_cb.recv (*C var*), 2216
bt_mesh_dfd_srv_cb.send (*C var*), 2216
BT_MESH_DFD_SRV_INIT (*C macro*), 2215
bt_mesh_dfd_status (*C enum*), 2223
bt_mesh_dfd_status.BT_MESH_DFD_ERR_BUSY_WITH_DISTRIBUTION (*C enumerator*), 2224
bt_mesh_dfd_status.BT_MESH_DFD_ERR_BUSY_WITH_UPLOAD (*C enumerator*), 2224

bt_mesh_dfd_status.BT_MESH_DFD_ERR_FW_NOT_FOUND (*C enumerator*), 2224
bt_mesh_dfd_status.BT_MESH_DFD_ERR_INSUFFICIENT_RESOURCES (*C enumerator*), 2223
bt_mesh_dfd_status.BT_MESH_DFD_ERR_INTERNAL (*C enumerator*), 2224
bt_mesh_dfd_status.BT_MESH_DFD_ERR_INVALID_APPKEY_INDEX (*C enumerator*), 2224
bt_mesh_dfd_status.BT_MESH_DFD_ERR_NEW_FW_NOT_AVAILABLE (*C enumerator*), 2224
bt_mesh_dfd_status.BT_MESH_DFD_ERR_RECEIVERS_LIST_EMPTY (*C enumerator*), 2224
bt_mesh_dfd_status.BT_MESH_DFD_ERR_SUSPEND_FAILED (*C enumerator*), 2224
bt_mesh_dfd_status.BT_MESH_DFD_ERR_URI_MALFORMED (*C enumerator*), 2224
bt_mesh_dfd_status.BT_MESH_DFD_ERR_URI_NOT_SUPPORTED (*C enumerator*), 2224
bt_mesh_dfd_status.BT_MESH_DFD_ERR_URI_UNREACHABLE (*C enumerator*), 2224
bt_mesh_dfd_status.BT_MESH_DFD_ERR_WRONG_PHASE (*C enumerator*), 2223
bt_mesh_dfd_status.BT_MESH_DFD_SUCCESS (*C enumerator*), 2223
bt_mesh_dfd_upload_phase (*C enum*), 2225
bt_mesh_dfd_upload_phase.BT_MESH_DFD_UPLOAD_PHASE_IDLE (*C enumerator*), 2225
bt_mesh_dfd_upload_phase.BT_MESH_DFD_UPLOAD_PHASE_TRANSFER_ACTIVE (*C enumerator*), 2225
bt_mesh_dfd_upload_phase.BT_MESH_DFD_UPLOAD_PHASE_TRANSFER_ERROR (*C enumerator*), 2225
bt_mesh_dfd_upload_phase.BT_MESH_DFD_UPLOAD_PHASE_TRANSFER_SUCCESS (*C enumerator*), 2225
bt_mesh_dfu_cli (*C struct*), 2214
bt_mesh_dfu_cli_apply (*C function*), 2209
bt_mesh_dfu_cli_cancel (*C function*), 2209
bt_mesh_dfu_cli_cb (*C struct*), 2213
bt_mesh_dfu_cli_cb.applied (*C var*), 2213
bt_mesh_dfu_cli_cb.confirmed (*C var*), 2213
bt_mesh_dfu_cli_cb.ended (*C var*), 2213
bt_mesh_dfu_cli_cb.lost_target (*C var*), 2213
bt_mesh_dfu_cli_cb.suspended (*C var*), 2213
bt_mesh_dfu_cli_confirm (*C function*), 2210
bt_mesh_dfu_cli_imgs_get (*C function*), 2210
BT_MESH_DFU_CLI_INIT (*C macro*), 2207
bt_mesh_dfu_cli_is_busy (*C function*), 2210
bt_mesh_dfu_cli_metadata_check (*C function*), 2210
bt_mesh_dfu_cli_progress (*C function*), 2210
bt_mesh_dfu_cli_resume (*C function*), 2209
bt_mesh_dfu_cli_send (*C function*), 2208
bt_mesh_dfu_cli_status_get (*C function*), 2211
bt_mesh_dfu_cli_suspend (*C function*), 2209
bt_mesh_dfu_cli_timeout_get (*C function*), 2211
bt_mesh_dfu_cli_timeout_set (*C function*), 2211
bt_mesh_dfu_cli_xfer (*C struct*), 2214
bt_mesh_dfu_cli_xfer_blob_params (*C struct*), 2214
bt_mesh_dfu_cli_xfer_blob_params.chunk_size (*C var*), 2214
bt_mesh_dfu_cli_xfer.blob_id (*C var*), 2214
bt_mesh_dfu_cli_xfer.blob_params (*C var*), 2214
bt_mesh_dfu_cli_xfer.mode (*C var*), 2214
bt_mesh_dfu_cli_xfer.slot (*C var*), 2214
bt_mesh_dfu_cli.blob (*C var*), 2214
bt_mesh_dfu_cli.cb (*C var*), 2214
bt_mesh_dfu_effect (*C enum*), 2227
bt_mesh_dfu_effect.BT_MESH_DFU_EFFECT_COMP_CHANGE (*C enumerator*), 2227
bt_mesh_dfu_effect.BT_MESH_DFU_EFFECT_COMP_CHANGE_NO_RPR (*C enumerator*), 2227
bt_mesh_dfu_effect.BT_MESH_DFU_EFFECT_NONE (*C enumerator*), 2227
bt_mesh_dfu_effect.BT_MESH_DFU_EFFECT_UNPROV (*C enumerator*), 2227
bt_mesh_dfu_img (*C struct*), 2227
bt_mesh_dfu_img_cb_t (*C type*), 2208
bt_mesh_dfu_img.fwid (*C var*), 2228
bt_mesh_dfu_img.fwid_len (*C var*), 2228
bt_mesh_dfu_img.uri (*C var*), 2228

bt_mesh_dfu_iter (*C enum*), 2227
bt_mesh_dfu_iter.BT_MESH_DFU_ITER_CONTINUE (*C enumerator*), 2227
bt_mesh_dfu_iter.BT_MESH_DFU_ITER_STOP (*C enumerator*), 2227
bt_mesh_dfu_metadata (*C struct*), 2230
bt_mesh_dfu_metadata_comp_hash_get (*C function*), 2229
bt_mesh_dfu_metadata_comp_hash_local_get (*C function*), 2229
bt_mesh_dfu_metadata_decode (*C function*), 2229
bt_mesh_dfu_metadata_encode (*C function*), 2229
bt_mesh_dfu_metadata_fw_core_type (*C enum*), 2228
bt_mesh_dfu_metadata_fw_core_type.BT_MESH_DFU_FW_CORE_TYPE_APP (*C enumerator*), 2228
bt_mesh_dfu_metadata_fw_core_type.BT_MESH_DFU_FW_CORE_TYPE_APP_SPECIFIC_BLOB (*C enumerator*), 2229
bt_mesh_dfu_metadata_fw_core_type.BT_MESH_DFU_FW_CORE_TYPE_NETWORK (*C enumerator*), 2229
bt_mesh_dfu_metadata_fw_ver (*C struct*), 2230
bt_mesh_dfu_metadata_fw_ver.build_num (*C var*), 2230
bt_mesh_dfu_metadata_fw_ver.major (*C var*), 2230
bt_mesh_dfu_metadata_fw_ver.minor (*C var*), 2230
bt_mesh_dfu_metadata_fw_ver.revision (*C var*), 2230
bt_mesh_dfu_metadata_status (*C struct*), 2212
bt_mesh_dfu_metadata_status.effect (*C var*), 2212
bt_mesh_dfu_metadata_status.idx (*C var*), 2212
bt_mesh_dfu_metadata_status.status (*C var*), 2212
bt_mesh_dfu_metadata.comp_hash (*C var*), 2230
bt_mesh_dfu_metadata.elems (*C var*), 2230
bt_mesh_dfu_metadata.fw_core_type (*C var*), 2230
bt_mesh_dfu_metadata.fw_size (*C var*), 2230
bt_mesh_dfu_metadata.fw_ver (*C var*), 2230
bt_mesh_dfu_metadata.user_data (*C var*), 2230
bt_mesh_dfu_metadata.user_data_len (*C var*), 2230
bt_mesh_dfu_phase (*C enum*), 2226
bt_mesh_dfu_phase.BT_MESH_DFU_PHASE_APPLY_FAIL (*C enumerator*), 2226
bt_mesh_dfu_phase.BT_MESH_DFU_PHASE_APPLY_SUCCESS (*C enumerator*), 2226
bt_mesh_dfu_phase.BT_MESH_DFU_PHASE_APPLYING (*C enumerator*), 2226
bt_mesh_dfu_phase.BT_MESH_DFU_PHASE_IDLE (*C enumerator*), 2226
bt_mesh_dfu_phase.BT_MESH_DFU_PHASE_TRANSFER_ACTIVE (*C enumerator*), 2226
bt_mesh_dfu_phase.BT_MESH_DFU_PHASE_TRANSFER_CANCELED (*C enumerator*), 2226
bt_mesh_dfu_phase.BT_MESH_DFU_PHASE_TRANSFER_ERR (*C enumerator*), 2226
bt_mesh_dfu_phase.BT_MESH_DFU_PHASE_UNKNOWN (*C enumerator*), 2226
bt_mesh_dfu_phase.BT_MESH_DFU_PHASE_VERIFY (*C enumerator*), 2226
bt_mesh_dfu_phase.BT_MESH_DFU_PHASE_VERIFY_FAIL (*C enumerator*), 2226
bt_mesh_dfu_phase.BT_MESH_DFU_PHASE_VERIFY_OK (*C enumerator*), 2226
bt_mesh_dfu_slot (*C struct*), 2228
bt_mesh_dfu_slot.fwid (*C var*), 2228
bt_mesh_dfu_slot.fwid_len (*C var*), 2228
bt_mesh_dfu_slot.metadata (*C var*), 2228
bt_mesh_dfu_slot.metadata_len (*C var*), 2228
bt_mesh_dfu_slot.size (*C var*), 2228
bt_mesh_dfu_srv (*C struct*), 2207
bt_mesh_dfu_srv_applied (*C function*), 2204
bt_mesh_dfu_srv_cancel (*C function*), 2204
bt_mesh_dfu_srv_cb (*C struct*), 2205
bt_mesh_dfu_srv_cb.apply (*C var*), 2206
bt_mesh_dfu_srv_cb.check (*C var*), 2205
bt_mesh_dfu_srv_cb.end (*C var*), 2206
bt_mesh_dfu_srv_cb.recover (*C var*), 2206
bt_mesh_dfu_srv_cb.start (*C var*), 2205
BT_MESH_DFU_SRV_INIT (*C macro*), 2204

bt_mesh_dfu_srv_is_busy (*C function*), 2204
bt_mesh_dfu_srv_progress (*C function*), 2205
bt_mesh_dfu_srv_rejected (*C function*), 2204
bt_mesh_dfu_srv_verified (*C function*), 2204
bt_mesh_dfu_srv.blob (*C var*), 2207
bt_mesh_dfu_srv.cb (*C var*), 2207
bt_mesh_dfu_srv.img_count (*C var*), 2207
bt_mesh_dfu_srv.imgs (*C var*), 2207
bt_mesh_dfu_status (*C enum*), 2226
bt_mesh_dfu_status.BT_MESH_DFU_ERR_BLOB_XFER_BUSY (*C enumerator*), 2227
bt_mesh_dfu_status.BT_MESH_DFU_ERR_FW_IDX (*C enumerator*), 2227
bt_mesh_dfu_status.BT_MESH_DFU_ERR_INTERNAL (*C enumerator*), 2227
bt_mesh_dfu_status.BT_MESH_DFU_ERR_METADATA (*C enumerator*), 2227
bt_mesh_dfu_status.BT_MESH_DFU_ERR_RESOURCES (*C enumerator*), 2226
bt_mesh_dfu_status.BT_MESH_DFU_ERR_TEMPORARILY_UNAVAILABLE (*C enumerator*), 2227
bt_mesh_dfu_status.BT_MESH_DFU_ERR_WRONG_PHASE (*C enumerator*), 2226
bt_mesh_dfu_status.BT_MESH_DFU_SUCCESS (*C enumerator*), 2226
bt_mesh_dfu_target (*C struct*), 2211
bt_mesh_dfu_target_status (*C struct*), 2212
bt_mesh_dfu_target_status.blob_id (*C var*), 2212
bt_mesh_dfu_target_status.effect (*C var*), 2212
bt_mesh_dfu_target_status.img_idx (*C var*), 2212
bt_mesh_dfu_target_status.phase (*C var*), 2212
bt_mesh_dfu_target_status.status (*C var*), 2212
bt_mesh_dfu_target_status.timeout_base (*C var*), 2212
bt_mesh_dfu_target_status.ttl (*C var*), 2212
bt_mesh_dfu_target.blob (*C var*), 2211
bt_mesh_dfu_target.effect (*C var*), 2211
bt_mesh_dfu_target.img_idx (*C var*), 2211
bt_mesh_dfu_target.phase (*C var*), 2212
bt_mesh_dfu_target.status (*C var*), 2212
BT_MESH_ELEM (*C macro*), 2089
bt_mesh_elem (*C struct*), 2098
bt_mesh_elem.bt_mesh_elem_rt_ctx (*C struct*), 2098
bt_mesh_elem.bt_mesh_elem_rt_ctx.addr (*C var*), 2098
bt_mesh_elem.loc (*C var*), 2098
bt_mesh_elem.model_count (*C var*), 2098
bt_mesh_elem.models (*C var*), 2098
bt_mesh_elem.vnd_model_count (*C var*), 2098
bt_mesh_elem.vnd_models (*C var*), 2098
BT_MESH_FEAT_FRIEND (*C macro*), 2075
BT_MESH_FEAT_LOW_POWER (*C macro*), 2075
BT_MESH_FEAT_PROXY (*C macro*), 2075
BT_MESH_FEAT_RELAY (*C macro*), 2075
bt_mesh_feat_state (*C enum*), 2259
bt_mesh_feat_state.BT_MESH_FEATURE_DISABLED (*C enumerator*), 2259
bt_mesh_feat_state.BT_MESH_FEATURE_ENABLED (*C enumerator*), 2259
bt_mesh_feat_state.BT_MESH_FEATURE_NOT_SUPPORTED (*C enumerator*), 2260
BT_MESH_FEAT_SUPPORTED (*C macro*), 2075
bt_mesh_friend_cb (*C struct*), 2078
BT_MESH_FRIEND_CB_DEFINE (*C macro*), 2075
bt_mesh_friend_cb.established (*C var*), 2079
bt_mesh_friend_cb.polled (*C var*), 2079
bt_mesh_friend_cb.terminated (*C var*), 2079
BT_MESH_FRIEND_DISABLED (*C macro*), 2259
BT_MESH_FRIEND_ENABLED (*C macro*), 2259
bt_mesh_friend_get (*C function*), 2264
BT_MESH_FRIEND_NOT_SUPPORTED (*C macro*), 2259

bt_mesh_friend_set (*C function*), 2264
bt_mesh_friend_terminate (*C function*), 2077
BT_MESH_GATT_PROXY_DISABLED (*C macro*), 2259
BT_MESH_GATT_PROXY_ENABLED (*C macro*), 2259
bt_mesh_gatt_proxy_get (*C function*), 2263
BT_MESH_GATT_PROXY_NOT_SUPPORTED (*C macro*), 2259
bt_mesh_gatt_proxy_set (*C function*), 2263
bt_mesh_hb_cb (*C struct*), 2257
BT_MESH_HB_CB_DEFINE (*C macro*), 2256
bt_mesh_hb_cb.pub_sent (*C var*), 2258
bt_mesh_hb_cb.recv (*C var*), 2257
bt_mesh_hb_cb.sub_end (*C var*), 2258
bt_mesh_hb_pub (*C struct*), 2256
bt_mesh_hb_pub_get (*C function*), 2256
bt_mesh_hb_pub.count (*C var*), 2256
bt_mesh_hb_pub.dst (*C var*), 2256
bt_mesh_hb_pub.feat (*C var*), 2256
bt_mesh_hb_pub.net_idx (*C var*), 2256
bt_mesh_hb_pub.period (*C var*), 2256
bt_mesh_hb_pub.ttl (*C var*), 2256
bt_mesh_hb_sub (*C struct*), 2257
bt_mesh_hb_sub_get (*C function*), 2256
bt_mesh_hb_sub.count (*C var*), 2257
bt_mesh_hb_sub.dst (*C var*), 2257
bt_mesh_hb_sub.max_hops (*C var*), 2257
bt_mesh_hb_sub.min_hops (*C var*), 2257
bt_mesh_hb_sub.period (*C var*), 2257
bt_mesh_hb_sub.remaining (*C var*), 2257
bt_mesh_hb_sub.src (*C var*), 2257
bt_mesh_health_cli (*C struct*), 2147
bt_mesh_health_cli_attention_get (*C function*), 2146
bt_mesh_health_cli_attention_set (*C function*), 2146
bt_mesh_health_cli_attention_set_unack (*C function*), 2147
bt_mesh_health_cli_fault_clear (*C function*), 2143
bt_mesh_health_cli_fault_clear_unack (*C function*), 2144
bt_mesh_health_cli_fault_get (*C function*), 2142
bt_mesh_health_cli_fault_test (*C function*), 2144
bt_mesh_health_cli_fault_test_unack (*C function*), 2144
bt_mesh_health_cli_period_get (*C function*), 2145
bt_mesh_health_cli_period_set (*C function*), 2145
bt_mesh_health_cli_period_set_unack (*C function*), 2146
bt_mesh_health_cli_timeout_get (*C function*), 2147
bt_mesh_health_cli_timeout_set (*C function*), 2147
bt_mesh_health_cli.attention_status (*C var*), 2148
bt_mesh_health_cli.current_status (*C var*), 2148
bt_mesh_health_cli.fault_status (*C var*), 2148
bt_mesh_health_cli.model (*C var*), 2147
bt_mesh_health_cli.period_status (*C var*), 2147
bt_mesh_health_cli.pub (*C var*), 2147
bt_mesh_health_cli.pub_buf (*C var*), 2147
bt_mesh_health_cli.pub_data (*C var*), 2147
BT_MESH_HEALTH_FAULT_ACTUATOR_BLOCKED_ERROR (*C macro*), 2154
BT_MESH_HEALTH_FAULT_ACTUATOR_BLOCKED_WARNING (*C macro*), 2154
BT_MESH_HEALTH_FAULT_BATTERY_LOW_ERROR (*C macro*), 2153
BT_MESH_HEALTH_FAULT_BATTERY_LOW_WARNING (*C macro*), 2153
BT_MESH_HEALTH_FAULT_CONDENSATION_ERROR (*C macro*), 2153
BT_MESH_HEALTH_FAULT_CONDENSATION_WARNING (*C macro*), 2153
BT_MESH_HEALTH_FAULT_CONFIGURATION_ERROR (*C macro*), 2154

BT_MESH_HEALTH_FAULT_CONFIGURATION_WARNING (*C macro*), 2153
BT_MESH_HEALTH_FAULT_DEVICE_DROPPED_ERROR (*C macro*), 2154
BT_MESH_HEALTH_FAULT_DEVICE_DROPPED_WARNING (*C macro*), 2154
BT_MESH_HEALTH_FAULT_DEVICE_MOVED_ERROR (*C macro*), 2154
BT_MESH_HEALTH_FAULT_DEVICE_MOVED_WARNING (*C macro*), 2154
BT_MESH_HEALTH_FAULT_ELEMENT_NOT_CALIBRATED_ERROR (*C macro*), 2154
BT_MESH_HEALTH_FAULT_ELEMENT_NOT_CALIBRATED_WARNING (*C macro*), 2154
BT_MESH_HEALTH_FAULT_EMPTY_ERROR (*C macro*), 2155
BT_MESH_HEALTH_FAULT_EMPTY_WARNING (*C macro*), 2155
BT_MESH_HEALTH_FAULT_HOUSING_OPENED_ERROR (*C macro*), 2154
BT_MESH_HEALTH_FAULT_HOUSING_OPENED_WARNING (*C macro*), 2154
BT_MESH_HEALTH_FAULT_INPUT_NO_CHANGE_ERROR (*C macro*), 2154
BT_MESH_HEALTH_FAULT_INPUT_NO_CHANGE_WARNING (*C macro*), 2154
BT_MESH_HEALTH_FAULT_INPUT_TOO_HIGH_ERROR (*C macro*), 2154
BT_MESH_HEALTH_FAULT_INPUT_TOO_HIGH_WARNING (*C macro*), 2154
BT_MESH_HEALTH_FAULT_INPUT_TOO_LOW_ERROR (*C macro*), 2154
BT_MESH_HEALTH_FAULT_INPUT_TOO_LOW_WARNING (*C macro*), 2154
BT_MESH_HEALTH_FAULT_INTERNAL_BUS_ERROR (*C macro*), 2155
BT_MESH_HEALTH_FAULT_INTERNAL_BUS_WARNING (*C macro*), 2155
BT_MESH_HEALTH_FAULT_MECHANISM_JAMMED_ERROR (*C macro*), 2155
BT_MESH_HEALTH_FAULT_MECHANISM_JAMMED_WARNING (*C macro*), 2155
BT_MESH_HEALTH_FAULT_MEMORY_ERROR (*C macro*), 2154
BT_MESH_HEALTH_FAULT_MEMORY_WARNING (*C macro*), 2154
BT_MESH_HEALTH_FAULT_NO_FAULT (*C macro*), 2153
BT_MESH_HEALTH_FAULT_NO_LOAD_ERROR (*C macro*), 2153
BT_MESH_HEALTH_FAULT_NO_LOAD_WARNING (*C macro*), 2153
BT_MESH_HEALTH_FAULT_OVERFLOW_ERROR (*C macro*), 2155
BT_MESH_HEALTH_FAULT_OVERFLOW_WARNING (*C macro*), 2155
BT_MESH_HEALTH_FAULT_OVERHEAT_ERROR (*C macro*), 2153
BT_MESH_HEALTH_FAULT_OVERHEAT_WARNING (*C macro*), 2153
BT_MESH_HEALTH_FAULT_OVERLOAD_ERROR (*C macro*), 2153
BT_MESH_HEALTH_FAULT_OVERLOAD_WARNING (*C macro*), 2153
BT_MESH_HEALTH_FAULT_POWER_SUPPLY_INTERRUPTED_ERROR (*C macro*), 2153
BT_MESH_HEALTH_FAULT_POWER_SUPPLY_INTERRUPTED_WARNING (*C macro*), 2153
BT_MESH_HEALTH_FAULT_SELF_TEST_ERROR (*C macro*), 2154
BT_MESH_HEALTH_FAULT_SELF_TEST_WARNING (*C macro*), 2154
BT_MESH_HEALTH_FAULT_SUPPLY_VOLTAGE_TOO_HIGH_ERROR (*C macro*), 2153
BT_MESH_HEALTH_FAULT_SUPPLY_VOLTAGE_TOO_HIGH_WARNING (*C macro*), 2153
BT_MESH_HEALTH_FAULT_SUPPLY_VOLTAGE_TOO_LOW_ERROR (*C macro*), 2153
BT_MESH_HEALTH_FAULT_SUPPLY_VOLTAGE_TOO_LOW_WARNING (*C macro*), 2153
BT_MESH_HEALTH_FAULT_TAMPER_ERROR (*C macro*), 2154
BT_MESH_HEALTH_FAULT_TAMPER_WARNING (*C macro*), 2154
BT_MESH_HEALTH_FAULT_VENDOR_SPECIFIC_START (*C macro*), 2155
BT_MESH_HEALTH_FAULT_VIBRATION_ERROR (*C macro*), 2153
BT_MESH_HEALTH_FAULT_VIBRATION_WARNING (*C macro*), 2153
BT_MESH_HEALTH_PUB_DEFINE (*C macro*), 2149
bt_mesh_health_srv (*C struct*), 2152
bt_mesh_health_srv_cb (*C struct*), 2150
bt_mesh_health_srv_cb.attn_off (*C var*), 2152
bt_mesh_health_srv_cb.attn_on (*C var*), 2152
bt_mesh_health_srv_cb.fault_clear (*C var*), 2151
bt_mesh_health_srv_cb.fault_get_cur (*C var*), 2150
bt_mesh_health_srv_cb.fault_get_reg (*C var*), 2151
bt_mesh_health_srv_cb.fault_test (*C var*), 2152
bt_mesh_health_srv_fault_update (*C function*), 2150
bt_mesh_health_srv.attn_timer (*C var*), 2152
bt_mesh_health_srv.cb (*C var*), 2152
bt_mesh_health_srv.model (*C var*), 2152

BT_MESH_HEALTH_TEST_INFO (*C macro*), 2150
BT_MESH_HEALTH_TEST_INFO_METADATA (*C macro*), 2150
BT_MESH_HEALTH_TEST_INFO_METADATA_ID (*C macro*), 2150
bt_mesh_init (*C function*), 2076
bt_mesh_input_action_t (*C enum*), 2242
bt_mesh_input_action_t.BT_MESH_ENTER_NUMBER (*C enumerator*), 2243
bt_mesh_input_action_t.BT_MESH_ENTER_STRING (*C enumerator*), 2243
bt_mesh_input_action_t.BT_MESH_NO_INPUT (*C enumerator*), 2243
bt_mesh_input_action_t.BT_MESH_PUSH (*C enumerator*), 2243
bt_mesh_input_action_t.BT_MESH_TWIST (*C enumerator*), 2243
bt_mesh_input_number (*C function*), 2244
bt_mesh_input_string (*C function*), 2244
BT_MESH_IS_DEV_KEY (*C macro*), 2089
bt_mesh_is_provisioned (*C function*), 2248
bt_mesh_iv_update (*C function*), 2076
bt_mesh_iv_update_test (*C function*), 2076
BT_MESH_KEY_ANY (*C macro*), 2083
BT_MESH_KEY_DEV (*C macro*), 2083
BT_MESH_KEY_DEV_ANY (*C macro*), 2083
BT_MESH_KEY_DEV_LOCAL (*C macro*), 2083
BT_MESH_KEY_DEV_REMOTE (*C macro*), 2083
bt_mesh_key_idx_unpack_list (*C function*), 2130
BT_MESH_KEY_UNUSED (*C macro*), 2083
BT_MESH_KR_NORMAL (*C macro*), 2258
BT_MESH_KR_PHASE_1 (*C macro*), 2258
BT_MESH_KR_PHASE_2 (*C macro*), 2258
BT_MESH_KR_PHASE_3 (*C macro*), 2258
bt_mesh_large_comp_data_cli (*C struct*), 2157
bt_mesh_large_comp_data_cli_cb (*C struct*), 2157
bt_mesh_large_comp_data_cli_cb.large_comp_data_status (*C var*), 2157
bt_mesh_large_comp_data_cli_cb.models_metadata_status (*C var*), 2157
bt_mesh_large_comp_data_cli.ack_ctx (*C var*), 2158
bt_mesh_large_comp_data_cli.cb (*C var*), 2158
bt_mesh_large_comp_data_cli.model (*C var*), 2158
bt_mesh_large_comp_data_get (*C function*), 2156
bt_mesh_large_comp_data_rsp (*C struct*), 2156
bt_mesh_large_comp_data_rsp.data (*C var*), 2157
bt_mesh_large_comp_data_rsp.offset (*C var*), 2157
bt_mesh_large_comp_data_rsp.page (*C var*), 2156
bt_mesh_large_comp_data_rsp.total_size (*C var*), 2157
BT_MESH_LEN_EXACT (*C macro*), 2089
BT_MESH_LEN_MIN (*C macro*), 2090
bt_mesh_lpn_cb (*C struct*), 2078
BT_MESH_LPN_CB_DEFINE (*C macro*), 2075
bt_mesh_lpn_cb.established (*C var*), 2078
bt_mesh_lpn_cb.polled (*C var*), 2078
bt_mesh_lpn_cb.terminated (*C var*), 2078
bt_mesh_lpn_poll (*C function*), 2077
bt_mesh_lpn_set (*C function*), 2077
BT_MESH_MIC_LONG (*C macro*), 2231
BT_MESH_MIC_SHORT (*C macro*), 2231
bt_mesh_mod_id_vnd (*C struct*), 2102
bt_mesh_mod_id_vnd.company (*C var*), 2102
bt_mesh_mod_id_vnd.id (*C var*), 2102
BT_MESH_MODEL (*C macro*), 2092
bt_mesh_model (*C struct*), 2102
BT_MESH_MODEL_BLOB_CLI (*C macro*), 2186
BT_MESH_MODEL_BLOB_SRV (*C macro*), 2181

BT_MESH_MODEL_BUF_DEFINE (*C macro*), 2231
BT_MESH_MODEL_BUF_LEN (*C macro*), 2231
BT_MESH_MODEL_BUF_LEN_LONG_MIC (*C macro*), 2231
BT_MESH_MODEL_CB (*C macro*), 2091
bt_mesh_model_cb (*C struct*), 2100
bt_mesh_model_cb.init (*C var*), 2101
bt_mesh_model_cb.pending_store (*C var*), 2102
bt_mesh_model_cb.reset (*C var*), 2101
bt_mesh_model_cb.settings_set (*C var*), 2101
bt_mesh_model_cb.start (*C var*), 2101
BT_MESH_MODEL_CFG_CLI (*C macro*), 2105
BT_MESH_MODEL_CFG_SRV (*C macro*), 2142
BT_MESH_MODEL_CNT_CB (*C macro*), 2090
BT_MESH_MODEL_CNT_VND_CB (*C macro*), 2090
bt_mesh_model_correspond (*C function*), 2096
bt_mesh_model_data_store (*C function*), 2096
bt_mesh_model_data_store_schedule (*C function*), 2096
BT_MESH_MODEL_DFD_SRV (*C macro*), 2215
BT_MESH_MODEL_DFU_CLI (*C macro*), 2208
BT_MESH_MODEL_DFU_SRV (*C macro*), 2204
bt_mesh_model_elem (*C function*), 2095
bt_mesh_model_extend (*C function*), 2096
bt_mesh_model_find (*C function*), 2095
bt_mesh_model_find_vnd (*C function*), 2095
BT_MESH_MODEL_HEALTH_CLI (*C macro*), 2142
BT_MESH_MODEL_HEALTH_SRV (*C macro*), 2150
BT_MESH_MODEL_ID_BLOB_CLI (*C macro*), 2088
BT_MESH_MODEL_ID_BLOB_SRV (*C macro*), 2088
BT_MESH_MODEL_ID_CFG_CLI (*C macro*), 2084
BT_MESH_MODEL_ID_CFG_SRV (*C macro*), 2084
BT_MESH_MODEL_ID_DFD_CLI (*C macro*), 2088
BT_MESH_MODEL_ID_DFD_SRV (*C macro*), 2088
BT_MESH_MODEL_ID_DFU_CLI (*C macro*), 2088
BT_MESH_MODEL_ID_DFU_SRV (*C macro*), 2088
BT_MESH_MODEL_ID_GEN_ADMIN_PROP_SRV (*C macro*), 2086
BT_MESH_MODEL_ID_GEN_BATTERY_CLI (*C macro*), 2086
BT_MESH_MODEL_ID_GEN_BATTERY_SRV (*C macro*), 2086
BT_MESH_MODEL_ID_GEN_CLIENT_PROP_SRV (*C macro*), 2086
BT_MESH_MODEL_ID_GEN_DEF_TRANS_TIME_CLI (*C macro*), 2085
BT_MESH_MODEL_ID_GEN_DEF_TRANS_TIME_SRV (*C macro*), 2085
BT_MESH_MODEL_ID_GEN_LEVEL_CLI (*C macro*), 2085
BT_MESH_MODEL_ID_GEN_LEVEL_SRV (*C macro*), 2085
BT_MESH_MODEL_ID_GEN_LOCATION_CLI (*C macro*), 2086
BT_MESH_MODEL_ID_GEN_LOCATION_SETUPSRV (*C macro*), 2086
BT_MESH_MODEL_ID_GEN_LOCATION_SRV (*C macro*), 2086
BT_MESH_MODEL_ID_GEN_MANUFACTURER_PROP_SRV (*C macro*), 2086
BT_MESH_MODEL_ID_GEN_ONOFF_CLI (*C macro*), 2085
BT_MESH_MODEL_ID_GEN_ONOFF_SRV (*C macro*), 2085
BT_MESH_MODEL_ID_GEN_POWER_LEVEL_CLI (*C macro*), 2085
BT_MESH_MODEL_ID_GEN_POWER_LEVEL_SETUP_SRV (*C macro*), 2085
BT_MESH_MODEL_ID_GEN_POWER_LEVEL_SRV (*C macro*), 2085
BT_MESH_MODEL_ID_GEN_POWER_ONOFF_CLI (*C macro*), 2085
BT_MESH_MODEL_ID_GEN_POWER_ONOFF_SETUP_SRV (*C macro*), 2085
BT_MESH_MODEL_ID_GEN_POWER_ONOFF_SRV (*C macro*), 2085
BT_MESH_MODEL_ID_GEN_PROP_CLI (*C macro*), 2086
BT_MESH_MODEL_ID_GEN_USER_PROP_SRV (*C macro*), 2086
BT_MESH_MODEL_ID_HEALTH_CLI (*C macro*), 2084
BT_MESH_MODEL_ID_HEALTH_SRV (*C macro*), 2084

BT_MESH_MODEL_ID_LARGE_COMP_DATA_CLI (*C macro*), 2084
BT_MESH_MODEL_ID_LARGE_COMP_DATA_SRV (*C macro*), 2084
BT_MESH_MODEL_ID_LIGHT_CTL_CLI (*C macro*), 2087
BT_MESH_MODEL_ID_LIGHT_CTL_SETUP_SRV (*C macro*), 2087
BT_MESH_MODEL_ID_LIGHT_CTL_SRV (*C macro*), 2087
BT_MESH_MODEL_ID_LIGHT_CTL_TEMP_SRV (*C macro*), 2087
BT_MESH_MODEL_ID_LIGHT_HSL_CLI (*C macro*), 2087
BT_MESH_MODEL_ID_LIGHT_HSL_HUE_SRV (*C macro*), 2088
BT_MESH_MODEL_ID_LIGHT_HSL_SAT_SRV (*C macro*), 2088
BT_MESH_MODEL_ID_LIGHT_HSL_SETUP_SRV (*C macro*), 2087
BT_MESH_MODEL_ID_LIGHT_HSL_SRV (*C macro*), 2087
BT_MESH_MODEL_ID_LIGHT_LC_CLI (*C macro*), 2088
BT_MESH_MODEL_ID_LIGHT_LC_SETUPSRV (*C macro*), 2088
BT_MESH_MODEL_ID_LIGHT_LC_SRV (*C macro*), 2088
BT_MESH_MODEL_ID_LIGHT_LIGHTNESS_CLI (*C macro*), 2087
BT_MESH_MODEL_ID_LIGHT_LIGHTNESS_SETUP_SRV (*C macro*), 2087
BT_MESH_MODEL_ID_LIGHT_LIGHTNESS_SRV (*C macro*), 2087
BT_MESH_MODEL_ID_LIGHT_XYL_CLI (*C macro*), 2088
BT_MESH_MODEL_ID_LIGHT_XYL_SETUP_SRV (*C macro*), 2088
BT_MESH_MODEL_ID_LIGHT_XYL_SRV (*C macro*), 2088
BT_MESH_MODEL_ID_ON_DEMAND_PROXY_CLI (*C macro*), 2085
BT_MESH_MODEL_ID_ON_DEMAND_PROXY_SRV (*C macro*), 2085
BT_MESH_MODEL_ID_OP_AGG_CLI (*C macro*), 2084
BT_MESH_MODEL_ID_OP_AGG_SRV (*C macro*), 2084
BT_MESH_MODEL_ID_PRIV_BEACON_CLI (*C macro*), 2084
BT_MESH_MODEL_ID_PRIV_BEACON_SRV (*C macro*), 2084
BT_MESH_MODEL_ID_REMOTE_PROV_CLI (*C macro*), 2084
BT_MESH_MODEL_ID_REMOTE_PROV_SRV (*C macro*), 2084
BT_MESH_MODEL_ID_SAR_CFG_CLI (*C macro*), 2084
BT_MESH_MODEL_ID_SAR_CFG_SRV (*C macro*), 2084
BT_MESH_MODEL_ID_SCENE_CLI (*C macro*), 2087
BT_MESH_MODEL_ID_SCENE_SETUP_SRV (*C macro*), 2087
BT_MESH_MODEL_ID_SCENE_SRV (*C macro*), 2087
BT_MESH_MODEL_ID_SCHEDULER_CLI (*C macro*), 2087
BT_MESH_MODEL_ID_SCHEDULER_SETUP_SRV (*C macro*), 2087
BT_MESH_MODEL_ID_SCHEDULER_SRV (*C macro*), 2087
BT_MESH_MODEL_ID_SENSOR_CLI (*C macro*), 2086
BT_MESH_MODEL_ID_SENSOR_SETUP_SRV (*C macro*), 2086
BT_MESH_MODEL_ID_SENSOR_SRV (*C macro*), 2086
BT_MESH_MODEL_ID_SOL_PDU_RPL_CLI (*C macro*), 2085
BT_MESH_MODEL_ID_SOL_PDU_RPL_SRV (*C macro*), 2084
BT_MESH_MODEL_ID_TIME_CLI (*C macro*), 2086
BT_MESH_MODEL_ID_TIME_SETUP_SRV (*C macro*), 2086
BT_MESH_MODEL_ID_TIME_SRV (*C macro*), 2086
bt_mesh_model_in_primary (*C function*), 2095
bt_mesh_model_is_extended (*C function*), 2097
BT_MESH_MODEL_LARGE_COMP_DATA_CLI (*C macro*), 2155
BT_MESH_MODEL_LARGE_COMP_DATA_SRV (*C macro*), 2158
BT_MESH_MODEL_METADATA_CB (*C macro*), 2091
bt_mesh_model_msg_init (*C function*), 2232
BT_MESH_MODEL_NO_OPS (*C macro*), 2090
BT_MESH_MODEL_NONE (*C macro*), 2090
BT_MESH_MODEL_OD_PRIV_PROXY_CLI (*C macro*), 2159
BT_MESH_MODEL_OD_PRIV_PROXY_SRV (*C macro*), 2161
bt_mesh_model_op (*C struct*), 2098
BT_MESH_MODEL_OP_1 (*C macro*), 2089
BT_MESH_MODEL_OP_2 (*C macro*), 2089
BT_MESH_MODEL_OP_3 (*C macro*), 2089

BT_MESH_MODEL_OP_AGG_CLI (*C macro*), 2161
BT_MESH_MODEL_OP_AGG_SRV (*C macro*), 2163
BT_MESH_MODEL_OP_END (*C macro*), 2090
BT_MESH_MODEL_OP_LEN (*C macro*), 2231
bt_mesh_model_op.func (*C var*), 2098
bt_mesh_model_op.len (*C var*), 2098
bt_mesh_model_op.opcode (*C var*), 2098
BT_MESH_MODEL_PRIV_BEACON_CLI (*C macro*), 2163
BT_MESH_MODEL_PRIV_BEACON_SRV (*C macro*), 2167
bt_mesh_model_pub (*C struct*), 2099
BT_MESH_MODEL_PUB_DEFINE (*C macro*), 2093
bt_mesh_model_pub_is_retransmission (*C function*), 2095
bt_mesh_model_pub.addr (*C var*), 2099
bt_mesh_model_pub.count (*C var*), 2100
bt_mesh_model_pub.cred (*C var*), 2099
bt_mesh_model_pub.delayable (*C var*), 2100
bt_mesh_model_pub.fast_period (*C var*), 2099
bt_mesh_model_pub.key (*C var*), 2099
bt_mesh_model_publish (*C function*), 2094
bt_mesh_model_pub.mod (*C var*), 2099
bt_mesh_model_pub.msg (*C var*), 2100
bt_mesh_model_pub.period (*C var*), 2100
bt_mesh_model_pub.period_div (*C var*), 2100
bt_mesh_model_pub.period_start (*C var*), 2100
bt_mesh_model_pub.retr_update (*C var*), 2099
bt_mesh_model_pub.retransmit (*C var*), 2099
bt_mesh_model_pub.send_rel (*C var*), 2099
bt_mesh_model_pub.timer (*C var*), 2100
bt_mesh_model_pub.ttl (*C var*), 2099
bt_mesh_model_pub.update (*C var*), 2100
bt_mesh_model_pub.uuid (*C var*), 2099
BT_MESH_MODEL_RPR_CLI (*C macro*), 2169
BT_MESH_MODEL_RPR_SRV (*C macro*), 2173
BT_MESH_MODEL_SAR_CFG_CLI (*C macro*), 2174
BT_MESH_MODEL_SAR_CFG_SRV (*C macro*), 2176
bt_mesh_model_send (*C function*), 2094
BT_MESH_MODEL_SOL_PDU_RPL_CLI (*C macro*), 2177
BT_MESH_MODEL_SOL_PDU_RPL_SRV (*C macro*), 2179
BT_MESH_MODEL_VND (*C macro*), 2092
BT_MESH_MODEL_VND_CB (*C macro*), 2091
BT_MESH_MODEL_VND_METADATA_CB (*C macro*), 2091
bt_mesh_model.bt_mesh_model_rt_ctx (*C struct*), 2103
bt_mesh_model.bt_mesh_model_rt_ctx.user_data (*C var*), 2103
bt_mesh_model.cb (*C var*), 2103
bt_mesh_model.groups (*C var*), 2102
bt_mesh_model.id (*C var*), 2102
bt_mesh_model.keys (*C var*), 2102
bt_mesh_model.op (*C var*), 2102
bt_mesh_model.pub (*C var*), 2102
bt_mesh_models_metadata_change_prepare (*C function*), 2097
BT_MESH_MODELS_METADATA_END (*C macro*), 2094
BT_MESH_MODELS_METADATA_ENTRY (*C macro*), 2094
bt_mesh_models_metadata_entry (*C struct*), 2100
bt_mesh_models_metadata_get (*C function*), 2156
BT_MESH_MODELS_METADATA_NONE (*C macro*), 2094
bt_mesh_model.uuids (*C var*), 2102
bt_mesh_model.vnd (*C var*), 2102
bt_mesh_msg_ack_ctx (*C struct*), 2235

bt_mesh_msg_ack_ctx_busy (*C function*), 2233
bt_mesh_msg_ack_ctx_clear (*C function*), 2233
bt_mesh_msg_ack_ctx_init (*C function*), 2232
bt_mesh_msg_ack_ctx_match (*C function*), 2234
bt_mesh_msg_ack_ctx_prepare (*C function*), 2233
bt_mesh_msg_ack_ctx_reset (*C function*), 2232
bt_mesh_msg_ack_ctx_rx (*C function*), 2233
bt_mesh_msg_ack_ctx_wait (*C function*), 2233
bt_mesh_msg_ack_ctx.dst (*C var*), 2235
bt_mesh_msg_ack_ctx.op (*C var*), 2235
bt_mesh_msg_ack_ctx.sem (*C var*), 2235
bt_mesh_msg_ack_ctx.user_data (*C var*), 2235
bt_mesh_msg_ctx (*C struct*), 2234
BT_MESH_MSG_CTX_INIT (*C macro*), 2231
BT_MESH_MSG_CTX_INIT_APP (*C macro*), 2232
BT_MESH_MSG_CTX_INIT_DEV (*C macro*), 2232
BT_MESH_MSG_CTX_INIT_PUB (*C macro*), 2232
bt_mesh_msg_ctx.addr (*C var*), 2234
bt_mesh_msg_ctx.app_idx (*C var*), 2234
bt_mesh_msg_ctx.net_idx (*C var*), 2234
bt_mesh_msg_ctx.recv_dst (*C var*), 2234
bt_mesh_msg_ctx.recv_rssi (*C var*), 2234
bt_mesh_msg_ctx.recv_ttl (*C var*), 2234
bt_mesh_msg_ctx.rnd_delay (*C var*), 2234
bt_mesh_msg_ctx.send_rel (*C var*), 2234
bt_mesh_msg_ctx.send_ttl (*C var*), 2235
bt_mesh_msg_ctx.uuid (*C var*), 2234
BT_MESH_NET_PRIMARY (*C macro*), 2075
bt_mesh_net_transmit_get (*C function*), 2262
bt_mesh_net_transmit_set (*C function*), 2261
BT_MESH_NODE_IDENTITY_NOT_SUPPORTED (*C macro*), 2259
BT_MESH_NODE_IDENTITY_RUNNING (*C macro*), 2259
BT_MESH_NODE_IDENTITY_STOPPED (*C macro*), 2259
bt_mesh_od_priv_proxy_cli (*C struct*), 2160
bt_mesh_od_priv_proxy_cli_get (*C function*), 2159
bt_mesh_od_priv_proxy_cli_set (*C function*), 2159
bt_mesh_od_priv_proxy_cli_timeout_set (*C function*), 2160
bt_mesh_od_priv_proxy_cli.model (*C var*), 2160
bt_mesh_od_priv_proxy_cli.od_status (*C var*), 2160
bt_mesh_od_priv_proxy_get (*C function*), 2261
bt_mesh_od_priv_proxy_set (*C function*), 2261
bt_mesh_op_agg_cli_seq_abort (*C function*), 2162
bt_mesh_op_agg_cli_seq_is_started (*C function*), 2162
bt_mesh_op_agg_cli_seq_send (*C function*), 2162
bt_mesh_op_agg_cli_seq_start (*C function*), 2161
bt_mesh_op_agg_cli_seq_tailroom (*C function*), 2162
bt_mesh_op_agg_cli_timeout_get (*C function*), 2162
bt_mesh_op_agg_cli_timeout_set (*C function*), 2162
bt_mesh_output_action_t (*C enum*), 2242
bt_mesh_output_action_t.BT_MESH_BEEP (*C enumerator*), 2242
bt_mesh_output_action_t.BT_MESH_BLINK (*C enumerator*), 2242
bt_mesh_output_action_t.BT_MESH_DISPLAY_NUMBER (*C enumerator*), 2242
bt_mesh_output_action_t.BT_MESH_DISPLAY_STRING (*C enumerator*), 2242
bt_mesh_output_action_t.BT_MESH_NO_OUTPUT (*C enumerator*), 2242
bt_mesh_output_action_t.BT_MESH_VIBRATE (*C enumerator*), 2242
bt_mesh_priv_beacon (*C struct*), 2165
bt_mesh_priv_beacon_cli (*C struct*), 2166
bt_mesh_priv_beacon_cli_cb (*C struct*), 2165

bt_mesh_priv_beacon_cli_cb.priv_beacon_status (*C var*), 2166
bt_mesh_priv_beacon_cli_cb.priv_gatt_proxy_status (*C var*), 2166
bt_mesh_priv_beacon_cli_cb.priv_node_id_status (*C var*), 2166
bt_mesh_priv_beacon_cli_gatt_proxy_get (*C function*), 2164
bt_mesh_priv_beacon_cli_gatt_proxy_set (*C function*), 2164
bt_mesh_priv_beacon_cli_get (*C function*), 2164
bt_mesh_priv_beacon_cli_node_id_get (*C function*), 2165
bt_mesh_priv_beacon_cli_node_id_set (*C function*), 2164
bt_mesh_priv_beacon_cli_set (*C function*), 2163
bt_mesh_priv_beacon_cli.cb (*C var*), 2166
BT_MESH_PRIV_BEACON_DISABLED (*C macro*), 2259
BT_MESH_PRIV_BEACON_ENABLED (*C macro*), 2259
bt_mesh_priv_beacon_get (*C function*), 2260
bt_mesh_priv_beacon_set (*C function*), 2260
bt_mesh_priv_beacon_update_interval_get (*C function*), 2260
bt_mesh_priv_beacon_update_interval_set (*C function*), 2260
bt_mesh_priv_beacon.enabled (*C var*), 2165
bt_mesh_priv_beacon.rand_interval (*C var*), 2165
BT_MESH_PRIV_GATT_PROXY_DISABLED (*C macro*), 2259
BT_MESH_PRIV_GATT_PROXY_ENABLED (*C macro*), 2259
bt_mesh_priv_gatt_proxy_get (*C function*), 2263
BT_MESH_PRIV_GATT_PROXY_NOT_SUPPORTED (*C macro*), 2259
bt_mesh_priv_gatt_proxy_set (*C function*), 2263
bt_mesh_priv_node_id (*C struct*), 2165
bt_mesh_priv_node_id.net_idx (*C var*), 2165
bt_mesh_priv_node_id.state (*C var*), 2165
bt_mesh_priv_node_id.status (*C var*), 2165
bt_mesh_prov (*C struct*), 2249
bt_mesh_prov_bearer_t (*C enum*), 2243
bt_mesh_prov_bearer_t.BT_MESH_PROV_ADV (*C enumerator*), 2243
bt_mesh_prov_bearer_t.BT_MESH_PROV_GATT (*C enumerator*), 2243
bt_mesh_prov_bearer_t.BT_MESH_PROV_REMOTE (*C enumerator*), 2243
bt_mesh_prov_disable (*C function*), 2246
bt_mesh_prov_enable (*C function*), 2246
bt_mesh_prov_oob_info_t (*C enum*), 2243
bt_mesh_prov_oob_info_t.BT_MESH_PROV_OOB_2D_CODE (*C enumerator*), 2243
bt_mesh_prov_oob_info_t.BT_MESH_PROV_OOB_BAR_CODE (*C enumerator*), 2243
bt_mesh_prov_oob_info_t.BT_MESH_PROV_OOB_CERTIFICATE (*C enumerator*), 2244
bt_mesh_prov_oob_info_t.BT_MESH_PROV_OOB_IN_BOX (*C enumerator*), 2244
bt_mesh_prov_oob_info_t.BT_MESH_PROV_OOB_IN_MANUAL (*C enumerator*), 2244
bt_mesh_prov_oob_info_t.BT_MESH_PROV_OOB_NFC (*C enumerator*), 2243
bt_mesh_prov_oob_info_t.BT_MESH_PROV_OOB_NUMBER (*C enumerator*), 2244
bt_mesh_prov_oob_info_t.BT_MESH_PROV_OOB_ON_BOX (*C enumerator*), 2244
bt_mesh_prov_oob_info_t.BT_MESH_PROV_OOB_ON_DEV (*C enumerator*), 2244
bt_mesh_prov_oob_info_t.BT_MESH_PROV_OOB_ON_PAPER (*C enumerator*), 2244
bt_mesh_prov_oob_info_t.BT_MESH_PROV_OOB_OTHER (*C enumerator*), 2243
bt_mesh_prov_oob_info_t.BT_MESH_PROV_OOB_RECORDS (*C enumerator*), 2244
bt_mesh_prov_oob_info_t.BT_MESH_PROV_OOB_STRING (*C enumerator*), 2244
bt_mesh_prov_oob_info_t.BT_MESH_PROV_OOB_URI (*C enumerator*), 2243
bt_mesh_prov_remote_pub_key_set (*C function*), 2245
bt_mesh_prov.capabilities (*C var*), 2250
bt_mesh_prov.complete (*C var*), 2252
bt_mesh_prov.input (*C var*), 2250
bt_mesh_prov.input_actions (*C var*), 2250
bt_mesh_prov.input_complete (*C var*), 2251
bt_mesh_prov.input_size (*C var*), 2250
bt_mesh_provision (*C function*), 2246
bt_mesh_provision_adv (*C function*), 2247

bt_mesh_provision_gatt (*C function*), 2247
bt_mesh_provision_remote (*C function*), 2247
bt_mesh_prov.link_close (*C var*), 2251
bt_mesh_prov.link_open (*C var*), 2251
bt_mesh_prov.node_added (*C var*), 2252
bt_mesh_prov.oob_info (*C var*), 2249
bt_mesh_prov.output_actions (*C var*), 2250
bt_mesh_prov.output_number (*C var*), 2250
bt_mesh_prov.output_size (*C var*), 2250
bt_mesh_prov.output_string (*C var*), 2250
bt_mesh_prov.private_key_be (*C var*), 2249
bt_mesh_prov.public_key_be (*C var*), 2249
bt_mesh_prov.reprovisioned (*C var*), 2252
bt_mesh_prov.reset (*C var*), 2252
bt_mesh_prov.static_val (*C var*), 2249
bt_mesh_prov.static_val_len (*C var*), 2250
bt_mesh_prov.unprovisioned_beacon (*C var*), 2251
bt_mesh_prov.unprovisioned_beacon_gatt (*C var*), 2251
bt_mesh_prov.uri (*C var*), 2249
bt_mesh_prov.uuid (*C var*), 2249
bt_mesh_proxy_cb (*C struct*), 2254
BT_MESH_PROXY_CB_DEFINE (*C macro*), 2253
bt_mesh_proxy_cb.identity_disabled (*C var*), 2255
bt_mesh_proxy_cb.identity_enabled (*C var*), 2255
bt_mesh_proxy_connect (*C function*), 2254
bt_mesh_proxy_disconnect (*C function*), 2254
bt_mesh_proxy_identity_enable (*C function*), 2253
bt_mesh_proxy_private_identity_enable (*C function*), 2254
bt_mesh_proxy_solicit (*C function*), 2254
BT_MESH_PUB_MSG_NUM (*C macro*), 2093
BT_MESH_PUB_MSG_TOTAL (*C macro*), 2093
BT_MESH_PUB_PERIOD_10MIN (*C macro*), 2106
BT_MESH_PUB_PERIOD_10SEC (*C macro*), 2106
BT_MESH_PUB_PERIOD_100MS (*C macro*), 2105
BT_MESH_PUB_PERIOD_SEC (*C macro*), 2105
BT_MESH_PUB_TRANSMIT (*C macro*), 2093
BT_MESH_PUB_TRANSMIT_COUNT (*C macro*), 2093
BT_MESH_PUB_TRANSMIT_INT (*C macro*), 2093
BT_MESH_RELAY_DISABLED (*C macro*), 2258
BT_MESH_RELAY_ENABLED (*C macro*), 2258
bt_mesh_relay_get (*C function*), 2262
BT_MESH_RELAY_NOT_SUPPORTED (*C macro*), 2259
bt_mesh_relay_retransmit_get (*C function*), 2263
bt_mesh_relay_set (*C function*), 2262
bt_mesh_reprovision_remote (*C function*), 2247
bt_mesh_reset (*C function*), 2076
bt_mesh_resume (*C function*), 2076
bt_mesh_rpl_pending_store (*C function*), 2077
bt_mesh_rpr_caps (*C struct*), 2172
bt_mesh_rpr_caps.active_scan (*C var*), 2172
bt_mesh_rpr_caps.max_devs (*C var*), 2172
bt_mesh_rpr_cli (*C struct*), 2172
bt_mesh_rpr_cli_timeout_get (*C function*), 2172
bt_mesh_rpr_cli_timeout_set (*C function*), 2172
bt_mesh_rpr_cli.scan_report (*C var*), 2173
bt_mesh_rpr_link_close (*C function*), 2171
bt_mesh_rpr_link_get (*C function*), 2171
bt_mesh_rpr_scan_caps_get (*C function*), 2169

bt_mesh_rpr_scan_get (*C function*), 2170
BT_MESH_RPR_SCAN_MAX_DEVS_ANY (*C macro*), 2169
bt_mesh_rpr_scan_start (*C function*), 2170
bt_mesh_rpr_scan_start_ext (*C function*), 2170
bt_mesh_rpr_scan_status (*C struct*), 2172
bt_mesh_rpr_scan_status.max_devs (*C var*), 2172
bt_mesh_rpr_scan_status.scan (*C var*), 2172
bt_mesh_rpr_scan_status.status (*C var*), 2172
bt_mesh_rpr_scan_status.timeout (*C var*), 2172
bt_mesh_rpr_scan_stop (*C function*), 2171
BT_MESH_RX_SDU_MAX (*C macro*), 2089
BT_MESH_RX_SEG_MAX (*C macro*), 2089
bt_mesh_sar_cfg_cli (*C struct*), 2175
bt_mesh_sar_cfg_cli_receiver_get (*C function*), 2175
bt_mesh_sar_cfg_cli_receiver_set (*C function*), 2175
bt_mesh_sar_cfg_cli_timeout_get (*C function*), 2175
bt_mesh_sar_cfg_cli_timeout_set (*C function*), 2175
bt_mesh_sar_cfg_cli_transmitter_get (*C function*), 2174
bt_mesh_sar_cfg_cli_transmitter_set (*C function*), 2175
bt_mesh_sar_cfg_cli.model (*C var*), 2176
bt_mesh_send_cb (*C struct*), 2103
bt_mesh_send_cb.end (*C var*), 2103
bt_mesh_send_cb.start (*C var*), 2103
bt_mesh_sol_pdu_rpl_clear (*C function*), 2177
bt_mesh_sol_pdu_rpl_clear_unack (*C function*), 2177
bt_mesh_sol_pdu_rpl_cli (*C struct*), 2178
bt_mesh_sol_pdu_rpl_cli_timeout_set (*C function*), 2177
bt_mesh_sol_pdu_rpl_cli.model (*C var*), 2178
bt_mesh_sol_pdu_rpl_cli.srpl_status (*C var*), 2178
bt_mesh_stat_get (*C function*), 2264
bt_mesh_stat_reset (*C function*), 2264
bt_mesh_statistic (*C struct*), 2264
bt_mesh_statistic.rx_adv (*C var*), 2265
bt_mesh_statistic.rx_loopback (*C var*), 2265
bt_mesh_statistic.rx_proxy (*C var*), 2265
bt_mesh_statistic.rx_uknown (*C var*), 2265
bt_mesh_statistic.tx_adv_relay_planned (*C var*), 2265
bt_mesh_statistic.tx_adv_relay_succeeded (*C var*), 2265
bt_mesh_statistic.tx_friend_planned (*C var*), 2265
bt_mesh_statistic.tx_friend_succeeded (*C var*), 2265
bt_mesh_statistic.tx_local_planned (*C var*), 2265
bt_mesh_statistic.tx_local_succeeded (*C var*), 2265
bt_mesh_suspend (*C function*), 2076
BT_MESH_TRANSMIT (*C macro*), 2092
BT_MESH_TRANSMIT_COUNT (*C macro*), 2092
BT_MESH_TRANSMIT_INT (*C macro*), 2093
BT_MESH_TTL_DEFAULT (*C macro*), 2094
BT_MESH_TTL_MAX (*C macro*), 2094
BT_MESH_TX_SDU_MAX (*C macro*), 2089
BT_MESH_TX_SEG_MAX (*C macro*), 2089
bt_mesh_va_uuid_get (*C function*), 2077
BT_MICP_ERR_MUTE_DISABLED (*C macro*), 1890
bt_micp_included (*C struct*), 1893
bt_micp_included.aics (*C var*), 1893
bt_micp_included.aics_cnt (*C var*), 1893
bt_micp_mic_ctlr_cb (*C struct*), 1893
bt_micp_mic_ctlr_cb_register (*C function*), 1892
bt_micp_mic_ctlr_cb.discover (*C var*), 1894

bt_micp_mic_ctlr_cb.mute (*C var*), 1894
bt_micp_mic_ctlr_cb.mute_written (*C var*), 1894
bt_micp_mic_ctlr_cb.unmute_written (*C var*), 1894
bt_micp_mic_ctlr_conn_get (*C function*), 1891
bt_micp_mic_ctlr_discover (*C function*), 1892
bt_micp_mic_ctlr_get_by_conn (*C function*), 1891
bt_micp_mic_ctlr_included_get (*C function*), 1891
bt_micp_mic_ctlr_mute (*C function*), 1892
bt_micp_mic_ctlr_mute_get (*C function*), 1892
bt_micp_mic_ctlr_unmute (*C function*), 1892
BT_MICP_MIC_DEV_AICS_CNT (*C macro*), 1890
bt_micp_mic_dev_cb (*C struct*), 1893
bt_micp_mic_dev_cb.mute (*C var*), 1893
bt_micp_mic_dev_included_get (*C function*), 1890
bt_micp_mic_dev_mute (*C function*), 1891
bt_micp_mic_dev_mute_disable (*C function*), 1891
bt_micp_mic_dev_mute_get (*C function*), 1891
bt_micp_mic_dev_register (*C function*), 1890
bt_micp_mic_dev_register_param (*C struct*), 1892
bt_micp_mic_dev_register_param.aics_param (*C var*), 1893
bt_micp_mic_dev_register_param.cb (*C var*), 1893
bt_micp_mic_dev_unmute (*C function*), 1890
BT_MICP_MUTE_DISABLED (*C macro*), 1890
BT_MICP_MUTE_MUTED (*C macro*), 1890
BT_MICP_MUTE_UNMUTED (*C macro*), 1890
bt_ots_cb (*C struct*), 1952
bt_ots_cb.obj_cal_checksum (*C var*), 1955
bt_ots_cb.obj_created (*C var*), 1952
bt_ots_cb.obj_deleted (*C var*), 1953
bt_ots_cb.obj_name_written (*C var*), 1955
bt_ots_cb.obj_read (*C var*), 1954
bt_ots_cb.obj_selected (*C var*), 1953
bt_ots_cb.obj_write (*C var*), 1954
bt_ots_client (*C struct*), 1955
bt_ots_client_cb (*C struct*), 1955
bt_ots_client_cb.obj_checksum_calculated (*C var*), 1957
bt_ots_client_cb.obj_data_read (*C var*), 1956
bt_ots_client_cb.obj_data_written (*C var*), 1956
bt_ots_client_cb.obj_metadata_read (*C var*), 1956
bt_ots_client_cb.obj_selected (*C var*), 1956
bt_ots_client_decode_dirlisting (*C function*), 1950
bt_ots_client_dirlisting_cb (*C type*), 1944
bt_ots_client_get_object_checksum (*C function*), 1950
bt_ots_client_indicate_handler (*C function*), 1948
bt_ots_client_read_feature (*C function*), 1948
bt_ots_client_read_object_data (*C function*), 1949
bt_ots_client_read_object_metadata (*C function*), 1949
bt_ots_client_register (*C function*), 1947
bt_ots_client_select_first (*C function*), 1948
bt_ots_client_select_id (*C function*), 1948
bt_ots_client_select_last (*C function*), 1948
bt_ots_client_select_next (*C function*), 1949
bt_ots_client_select_prev (*C function*), 1949
bt_ots_client_unregister (*C function*), 1947
bt_ots_client_write_object_data (*C function*), 1949
BT_OTS_CONTINUE (*C macro*), 1943
bt_ots_date_time (*C struct*), 1951
BT_OTS_DATE_TIME_FIELD_SIZE (*C macro*), 1943

bt_ots_feat (*C struct*), 1951
bt_ots_free_instance_get (*C function*), 1947
bt_ots_init (*C function*), 1947
bt_ots_init_param (*C struct*), 1955
bt_ots_metadata_display (*C function*), 1951
BT_OTS_OACP_GET_FEAT_ABORT (*C macro*), 1942
BT_OTS_OACP_GET_FEAT_APPEND (*C macro*), 1942
BT_OTS_OACP_GET_FEAT_CHECKSUM (*C macro*), 1942
BT_OTS_OACP_GET_FEAT_CREATE (*C macro*), 1941
BT_OTS_OACP_GET_FEAT_DELETE (*C macro*), 1942
BT_OTS_OACP_GET_FEAT_EXECUTE (*C macro*), 1942
BT_OTS_OACP_GET_FEAT_PATCH (*C macro*), 1942
BT_OTS_OACP_GET_FEAT_READ (*C macro*), 1942
BT_OTS_OACP_GET_FEAT_TRUNCATE (*C macro*), 1942
BT_OTS_OACP_GET_FEAT_WRITE (*C macro*), 1942
BT_OTS_OACP_SET_FEAT_ABORT (*C macro*), 1941
BT_OTS_OACP_SET_FEAT_APPEND (*C macro*), 1941
BT_OTS_OACP_SET_FEAT_CHECKSUM (*C macro*), 1941
BT_OTS_OACP_SET_FEAT_CREATE (*C macro*), 1940
BT_OTS_OACP_SET_FEAT_DELETE (*C macro*), 1941
BT_OTS_OACP_SET_FEAT_EXECUTE (*C macro*), 1941
BT_OTS_OACP_SET_FEAT_PATCH (*C macro*), 1941
BT_OTS_OACP_SET_FEAT_READ (*C macro*), 1941
BT_OTS_OACP_SET_FEAT_TRUNCATE (*C macro*), 1941
BT_OTS_OACP_SET_FEAT_WRITE (*C macro*), 1941
bt_ots_oacp_write_op_mode (*C enum*), 1945
bt_ots_oacp_write_op_mode.BT_OTS_OACP_WRITE_OP_MODE_NONE (*C enumerator*), 1945
bt_ots_oacp_write_op_mode.BT_OTS_OACP_WRITE_OP_MODE_TRUNCATE (*C enumerator*), 1945
bt_ots_obj_add (*C function*), 1946
bt_ots_obj_add_param (*C struct*), 1952
bt_ots_obj_add_param.size (*C var*), 1952
bt_ots_obj_add_param.type (*C var*), 1952
bt_ots_obj_created_desc (*C struct*), 1952
bt_ots_obj_created_desc.name (*C var*), 1952
bt_ots_obj_created_desc.props (*C var*), 1952
bt_ots_obj_created_desc.size (*C var*), 1952
bt_ots_obj_delete (*C function*), 1947
BT_OTS_OBJ_GET_PROP_APPEND (*C macro*), 1940
BT_OTS_OBJ_GET_PROP_DELETE (*C macro*), 1940
BT_OTS_OBJ_GET_PROP_EXECUTE (*C macro*), 1940
BT_OTS_OBJ_GET_PROP_MARKED (*C macro*), 1940
BT_OTS_OBJ_GET_PROP_PATCH (*C macro*), 1940
BT_OTS_OBJ_GET_PROP_READ (*C macro*), 1940
BT_OTS_OBJ_GET_PROP_TRUNCATE (*C macro*), 1940
BT_OTS_OBJ_GET_PROP_WRITE (*C macro*), 1940
BT_OTS_OBJ_ID_MASK (*C macro*), 1939
BT_OTS_OBJ_ID_MAX (*C macro*), 1939
BT_OTS_OBJ_ID_MIN (*C macro*), 1939
BT_OTS_OBJ_ID_SIZE (*C macro*), 1939
BT_OTS_OBJ_ID_STR_LEN (*C macro*), 1939
bt_ots_obj_id_to_str (*C function*), 1950
bt_ots_obj_metadata (*C struct*), 1951
bt_ots_obj_metadata.props (*C var*), 1951
bt_ots_obj_metadata.size (*C var*), 1951
bt_ots_obj_metadata.type (*C var*), 1951
BT_OTS_OBJ_SET_PROP_APPEND (*C macro*), 1939
BT_OTS_OBJ_SET_PROP_DELETE (*C macro*), 1939
BT_OTS_OBJ_SET_PROP_EXECUTE (*C macro*), 1939

BT_OTS_OBJ_SET_PROP_MARKED (*C macro*), 1940
BT_OTS_OBJ_SET_PROP_PATCH (*C macro*), 1939
BT_OTS_OBJ_SET_PROP_READ (*C macro*), 1939
BT_OTS_OBJ_SET_PROP_TRUNCATE (*C macro*), 1939
BT_OTS_OBJ_SET_PROP_WRITE (*C macro*), 1939
bt_ots_obj_size (*C struct*), 1951
bt_ots_obj_size.alloc (*C var*), 1951
bt_ots_obj_size.cur (*C var*), 1951
bt_ots_obj_type (*C struct*), 1951
BT_OTS_OLCP_GET_FEAT_CLEAR (*C macro*), 1943
BT_OTS_OLCP_GET_FEAT_GO_TO (*C macro*), 1943
BT_OTS_OLCP_GET_FEAT_NUM_REQ (*C macro*), 1943
BT_OTS_OLCP_GET_FEAT_ORDER (*C macro*), 1943
BT_OTS_OLCP_SET_FEAT_CLEAR (*C macro*), 1943
BT_OTS_OLCP_SET_FEAT_GO_TO (*C macro*), 1942
BT_OTS_OLCP_SET_FEAT_NUM_REQ (*C macro*), 1943
BT_OTS_OLCP_SET_FEAT_ORDER (*C macro*), 1943
BT_OTS_STOP (*C macro*), 1943
bt_ots_svc_decl_get (*C function*), 1947
bt_pacs_cap (*C struct*), 1834
bt_pacs_cap_foreach (*C function*), 1830
bt_pacs_cap_foreach_func_t (*C type*), 1826
bt_pacs_cap_register (*C function*), 1830
bt_pacs_cap_unregister (*C function*), 1830
bt_pacs_cap.codec_cap (*C var*), 1835
bt_pacs_conn_set_available_contexts_for_conn (*C function*), 1831
bt_pacs_get_available_contexts (*C function*), 1831
bt_pacs_get_available_contexts_for_conn (*C function*), 1831
bt_pacs_set_available_contexts (*C function*), 1831
bt_pacs_set_location (*C function*), 1830
bt_pacs_set_supported_contexts (*C function*), 1831
BT_PASSKEY_INVALID (*C macro*), 2312
bt_passkey_set (*C function*), 2326
bt_rand (*C function*), 2352
bt_ready_cb_t (*C type*), 1962
bt_recv (*C function*), 2347
bt_rfcomm_create_pdu (*C function*), 1709
bt_rfcomm_dlc (*C struct*), 1710
bt_rfcomm_dlc_connect (*C function*), 1709
bt_rfcomm_dlc_disconnect (*C function*), 1709
bt_rfcomm_dlc_ops (*C struct*), 1710
bt_rfcomm_dlc_ops.connected (*C var*), 1710
bt_rfcomm_dlc_ops.disconnected (*C var*), 1710
bt_rfcomm_dlc_ops.recv (*C var*), 1710
bt_rfcomm_dlc_ops.sent (*C var*), 1710
bt_rfcomm_dlc_send (*C function*), 1709
bt_rfcomm_role (*C enum*), 1708
bt_rfcomm_role_t (*C type*), 1708
bt_rfcomm_role.BT_RFCOMM_ROLE_ACCEPTOR (*C enumerator*), 1708
bt_rfcomm_role.BT_RFCOMM_ROLE_INITIATOR (*C enumerator*), 1709
bt_rfcomm_server (*C struct*), 1710
bt_rfcomm_server_register (*C function*), 1709
bt_rfcomm_server.accept (*C var*), 1711
bt_rfcomm_server.channel (*C var*), 1710
BT_SDP_ADVANCED_AUDIO_SVCLASS (*C macro*), 1712
BT_SDP_ALT8 (*C macro*), 1722
BT_SDP_ALT16 (*C macro*), 1722
BT_SDP_ALT32 (*C macro*), 1722

BT_SDP_ALT_UNSPEC (*C macro*), 1722
BT_SDP_APPLE_AGENT_SVCLASS (*C macro*), 1716
BT_SDP_ARRAY_8 (*C macro*), 1723
BT_SDP_ARRAY_16 (*C macro*), 1723
BT_SDP_ARRAY_32 (*C macro*), 1723
BT_SDP_ATTR_ADD_PROTO_DESC_LIST (*C macro*), 1717
BT_SDP_ATTR_AUDIO_FEEDBACK_SUPPORT (*C macro*), 1718
BT_SDP_ATTR_BROWSE_GRP_LIST (*C macro*), 1716
BT_SDP_ATTR_CLNT_EXEC_URL (*C macro*), 1717
BT_SDP_ATTR_DATA_EXCHANGE_SPEC (*C macro*), 1717
BT_SDP_ATTR_DOC_URL (*C macro*), 1716
BT_SDP_ATTR_EXTERNAL_NETWORK (*C macro*), 1717
BT_SDP_ATTR_FAX_CLASS1_SUPPORT (*C macro*), 1718
BT_SDP_ATTR_FAX_CLASS2_SUPPORT (*C macro*), 1718
BT_SDP_ATTR_FAX_CLASS20_SUPPORT (*C macro*), 1718
BT_SDP_ATTR_GOEP_L2CAP_PSM (*C macro*), 1717
BT_SDP_ATTR_GROUP_ID (*C macro*), 1717
BT_SDP_ATTR_HID_BATTERY_POWER (*C macro*), 1720
BT_SDP_ATTR_HID_BOOT_DEVICE (*C macro*), 1720
BT_SDP_ATTR_HID_COUNTRY_CODE (*C macro*), 1720
BT_SDP_ATTR_HID_DESCRIPTOR_LIST (*C macro*), 1720
BT_SDP_ATTR_HID_DEVICE_RELEASE_NUMBER (*C macro*), 1720
BT_SDP_ATTR_HID_DEVICE_SUBCLASS (*C macro*), 1720
BT_SDP_ATTR_HID_LANG_ID_BASE_LIST (*C macro*), 1720
BT_SDP_ATTR_HID_NORMALLY_CONNECTABLE (*C macro*), 1720
BT_SDP_ATTR_HID_PARSER_VERSION (*C macro*), 1720
BT_SDP_ATTR_HID_PROFILE_VERSION (*C macro*), 1720
BT_SDP_ATTR_HID_RECONNECT_INITIATE (*C macro*), 1720
BT_SDP_ATTR_HID_REMOTE_WAKEUP (*C macro*), 1720
BT_SDP_ATTR_HID_SDP_DISABLE (*C macro*), 1720
BT_SDP_ATTR_HID_SUPERVISION_TIMEOUT (*C macro*), 1720
BT_SDP_ATTR_HID_VIRTUAL_CABLE (*C macro*), 1720
BT_SDP_ATTR_HOMEPAGE_URL (*C macro*), 1718
BT_SDP_ATTR_ICON_URL (*C macro*), 1717
BT_SDP_ATTR_IP4_SUBNET (*C macro*), 1718
BT_SDP_ATTR_IP6_SUBNET (*C macro*), 1719
BT_SDP_ATTR_IP_SUBNET (*C macro*), 1717
BT_SDP_ATTR_LANG_BASE_ATTR_ID_LIST (*C macro*), 1716
BT_SDP_ATTR_MAP_SUPPORTED_FEATURES (*C macro*), 1719
BT_SDP_ATTR_MAS_INSTANCE_ID (*C macro*), 1719
BT_SDP_ATTR_MAX_NET_ACCESSRATE (*C macro*), 1718
BT_SDP_ATTR_MCAP_SUPPORTED_PROCEDURES (*C macro*), 1718
BT_SDP_ATTR_MPMD_SCENARIOS (*C macro*), 1717
BT_SDP_ATTR_MPS_DEPENDENCIES (*C macro*), 1717
BT_SDP_ATTR_MPSD_SCENARIOS (*C macro*), 1717
BT_SDP_ATTR_NET_ACCESS_TYPE (*C macro*), 1718
BT_SDP_ATTR_NETWORK (*C macro*), 1718
BT_SDP_ATTR_NETWORK_ADDRESS (*C macro*), 1718
BT_SDP_ATTR_PBAP_SUPPORTED_FEATURES (*C macro*), 1719
BT_SDP_ATTR_PRIMARY_RECORD (*C macro*), 1719
BT_SDP_ATTR_PRODUCT_ID (*C macro*), 1719
BT_SDP_ATTR_PROFILE_DESC_LIST (*C macro*), 1716
BT_SDP_ATTR_PROTO_DESC_LIST (*C macro*), 1716
BT_SDP_ATTR_PROVNAME_PRIMARY (*C macro*), 1723
BT_SDP_ATTR_RECORD_HANDLE (*C macro*), 1716
BT_SDP_ATTR_RECORD_STATE (*C macro*), 1716
BT_SDP_ATTR_REMOTE_AUDIO_VOLUME_CONTROL (*C macro*), 1718
BT_SDP_ATTR_SECURITY_DESC (*C macro*), 1718

BT_SDP_ATTR_SERVICE_AVAILABILITY (*C macro*), 1716
BT_SDP_ATTR_SERVICE_ID (*C macro*), 1716
BT_SDP_ATTR_SERVICE_VERSION (*C macro*), 1717
BT_SDP_ATTR_SPECIFICATION_ID (*C macro*), 1719
BT_SDP_ATTR_SUPPORTED_CAPABILITIES (*C macro*), 1719
BT_SDP_ATTR_SUPPORTED_DATA_STORES_LIST (*C macro*), 1717
BT_SDP_ATTR_SUPPORTED_FEATURES (*C macro*), 1719
BT_SDP_ATTR_SUPPORTED_FEATURES_LIST (*C macro*), 1717
BT_SDP_ATTR_SUPPORTED_FORMATS_LIST (*C macro*), 1718
BT_SDP_ATTR_SUPPORTED_FUNCTIONS (*C macro*), 1719
BT_SDP_ATTR_SUPPORTED_MESSAGE_TYPES (*C macro*), 1719
BT_SDP_ATTR_SUPPORTED_REPOSITORIES (*C macro*), 1719
BT_SDP_ATTR_SVCDB_STATE (*C macro*), 1717
BT_SDP_ATTR_SVCDESC_PRIMARY (*C macro*), 1723
BT_SDP_ATTR_SVCINFO_TTL (*C macro*), 1716
BT_SDP_ATTR_SVCLASS_ID_LIST (*C macro*), 1716
BT_SDP_ATTR_SVCNAME_PRIMARY (*C macro*), 1723
BT_SDP_ATTR_TOTAL_IMAGING_DATA_CAPACITY (*C macro*), 1719
BT_SDP_ATTR_VENDOR_ID (*C macro*), 1719
BT_SDP_ATTR_VENDOR_ID_SOURCE (*C macro*), 1719
BT_SDP_ATTR_VERSION (*C macro*), 1719
BT_SDP_ATTR_VERSION_NUM_LIST (*C macro*), 1717
BT_SDP_ATTR_WAP_GATEWAY (*C macro*), 1718
BT_SDP_ATTR_WAP_STACK_TYPE (*C macro*), 1718
bt_sdp_attribute (*C struct*), 1727
bt_sdp_attribute.id (*C var*), 1728
bt_sdp_attribute.val (*C var*), 1728
BT_SDP_AUDIO_SINK_SVCLASS (*C macro*), 1712
BT_SDP_AUDIO_SOURCE_SVCLASS (*C macro*), 1712
BT_SDP_AV_REMOTE_CONTROLLER_SVCLASS (*C macro*), 1712
BT_SDP_AV_REMOTE_SVCLASS (*C macro*), 1712
BT_SDP_AV_REMOTE_TARGET_SVCLASS (*C macro*), 1712
BT_SDP_AV_SVCLASS (*C macro*), 1714
BT_SDP_BASIC_PRINTING_SVCLASS (*C macro*), 1713
BT_SDP_BOOL (*C macro*), 1722
BT_SDP_BROWSE_GRP_DESC_SVCLASS (*C macro*), 1711
BT_SDP_CIP_SVCLASS (*C macro*), 1714
bt_sdp_client_result (*C struct*), 1728
bt_sdp_client_result.next_record_hint (*C var*), 1728
bt_sdp_client_result.resp_buf (*C var*), 1728
bt_sdp_client_result.uuid (*C var*), 1728
BT_SDP_CORDLESS_TELEPHONY_SVCLASS (*C macro*), 1712
bt_sdp_data_elem (*C struct*), 1727
BT_SDP_DATA_ELEM_LIST (*C macro*), 1723
bt_sdp_data_elem.data_size (*C var*), 1727
bt_sdp_data_elem.total_size (*C var*), 1727
bt_sdp_data_elem.type (*C var*), 1727
BT_SDP_DATA_NIL (*C macro*), 1721
BT_SDP_DIALUP_NET_SVCLASS (*C macro*), 1711
BT_SDP_DIRECT_PRINTING_SVCLASS (*C macro*), 1713
BT_SDP_DIRECT_PRT_REFOBJS_SVCLASS (*C macro*), 1713
bt_sdp_discover (*C function*), 1726
bt_sdp_discover_cancel (*C function*), 1726
bt_sdp_discover_func_t (*C type*), 1724
bt_sdp_discover_params (*C struct*), 1728
bt_sdp_discover_params.func (*C var*), 1729
bt_sdp_discover_params.pool (*C var*), 1729
bt_sdp_discover_params.uuid (*C var*), 1729

BT_SDP_FAX_SVCLASS (*C macro*), 1712
BT_SDP_GENERIC_ACCESS_SVCLASS (*C macro*), 1716
BT_SDP_GENERIC_ATTRIB_SVCLASS (*C macro*), 1716
BT_SDP_GENERIC_AUDIO_SVCLASS (*C macro*), 1715
BT_SDP_GENERIC_FILETRANS_SVCLASS (*C macro*), 1715
BT_SDP_GENERIC_NETWORKING_SVCLASS (*C macro*), 1715
BT_SDP_GENERIC_TELEPHONY_SVCLASS (*C macro*), 1715
bt_sdp_get_addl_proto_param (*C function*), 1726
bt_sdp_get_features (*C function*), 1727
bt_sdp_get_profile_version (*C function*), 1727
bt_sdp_get_proto_param (*C function*), 1726
BT_SDP_GN_SVCLASS (*C macro*), 1712
BT_SDP_GNSS_SERVER_SVCLASS (*C macro*), 1714
BT_SDP_GNSS_SVCLASS (*C macro*), 1714
BT_SDP_HANDSFREE_AGW_SVCLASS (*C macro*), 1713
BT_SDP_HANDSFREE_SVCLASS (*C macro*), 1713
BT_SDP_HCR_PRINT_SVCLASS (*C macro*), 1713
BT_SDP_HCR_SCAN_SVCLASS (*C macro*), 1713
BT_SDP_HCR_SVCLASS (*C macro*), 1713
BT_SDP_HDP_SINK_SVCLASS (*C macro*), 1715
BT_SDP_HDP_SOURCE_SVCLASS (*C macro*), 1715
BT_SDP_HDP_SVCLASS (*C macro*), 1715
BT_SDP_HEADSET_AGW_SVCLASS (*C macro*), 1712
BT_SDP_HEADSET_SVCLASS (*C macro*), 1712
BT_SDP_HID_SVCLASS (*C macro*), 1713
BT_SDP_IMAGING_ARCHIVE_SVCLASS (*C macro*), 1713
BT_SDP_IMAGING_REFOBJS_SVCLASS (*C macro*), 1713
BT_SDP_IMAGING_RESPONDER_SVCLASS (*C macro*), 1713
BT_SDP_IMAGING_SVCLASS (*C macro*), 1713
BT_SDP_INT8 (*C macro*), 1721
BT_SDP_INT16 (*C macro*), 1721
BT_SDP_INT32 (*C macro*), 1721
BT_SDP_INT64 (*C macro*), 1721
BT_SDP_INT128 (*C macro*), 1721
BT_SDP_INTERCOM_SVCLASS (*C macro*), 1712
BT_SDP_IRMC_SYNC_CMD_SVCLASS (*C macro*), 1711
BT_SDP_IRMC_SYNC_SVCLASS (*C macro*), 1711
BT_SDP_LAN_ACCESS_SVCLASS (*C macro*), 1711
BT_SDP_LIST (*C macro*), 1724
BT_SDP_MAP_MCE_SVCLASS (*C macro*), 1714
BT_SDP_MAP_MSE_SVCLASS (*C macro*), 1714
BT_SDP_MAP_SVCLASS (*C macro*), 1714
BT_SDP_MPS_SC_SVCLASS (*C macro*), 1714
BT_SDP_MPS_SVCLASS (*C macro*), 1714
BT_SDP_NAP_SVCLASS (*C macro*), 1712
BT_SDP_NEW_SERVICE (*C macro*), 1724
BT_SDP_OBEX_FILETRANS_SVCLASS (*C macro*), 1711
BT_SDP_OBEX_OBJPUSH_SVCLASS (*C macro*), 1711
BT_SDP_PANU_SVCLASS (*C macro*), 1712
BT_SDP_PBAP_PCE_SVCLASS (*C macro*), 1714
BT_SDP_PBAP_PSE_SVCLASS (*C macro*), 1714
BT_SDP_PBAP_SVCLASS (*C macro*), 1714
BT_SDP_PNP_INFO_SVCLASS (*C macro*), 1715
BT_SDP_PRIMARY_LANG_BASE (*C macro*), 1723
BT_SDP_PRINTING_STATUS_SVCLASS (*C macro*), 1713
bt_sdp_proto (*C enum*), 1725
bt_sdp_proto.BT_SDP_PROTO_L2CAP (*C enumerator*), 1725
bt_sdp_proto.BT_SDP_PROTO_RFCOMM (*C enumerator*), 1725

BT_SDP_PUBLIC_BROWSE_GROUP (*C macro*), 1711
BT_SDP_RECORD (*C macro*), 1724
bt_sdp_record (*C struct*), 1728
bt_sdp_record.attr_count (*C var*), 1728
bt_sdp_record.attrs (*C var*), 1728
bt_sdp_record.handle (*C var*), 1728
bt_sdp_record.index (*C var*), 1728
bt_sdp_record.next (*C var*), 1728
BT_SDP_REFERENCE_PRINTING_SVCLASS (*C macro*), 1713
BT_SDP_REFLECTED_UI_SVCLASS (*C macro*), 1713
bt_sdp_register_service (*C function*), 1725
BT_SDP_SAP_SVCLASS (*C macro*), 1714
BT_SDP_SDP_SERVER_SVCLASS (*C macro*), 1711
BT_SDP_SEQ8 (*C macro*), 1722
BT_SDP_SEQ16 (*C macro*), 1722
BT_SDP_SEQ32 (*C macro*), 1722
BT_SDP_SEQ_UNSPEC (*C macro*), 1722
BT_SDP_SERIAL_PORT_SVCLASS (*C macro*), 1711
BT_SDP_SERVER_RECORD_HANDLE (*C macro*), 1723
BT_SDP_SERVICE_ID (*C macro*), 1724
BT_SDP_SERVICE_NAME (*C macro*), 1724
BT_SDP_SIZE_DESC_MASK (*C macro*), 1723
BT_SDP_SIZE_INDEX_OFFSET (*C macro*), 1723
BT_SDP_SUPPORTED_FEATURES (*C macro*), 1724
BT_SDP_TEXT_STR8 (*C macro*), 1722
BT_SDP_TEXT_STR16 (*C macro*), 1722
BT_SDP_TEXT_STR32 (*C macro*), 1722
BT_SDP_TEXT_STR_UNSPEC (*C macro*), 1722
BT_SDP_TYPE_DESC_MASK (*C macro*), 1723
BT_SDP_TYPE_SIZE (*C macro*), 1723
BT_SDP_TYPE_SIZE_VAR (*C macro*), 1723
BT_SDP_UDI_MT_SVCLASS (*C macro*), 1714
BT_SDP_UDI_TA_SVCLASS (*C macro*), 1714
BT_SDP_UINT8 (*C macro*), 1721
BT_SDP_UINT16 (*C macro*), 1721
BT_SDP_UINT32 (*C macro*), 1721
BT_SDP_UINT64 (*C macro*), 1721
BT_SDP_UINT128 (*C macro*), 1721
BT_SDP_UPNP_IP_SVCLASS (*C macro*), 1715
BT_SDP_UPNP_L2CAP_SVCLASS (*C macro*), 1715
BT_SDP_UPNP_LAP_SVCLASS (*C macro*), 1715
BT_SDP_UPNP_PAN_SVCLASS (*C macro*), 1715
BT_SDP_UPNP_SVCLASS (*C macro*), 1715
BT_SDP_URL_STR8 (*C macro*), 1723
BT_SDP_URL_STR16 (*C macro*), 1723
BT_SDP_URL_STR32 (*C macro*), 1723
BT_SDP_URL_STR_UNSPEC (*C macro*), 1722
BT_SDP_UUID16 (*C macro*), 1721
BT_SDP_UUID32 (*C macro*), 1722
BT_SDP_UUID128 (*C macro*), 1722
BT_SDP_UUID_UNSPEC (*C macro*), 1721
BT_SDP_VIDEO_CONF_GW_SVCLASS (*C macro*), 1714
BT_SDP_VIDEO_DISTRIBUTION_SVCLASS (*C macro*), 1715
BT_SDP_VIDEO_SINK_SVCLASS (*C macro*), 1715
BT_SDP_VIDEO_SOURCE_SVCLASS (*C macro*), 1715
BT_SDP_WAP_CLIENT_SVCLASS (*C macro*), 1712
BT_SDP_WAP_SVCLASS (*C macro*), 1712
bt_security_err (*C enum*), 2316

bt_security_err_to_str (*C function*), 2324
bt_security_err.BT_SECURITY_ERR_AUTH_FAIL (*C enumerator*), 2316
bt_security_err.BT_SECURITY_ERR_AUTH_REQUIREMENT (*C enumerator*), 2316
bt_security_err.BT_SECURITY_ERR_INVALID_PARAM (*C enumerator*), 2316
bt_security_err.BT_SECURITY_ERR_KEY_REJECTED (*C enumerator*), 2316
bt_security_err.BT_SECURITY_ERR_OOB_NOT_AVAILABLE (*C enumerator*), 2316
bt_security_err.BT_SECURITY_ERR_PAIR_NOT_ALLOWED (*C enumerator*), 2316
bt_security_err.BT_SECURITY_ERR_PAIR_NOT_SUPPORTED (*C enumerator*), 2316
bt_security_err.BT_SECURITY_ERR_PIN_OR_KEY_MISSING (*C enumerator*), 2316
bt_security_err.BT_SECURITY_ERR_SUCCESS (*C enumerator*), 2316
bt_security_err.BT_SECURITY_ERR_UNSPECIFIED (*C enumerator*), 2316
bt_security_flag (*C enum*), 2314
bt_security_flag.BT_SECURITY_FLAG_OOB (*C enumerator*), 2314
bt_security_flag.BT_SECURITY_FLAG_SC (*C enumerator*), 2314
bt_security_info (*C struct*), 2331
bt_security_info.enc_key_size (*C var*), 2331
bt_security_info.flags (*C var*), 2331
bt_security_info.level (*C var*), 2331
bt_security_t (*C enum*), 2313
bt_security_t.BT_SECURITY_FORCE_PAIR (*C enumerator*), 2314
bt_security_t.BT_SECURITY_L0 (*C enumerator*), 2313
bt_security_t.BT_SECURITY_L1 (*C enumerator*), 2314
bt_security_t.BT_SECURITY_L2 (*C enumerator*), 2314
bt_security_t.BT_SECURITY_L3 (*C enumerator*), 2314
bt_security_t.BT_SECURITY_L4 (*C enumerator*), 2314
bt_send (*C function*), 2351
bt_set_appearance (*C function*), 1970
bt_set_bondable (*C function*), 2324
bt_set_name (*C function*), 1969
bt_unpair (*C function*), 1986
bt_uuid (*C struct*), 2428
BT_UUID_16 (*C macro*), 2356
bt_uuid_16 (*C struct*), 2429
BT_UUID_16_ENCODE (*C macro*), 2356
bt_uuid_16.uuid (*C var*), 2429
bt_uuid_16.val (*C var*), 2429
BT_UUID_32 (*C macro*), 2356
bt_uuid_32 (*C struct*), 2429
BT_UUID_32_ENCODE (*C macro*), 2357
bt_uuid_32.uuid (*C var*), 2429
bt_uuid_32.val (*C var*), 2429
BT_UUID_128 (*C macro*), 2356
bt_uuid_128 (*C struct*), 2429
BT_UUID_128_ENCODE (*C macro*), 2356
bt_uuid_128.uuid (*C var*), 2429
bt_uuid_128.val (*C var*), 2429
BT_UUID_ACLS (*C macro*), 2363
BT_UUID_ACLS_VAL (*C macro*), 2363
BT_UUID_AICS (*C macro*), 2363
BT_UUID_AICS_CONTROL (*C macro*), 2410
BT_UUID_AICS_CONTROL_VAL (*C macro*), 2410
BT_UUID_AICS_DESCRIPTION (*C macro*), 2410
BT_UUID_AICS_DESCRIPTION_VAL (*C macro*), 2410
BT_UUID_AICS_GAIN_SETTINGS (*C macro*), 2410
BT_UUID_AICS_GAIN_SETTINGS_VAL (*C macro*), 2410
BT_UUID_AICS_INPUT_STATUS (*C macro*), 2410
BT_UUID_AICS_INPUT_STATUS_VAL (*C macro*), 2410
BT_UUID_AICS_INPUT_TYPE (*C macro*), 2410

BT_UUID_AICS_INPUT_TYPE_VAL (*C macro*), 2410
BT_UUID_AICS_STATE (*C macro*), 2409
BT_UUID_AICS_STATE_VAL (*C macro*), 2409
BT_UUID_AICS_VAL (*C macro*), 2363
BT_UUID_AIOS (*C macro*), 2360
BT_UUID_AIOS_VAL (*C macro*), 2360
BT_UUID_ALERT_LEVEL (*C macro*), 2368
BT_UUID_ALERT_LEVEL_VAL (*C macro*), 2368
BT_UUID_ANS (*C macro*), 2359
BT_UUID_ANS_VAL (*C macro*), 2359
BT_UUID_APPARENT_WIND_DIR (*C macro*), 2382
BT_UUID_APPARENT_WIND_DIR_VAL (*C macro*), 2382
BT_UUID_APPARENT_WIND_SPEED (*C macro*), 2381
BT_UUID_APPARENT_WIND_SPEED_VAL (*C macro*), 2381
BT_UUID_ASCS (*C macro*), 2364
BT_UUID_ASCS_ASE_CP (*C macro*), 2419
BT_UUID_ASCS_ASE_CP_VAL (*C macro*), 2419
BT_UUID_ASCS_ASE_SNK (*C macro*), 2419
BT_UUID_ASCS_ASE_SNK_VAL (*C macro*), 2419
BT_UUID_ASCS_ASE_SRC (*C macro*), 2419
BT_UUID_ASCS_ASE_SRC_VAL (*C macro*), 2419
BT_UUID_ASCS_VAL (*C macro*), 2364
BT_UUID_ATT (*C macro*), 2426
BT_UUID_ATT_VAL (*C macro*), 2426
BT_UUID_AVCTP (*C macro*), 2427
BT_UUID_AVCTP_VAL (*C macro*), 2427
BT_UUID_AVDTP (*C macro*), 2427
BT_UUID_AVDTP_VAL (*C macro*), 2427
BT_UUID_BAR_PRESSURE_TREND (*C macro*), 2388
BT_UUID_BAR_PRESSURE_TREND_VAL (*C macro*), 2388
BT_UUID_BAS (*C macro*), 2359
BT_UUID_BAS_BATTERY_CRIT_STATUS (*C macro*), 2423
BT_UUID_BAS_BATTERY_CRIT_STATUS_VAL (*C macro*), 2423
BT_UUID_BAS_BATTERY_ENERGY_STATUS (*C macro*), 2424
BT_UUID_BAS_BATTERY_ENERGY_STATUS_VAL (*C macro*), 2424
BT_UUID_BAS_BATTERY_HEALTH_INF (*C macro*), 2424
BT_UUID_BAS_BATTERY_HEALTH_INF_VAL (*C macro*), 2424
BT_UUID_BAS_BATTERY_HEALTH_STATUS (*C macro*), 2424
BT_UUID_BAS_BATTERY_HEALTH_STATUS_VAL (*C macro*), 2424
BT_UUID_BAS_BATTERY_INF (*C macro*), 2424
BT_UUID_BAS_BATTERY_INF_VAL (*C macro*), 2424
BT_UUID_BAS_BATTERY_LEVEL (*C macro*), 2370
BT_UUID_BAS_BATTERY_LEVEL_STATE (*C macro*), 2371
BT_UUID_BAS_BATTERY_LEVEL_STATE_VAL (*C macro*), 2371
BT_UUID_BAS_BATTERY_LEVEL_STATUS (*C macro*), 2424
BT_UUID_BAS_BATTERY_LEVEL_STATUS_VAL (*C macro*), 2424
BT_UUID_BAS_BATTERY_LEVEL_VAL (*C macro*), 2370
BT_UUID_BAS_BATTERY_POWER_STATE (*C macro*), 2370
BT_UUID_BAS_BATTERY_POWER_STATE_VAL (*C macro*), 2370
BT_UUID_BAS_BATTERY_TIME_STATUS (*C macro*), 2424
BT_UUID_BAS_BATTERY_TIME_STATUS_VAL (*C macro*), 2424
BT_UUID_BAS_VAL (*C macro*), 2359
BT_UUID_BASIC_AUDIO (*C macro*), 2365
BT_UUID_BASIC_AUDIO_VAL (*C macro*), 2365
BT_UUID_BASS (*C macro*), 2364
BT_UUID_BASS_CONTROL_POINT (*C macro*), 2419
BT_UUID_BASS_CONTROL_POINT_VAL (*C macro*), 2419
BT_UUID_BASS_RECV_STATE (*C macro*), 2419

BT_UUID_BASS_RECV_STATE_VAL (*C macro*), 2419
BT_UUID_BASS_VAL (*C macro*), 2364
BT_UUID_BCS (*C macro*), 2360
BT_UUID_BCS_VAL (*C macro*), 2360
BT_UUID_BMS (*C macro*), 2361
BT_UUID_BMS_CONTROL_POINT (*C macro*), 2388
BT_UUID_BMS_CONTROL_POINT_VAL (*C macro*), 2388
BT_UUID_BMS_FEATURE (*C macro*), 2388
BT_UUID_BMS_FEATURE_VAL (*C macro*), 2388
BT_UUID_BMS_VAL (*C macro*), 2361
BT_UUID_BNEP (*C macro*), 2426
BT_UUID_BNEP_VAL (*C macro*), 2426
BT_UUID_BPS (*C macro*), 2359
BT_UUID_BPS_VAL (*C macro*), 2359
BT_UUID_BROADCAST_AUDIO (*C macro*), 2365
BT_UUID_BROADCAST_AUDIO_VAL (*C macro*), 2365
BT_UUID_BSS (*C macro*), 2362
BT_UUID_BSS_VAL (*C macro*), 2362
BT_UUID_CAS (*C macro*), 2365
BT_UUID_CAS_VAL (*C macro*), 2365
BT_UUID_CCID (*C macro*), 2418
BT_UUID_CCID_VAL (*C macro*), 2418
BT_UUID_CENTRAL_ADDR_RES (*C macro*), 2388
BT_UUID_CENTRAL_ADDR_RES_VAL (*C macro*), 2388
BT_UUID_CGM_FEATURE (*C macro*), 2388
BT_UUID_CGM_FEATURE_VAL (*C macro*), 2388
BT_UUID_CGM_MEASUREMENT (*C macro*), 2388
BT_UUID_CGM_MEASUREMENT_VAL (*C macro*), 2388
BT_UUID_CGM_SESSION_RUN_TIME (*C macro*), 2389
BT_UUID_CGM_SESSION_RUN_TIME_VAL (*C macro*), 2389
BT_UUID_CGM_SESSION_START_TIME (*C macro*), 2388
BT_UUID_CGM_SESSION_START_TIME_VAL (*C macro*), 2388
BT_UUID_CGM_SPECIFIC_OPS_CONTROL_POINT (*C macro*), 2389
BT_UUID_CGM_SPECIFIC_OPS_CONTROL_POINT_VAL (*C macro*), 2389
BT_UUID_CGM_STATUS (*C macro*), 2388
BT_UUID_CGM_STATUS_VAL (*C macro*), 2388
BT_UUID_CGMS (*C macro*), 2361
BT_UUID_CGMS_VAL (*C macro*), 2361
bt_uuid_cmp (*C function*), 2428
BT_UUID_CMTP (*C macro*), 2427
BT_UUID_CMTP_VAL (*C macro*), 2427
BT_UUID_CPS (*C macro*), 2360
BT_UUID_CPS_VAL (*C macro*), 2360
bt_uuid_create (*C function*), 2428
BT_UUID_CSC (*C macro*), 2360
BT_UUID_CSC_FEATURE (*C macro*), 2379
BT_UUID_CSC_FEATURE_VAL (*C macro*), 2379
BT_UUID_CSC_MEASUREMENT (*C macro*), 2379
BT_UUID_CSC_MEASUREMENT_VAL (*C macro*), 2379
BT_UUID_CSC_VAL (*C macro*), 2360
BT_UUID_CSIS (*C macro*), 2363
BT_UUID_CSIS_RANK (*C macro*), 2411
BT_UUID_CSIS_RANK_VAL (*C macro*), 2411
BT_UUID_CSIS_SET_LOCK (*C macro*), 2411
BT_UUID_CSIS_SET_LOCK_VAL (*C macro*), 2411
BT_UUID_CSIS_SET_SIZE (*C macro*), 2411
BT_UUID_CSIS_SET_SIZE_VAL (*C macro*), 2411
BT_UUID_CSIS_SIRK (*C macro*), 2411

BT_UUID_CSIS_SIRK_VAL (*C macro*), 2411
BT_UUID_CSIS_VAL (*C macro*), 2363
BT_UUID_CTES (*C macro*), 2364
BT_UUID_CTES_VAL (*C macro*), 2364
BT_UUID_CTS (*C macro*), 2358
BT_UUID_CTS_CURRENT_TIME (*C macro*), 2373
BT_UUID_CTS_CURRENT_TIME_VAL (*C macro*), 2373
BT_UUID_CTS_VAL (*C macro*), 2358
BT_UUID_DECLARE_16 (*C macro*), 2355
BT_UUID_DECLARE_32 (*C macro*), 2355
BT_UUID_DECLARE_128 (*C macro*), 2355
BT_UUID_DESC_VALUE_CHANGED (*C macro*), 2383
BT_UUID_DESC_VALUE_CHANGED_VAL (*C macro*), 2383
BT_UUID_DEW_POINT (*C macro*), 2383
BT_UUID_DEW_POINT_VAL (*C macro*), 2383
BT_UUID_DIS (*C macro*), 2358
BT_UUID_DIS_FIRMWARE_REVISION (*C macro*), 2372
BT_UUID_DIS_FIRMWARE_REVISION_VAL (*C macro*), 2372
BT_UUID_DIS_HARDWARE_REVISION (*C macro*), 2372
BT_UUID_DIS_HARDWARE_REVISION_VAL (*C macro*), 2372
BT_UUID_DIS_MANUFACTURER_NAME (*C macro*), 2372
BT_UUID_DIS_MANUFACTURER_NAME_VAL (*C macro*), 2372
BT_UUID_DIS_MODEL_NUMBER (*C macro*), 2372
BT_UUID_DIS_MODEL_NUMBER_VAL (*C macro*), 2372
BT_UUID_DIS_PNP_ID (*C macro*), 2377
BT_UUID_DIS_PNP_ID_VAL (*C macro*), 2377
BT_UUID_DIS_SERIAL_NUMBER (*C macro*), 2372
BT_UUID_DIS_SERIAL_NUMBER_VAL (*C macro*), 2372
BT_UUID_DIS_SOFTWARE_REVISION (*C macro*), 2372
BT_UUID_DIS_SOFTWARE_REVISION_VAL (*C macro*), 2372
BT_UUID_DIS_SYSTEM_ID (*C macro*), 2372
BT_UUID_DIS_SYSTEM_ID_VAL (*C macro*), 2372
BT_UUID_DIS_VAL (*C macro*), 2358
BT_UUID_DTS (*C macro*), 2363
BT_UUID_DTS_VAL (*C macro*), 2363
BT_UUID_ECS (*C macro*), 2362
BT_UUID_ECS_VAL (*C macro*), 2362
BT_UUID_ELEVATION (*C macro*), 2381
BT_UUID_ELEVATION_VAL (*C macro*), 2381
BT_UUID_ES_CONFIGURATION (*C macro*), 2367
BT_UUID_ES_CONFIGURATION_VAL (*C macro*), 2367
BT_UUID_ES_MEASUREMENT (*C macro*), 2367
BT_UUID_ES_MEASUREMENT_VAL (*C macro*), 2367
BT_UUID_ES_TRIGGER_SETTING (*C macro*), 2367
BT_UUID_ES_TRIGGER_SETTING_VAL (*C macro*), 2367
BT_UUID_ESS (*C macro*), 2360
BT_UUID_ESS_VAL (*C macro*), 2360
BT_UUID_FMS (*C macro*), 2362
BT_UUID_FMS_VAL (*C macro*), 2362
BT_UUID_FTP (*C macro*), 2426
BT_UUID_FTP_VAL (*C macro*), 2426
BT_UUID_GAP (*C macro*), 2357
BT_UUID_GAP_APPEARANCE (*C macro*), 2368
BT_UUID_GAP_APPEARANCE_VAL (*C macro*), 2368
BT_UUID_GAP_DEVICE_NAME (*C macro*), 2368
BT_UUID_GAP_DEVICE_NAME_VAL (*C macro*), 2368
BT_UUID_GAP_PPCP (*C macro*), 2368
BT_UUID_GAP_PPCP_VAL (*C macro*), 2368

BT_UUID_GAP_PPF (*C macro*), 2368
BT_UUID_GAP_PPF_VAL (*C macro*), 2368
BT_UUID_GAP_RA (*C macro*), 2368
BT_UUID_GAP_RA_VAL (*C macro*), 2368
BT_UUID_GAP_VAL (*C macro*), 2357
BT_UUID_GATT (*C macro*), 2357
BT_UUID_GATT_2ZHRL (*C macro*), 2386
BT_UUID_GATT_2ZHRL_VAL (*C macro*), 2386
BT_UUID_GATT_3ZHRL (*C macro*), 2386
BT_UUID_GATT_3ZHRL_VAL (*C macro*), 2386
BT_UUID_GATT_4ZHRL (*C macro*), 2409
BT_UUID_GATT_4ZHRL_VAL (*C macro*), 2409
BT_UUID_GATT_5ZHRL (*C macro*), 2385
BT_UUID_GATT_5ZHRL_VAL (*C macro*), 2385
BT_UUID_GATT_AC (*C macro*), 2395
BT_UUID_GATT_AC_VAL (*C macro*), 2395
BT_UUID_GATT_ACS (*C macro*), 2408
BT_UUID_GATT_ACS_CP (*C macro*), 2406
BT_UUID_GATT_ACS_CP_VAL (*C macro*), 2406
BT_UUID_GATT_ACS_DI (*C macro*), 2405
BT_UUID_GATT_ACS_DI_VAL (*C macro*), 2405
BT_UUID_GATT_ACS_DOI (*C macro*), 2405
BT_UUID_GATT_ACS_DOI_VAL (*C macro*), 2405
BT_UUID_GATT_ACS_DON (*C macro*), 2405
BT_UUID_GATT_ACS_DON_VAL (*C macro*), 2405
BT_UUID_GATT_ACS_S (*C macro*), 2405
BT_UUID_GATT_ACS_S_VAL (*C macro*), 2405
BT_UUID_GATT_ACS_VAL (*C macro*), 2408
BT_UUID_GATT_ACTEI (*C macro*), 2417
BT_UUID_GATT_ACTEI_VAL (*C macro*), 2417
BT_UUID_GATT_ACTEML (*C macro*), 2416
BT_UUID_GATT_ACTEML_VAL (*C macro*), 2416
BT_UUID_GATT_ACTEMTC (*C macro*), 2416
BT_UUID_GATT_ACTEMTC_VAL (*C macro*), 2416
BT_UUID_GATT_ACTEP (*C macro*), 2417
BT_UUID_GATT_ACTEP_VAL (*C macro*), 2417
BT_UUID_GATT_ACTETD (*C macro*), 2416
BT_UUID_GATT_ACTETD_VAL (*C macro*), 2416
BT_UUID_GATT_AE32 (*C macro*), 2412
BT_UUID_GATT_AE32_VAL (*C macro*), 2412
BT_UUID_GATT_AEANTHR (*C macro*), 2386
BT_UUID_GATT_AEANTHR_VAL (*C macro*), 2386
BT_UUID_GATT_AEHRLL (*C macro*), 2383
BT_UUID_GATT_AEHRLL_VAL (*C macro*), 2383
BT_UUID_GATT_AEHRUL (*C macro*), 2384
BT_UUID_GATT_AEHRUL_VAL (*C macro*), 2384
BT_UUID_GATT_AETHR (*C macro*), 2383
BT_UUID_GATT_AETHR_VAL (*C macro*), 2383
BT_UUID_GATT_AG (*C macro*), 2409
BT_UUID_GATT_AG_VAL (*C macro*), 2409
BT_UUID_GATT_AGE (*C macro*), 2383
BT_UUID_GATT_AGE_VAL (*C macro*), 2383
BT_UUID_GATT_AGGR (*C macro*), 2378
BT_UUID_GATT_AGGR_VAL (*C macro*), 2378
BT_UUID_GATT_AI (*C macro*), 2378
BT_UUID_GATT_AI_VAL (*C macro*), 2378
BT_UUID_GATT_ALRTCID (*C macro*), 2376
BT_UUID_GATT_ALRTCID_MASK (*C macro*), 2375

BT_UUID_GATT_ALRTCID_MASK_VAL (*C macro*), 2375
BT_UUID_GATT_ALRTCID_VAL (*C macro*), 2376
BT_UUID_GATT_ALRTNCP (*C macro*), 2376
BT_UUID_GATT_ALRTNCP_VAL (*C macro*), 2376
BT_UUID_GATT_ALRTS (*C macro*), 2375
BT_UUID_GATT_ALRTS_VAL (*C macro*), 2375
BT_UUID_GATT_ALT (*C macro*), 2390
BT_UUID_GATT_ALT_VAL (*C macro*), 2390
BT_UUID_GATT_ANHRLL (*C macro*), 2383
BT_UUID_GATT_ANHRLL_VAL (*C macro*), 2383
BT_UUID_GATT_ANHRUL (*C macro*), 2383
BT_UUID_GATT_ANHRUL_VAL (*C macro*), 2383
BT_UUID_GATT_ANTHR (*C macro*), 2384
BT_UUID_GATT_ANTHR_VAL (*C macro*), 2384
BT_UUID_GATT_AO (*C macro*), 2378
BT_UUID_GATT_AO_VAL (*C macro*), 2378
BT_UUID_GATT_AP (*C macro*), 2412
BT_UUID_GATT_AP_VAL (*C macro*), 2412
BT_UUID_GATT_AV (*C macro*), 2395
BT_UUID_GATT_AV_VAL (*C macro*), 2395
BT_UUID_GATT_BCF (*C macro*), 2387
BT_UUID_GATT_BCF_VAL (*C macro*), 2387
BT_UUID_GATT_BCM (*C macro*), 2387
BT_UUID_GATT_BCM_VAL (*C macro*), 2387
BT_UUID_GATT_BOOLEAN (*C macro*), 2395
BT_UUID_GATT_BOOLEAN_VAL (*C macro*), 2395
BT_UUID_GATT_BPF (*C macro*), 2376
BT_UUID_GATT_BPF_VAL (*C macro*), 2376
BT_UUID_GATT_BPM (*C macro*), 2374
BT_UUID_GATT_BPM_VAL (*C macro*), 2374
BT_UUID_GATT_BPR (*C macro*), 2406
BT_UUID_GATT_BPR_VAL (*C macro*), 2406
BT_UUID_GATT_BR_EDR_HD (*C macro*), 2406
BT_UUID_GATT_BR_EDR_HD_VAL (*C macro*), 2406
BT_UUID_GATT_BSS_CP (*C macro*), 2405
BT_UUID_GATT_BSS_CP_VAL (*C macro*), 2405
BT_UUID_GATT_BSS_R (*C macro*), 2405
BT_UUID_GATT_BSS_R_VAL (*C macro*), 2405
BT_UUID_GATT_BT_SIG_D (*C macro*), 2406
BT_UUID_GATT_BT_SIG_D_VAL (*C macro*), 2406
BT_UUID_GATT_CAF (*C macro*), 2366
BT_UUID_GATT_CAF_VAL (*C macro*), 2366
BT_UUID_GATT_CCC (*C macro*), 2366
BT_UUID_GATT_CCC_VAL (*C macro*), 2366
BT_UUID_GATT_CCTEMP (*C macro*), 2396
BT_UUID_GATT_CCTEMP_VAL (*C macro*), 2396
BT_UUID_GATT_CEP (*C macro*), 2366
BT_UUID_GATT_CEP_VAL (*C macro*), 2366
BT_UUID_GATT_CH4CONC (*C macro*), 2421
BT_UUID_GATT_CH4CONC_VAL (*C macro*), 2421
BT_UUID_GATT_CHRC (*C macro*), 2366
BT_UUID_GATT_CHRC_VAL (*C macro*), 2366
BT_UUID_GATT_CI (*C macro*), 2409
BT_UUID_GATT_CI_VAL (*C macro*), 2409
BT_UUID_GATT_CIEIDX (*C macro*), 2396
BT_UUID_GATT_CIEIDX_VAL (*C macro*), 2396
BT_UUID_GATT_CLIENT_FEATURES (*C macro*), 2404
BT_UUID_GATT_CLIENT_FEATURES_VAL (*C macro*), 2404

BT_UUID_GATT_CNTRCODE (*C macro*), 2397
BT_UUID_GATT_CNTRCODE_VAL (*C macro*), 2397
BT_UUID_GATT_CO2CONC (*C macro*), 2412
BT_UUID_GATT_CO2CONC_VAL (*C macro*), 2412
BT_UUID_GATT_COCONC (*C macro*), 2420
BT_UUID_GATT_COCONC_VAL (*C macro*), 2420
BT_UUID_GATT_COEFFICIENT (*C macro*), 2396
BT_UUID_GATT_COEFFICIENT_VAL (*C macro*), 2396
BT_UUID_GATT_COS (*C macro*), 2412
BT_UUID_GATT_COS_VAL (*C macro*), 2412
BT_UUID_GATT_COUNT16 (*C macro*), 2396
BT_UUID_GATT_COUNT16_VAL (*C macro*), 2396
BT_UUID_GATT_COUNT24 (*C macro*), 2397
BT_UUID_GATT_COUNT24_VAL (*C macro*), 2397
BT_UUID_GATT_CPF (*C macro*), 2366
BT_UUID_GATT_CPF_VAL (*C macro*), 2366
BT_UUID_GATT_CPS_CPCP (*C macro*), 2380
BT_UUID_GATT_CPS_CPCP_VAL (*C macro*), 2380
BT_UUID_GATT_CPS_CPF (*C macro*), 2380
BT_UUID_GATT_CPS_CPF_VAL (*C macro*), 2380
BT_UUID_GATT_CPS_CPM (*C macro*), 2380
BT_UUID_GATT_CPS_CPM_VAL (*C macro*), 2380
BT_UUID_GATT_CPS_CPV (*C macro*), 2380
BT_UUID_GATT_CPS_CPV_VAL (*C macro*), 2380
BT_UUID_GATT_CR_AID (*C macro*), 2407
BT_UUID_GATT_CR_AID_VAL (*C macro*), 2407
BT_UUID_GATT_CR_ASD (*C macro*), 2407
BT_UUID_GATT_CR_ASD_VAL (*C macro*), 2407
BT_UUID_GATT_CRCCT (*C macro*), 2396
BT_UUID_GATT_CRCCT_VAL (*C macro*), 2396
BT_UUID_GATT_CRCOORD (*C macro*), 2403
BT_UUID_GATT_CRCOORD_VAL (*C macro*), 2403
BT_UUID_GATT_CRCOORDS (*C macro*), 2396
BT_UUID_GATT_CRCOORDS_VAL (*C macro*), 2396
BT_UUID_GATT_CRDFP (*C macro*), 2396
BT_UUID_GATT_CRDFP_VAL (*C macro*), 2396
BT_UUID_GATT_CRT (*C macro*), 2396
BT_UUID_GATT_CRT_VAL (*C macro*), 2396
BT_UUID_GATT_CTD (*C macro*), 2393
BT_UUID_GATT_CTD_VAL (*C macro*), 2393
BT_UUID_GATT_CTEE (*C macro*), 2416
BT_UUID_GATT_CTEE_VAL (*C macro*), 2416
BT_UUID_GATT_CUD (*C macro*), 2366
BT_UUID_GATT_CUD_VAL (*C macro*), 2366
BT_UUID_GATT_DATE_BIRTH (*C macro*), 2384
BT_UUID_GATT_DATE_BIRTH_VAL (*C macro*), 2384
BT_UUID_GATT_DATE_THRASS (*C macro*), 2384
BT_UUID_GATT_DATE_THRASS_VAL (*C macro*), 2384
BT_UUID_GATT_DATEUTC (*C macro*), 2397
BT_UUID_GATT_DATEUTC_VAL (*C macro*), 2397
BT_UUID_GATT_DB_HASH (*C macro*), 2404
BT_UUID_GATT_DB_HASH_VAL (*C macro*), 2404
BT_UUID_GATT_DBCHINC (*C macro*), 2386
BT_UUID_GATT_DBCHINC_VAL (*C macro*), 2386
BT_UUID_GATT_DDT (*C macro*), 2369
BT_UUID_GATT_DDT_VAL (*C macro*), 2369
BT_UUID_GATT_DEVICE_WP (*C macro*), 2409
BT_UUID_GATT_DEVICE_WP_VAL (*C macro*), 2409

BT_UUID_GATT_DEVT (*C macro*), 2412
BT_UUID_GATT_DEVT_VAL (*C macro*), 2412
BT_UUID_GATT_DEVTCP (*C macro*), 2413
BT_UUID_GATT_DEVTCP_VAL (*C macro*), 2413
BT_UUID_GATT_DEVTF (*C macro*), 2412
BT_UUID_GATT_DEVTF_VAL (*C macro*), 2412
BT_UUID_GATT_DEVTP (*C macro*), 2412
BT_UUID_GATT_DEVTP_VAL (*C macro*), 2412
BT_UUID_GATT_DI (*C macro*), 2378
BT_UUID_GATT_DI_VAL (*C macro*), 2378
BT_UUID_GATT_DO (*C macro*), 2378
BT_UUID_GATT_DO_VAL (*C macro*), 2378
BT_UUID_GATT_DST (*C macro*), 2369
BT_UUID_GATT_DST_VAL (*C macro*), 2369
BT_UUID_GATT_DT (*C macro*), 2369
BT_UUID_GATT_DT_VAL (*C macro*), 2369
BT_UUID_GATT_DW (*C macro*), 2369
BT_UUID_GATT_DW_VAL (*C macro*), 2369
BT_UUID_GATT_E32 (*C macro*), 2415
BT_UUID_GATT_E32_VAL (*C macro*), 2415
BT_UUID_GATT_EBPM (*C macro*), 2406
BT_UUID_GATT_EBPM_VAL (*C macro*), 2406
BT_UUID_GATT_EC (*C macro*), 2397
BT_UUID_GATT_EC_VAL (*C macro*), 2397
BT_UUID_GATT_ECR (*C macro*), 2397
BT_UUID_GATT_ECR_VAL (*C macro*), 2397
BT_UUID_GATT_ECSPEC (*C macro*), 2397
BT_UUID_GATT_ECSPEC_VAL (*C macro*), 2397
BT_UUID_GATT_ECSTAT (*C macro*), 2397
BT_UUID_GATT_ECSTAT_VAL (*C macro*), 2397
BT_UUID_GATT_EDKM (*C macro*), 2412
BT_UUID_GATT_EDKM_VAL (*C macro*), 2412
BT_UUID_GATT_EICP (*C macro*), 2406
BT_UUID_GATT_EICP_VAL (*C macro*), 2406
BT_UUID_GATT_EMAIL (*C macro*), 2384
BT_UUID_GATT_EMAIL_VAL (*C macro*), 2384
BT_UUID_GATT_EMG_ID (*C macro*), 2405
BT_UUID_GATT_EMG_ID_VAL (*C macro*), 2405
BT_UUID_GATT_EMG_TXT (*C macro*), 2405
BT_UUID_GATT_EMG_TXT_VAL (*C macro*), 2405
BT_UUID_GATT_ENERGY (*C macro*), 2397
BT_UUID_GATT_ENERGY_VAL (*C macro*), 2397
BT_UUID_GATT_EPOD (*C macro*), 2398
BT_UUID_GATT_EPOD_VAL (*C macro*), 2398
BT_UUID_GATT_ESD (*C macro*), 2424
BT_UUID_GATT_ESD_VAL (*C macro*), 2424
BT_UUID_GATT_ET256 (*C macro*), 2369
BT_UUID_GATT_ET256_VAL (*C macro*), 2369
BT_UUID_GATT_EVTSTAT (*C macro*), 2398
BT_UUID_GATT_EVTSTAT_VAL (*C macro*), 2398
BT_UUID_GATT_FBHRLL (*C macro*), 2384
BT_UUID_GATT_FBHRLL_VAL (*C macro*), 2384
BT_UUID_GATT_FBHRUL (*C macro*), 2384
BT_UUID_GATT_FBHRUL_VAL (*C macro*), 2384
BT_UUID_GATT_FIRST_NAME (*C macro*), 2384
BT_UUID_GATT_FIRST_NAME_VAL (*C macro*), 2384
BT_UUID_GATT_FMCP (*C macro*), 2394
BT_UUID_GATT_FMCP_VAL (*C macro*), 2394

BT_UUID_GATT_FMF (*C macro*), 2393
BT_UUID_GATT_FMF_VAL (*C macro*), 2393
BT_UUID_GATT_FMS (*C macro*), 2394
BT_UUID_GATT_FMS_VAL (*C macro*), 2394
BT_UUID_GATT_FN (*C macro*), 2389
BT_UUID_GATT_FN_VAL (*C macro*), 2389
BT_UUID_GATT_FSTR8 (*C macro*), 2398
BT_UUID_GATT_FSTR8_VAL (*C macro*), 2398
BT_UUID_GATT_FSTR16 (*C macro*), 2398
BT_UUID_GATT_FSTR16_VAL (*C macro*), 2398
BT_UUID_GATT_FSTR24 (*C macro*), 2398
BT_UUID_GATT_FSTR24_VAL (*C macro*), 2398
BT_UUID_GATT_FSTR36 (*C macro*), 2398
BT_UUID_GATT_FSTR36_VAL (*C macro*), 2398
BT_UUID_GATT_FSTR64 (*C macro*), 2422
BT_UUID_GATT_FSTR64_VAL (*C macro*), 2422
BT_UUID_GATT_GEN_AID (*C macro*), 2407
BT_UUID_GATT_GEN_AID_VAL (*C macro*), 2407
BT_UUID_GATT_GEN_ASD (*C macro*), 2407
BT_UUID_GATT_GEN_ASD_VAL (*C macro*), 2407
BT_UUID_GATT_GENDER (*C macro*), 2385
BT_UUID_GATT_GENDER_VAL (*C macro*), 2385
BT_UUID_GATT_GENLVL (*C macro*), 2398
BT_UUID_GATT_GENLVL_VAL (*C macro*), 2398
BT_UUID_GATT_GF (*C macro*), 2377
BT_UUID_GATT_GF_VAL (*C macro*), 2377
BT_UUID_GATT_GM (*C macro*), 2370
BT_UUID_GATT_GM_VAL (*C macro*), 2370
BT_UUID_GATT_GMC (*C macro*), 2374
BT_UUID_GATT_GMC_VAL (*C macro*), 2374
BT_UUID_GATT_GTIN (*C macro*), 2398
BT_UUID_GATT_GTIN_VAL (*C macro*), 2398
BT_UUID_GATT_HANDEDNESS (*C macro*), 2408
BT_UUID_GATT_HANDEDNESS_VAL (*C macro*), 2408
BT_UUID_GATT_HC (*C macro*), 2385
BT_UUID_GATT_HC_VAL (*C macro*), 2385
BT_UUID_GATT_HEIGHT (*C macro*), 2385
BT_UUID_GATT_HEIGHT_VAL (*C macro*), 2385
BT_UUID_GATT_HIET (*C macro*), 2409
BT_UUID_GATT_HIET_VAL (*C macro*), 2409
BT_UUID_GATT_HITEMP (*C macro*), 2422
BT_UUID_GATT_HITEMP_VAL (*C macro*), 2422
BT_UUID_GATT_HR_MAX (*C macro*), 2385
BT_UUID_GATT_HR_MAX_VAL (*C macro*), 2385
BT_UUID_GATT_HRES_H (*C macro*), 2408
BT_UUID_GATT_HRES_H_VAL (*C macro*), 2408
BT_UUID_GATT_HV (*C macro*), 2422
BT_UUID_GATT_HV_VAL (*C macro*), 2422
BT_UUID_GATT_IBD (*C macro*), 2393
BT_UUID_GATT_IBD_VAL (*C macro*), 2393
BT_UUID_GATT_ICP (*C macro*), 2374
BT_UUID_GATT_ICP_VAL (*C macro*), 2374
BT_UUID_GATT_IDD_AS (*C macro*), 2403
BT_UUID_GATT_IDD_AS_VAL (*C macro*), 2403
BT_UUID_GATT_IDD_CCP (*C macro*), 2404
BT_UUID_GATT_IDD_CCP_VAL (*C macro*), 2404
BT_UUID_GATT_IDD_CD (*C macro*), 2404
BT_UUID_GATT_IDD_CD_VAL (*C macro*), 2404

BT_UUID_GATT_IDD_F (*C macro*), 2404
BT_UUID_GATT_IDD_F_VAL (*C macro*), 2404
BT_UUID_GATT_IDD_HD (*C macro*), 2404
BT_UUID_GATT_IDD_HD_VAL (*C macro*), 2404
BT_UUID_GATT_IDD_RACP (*C macro*), 2404
BT_UUID_GATT_IDD_RACP_VAL (*C macro*), 2404
BT_UUID_GATT_IDD_S (*C macro*), 2403
BT_UUID_GATT_IDD_S_VAL (*C macro*), 2403
BT_UUID_GATT_IDD_SC (*C macro*), 2403
BT_UUID_GATT_IDD_SC_VAL (*C macro*), 2403
BT_UUID_GATT_IDD_SRCP (*C macro*), 2404
BT_UUID_GATT_IDD_SRCP_VAL (*C macro*), 2404
BT_UUID_GATT_IEEE_RCDL (*C macro*), 2372
BT_UUID_GATT_IEEE_RCDL_VAL (*C macro*), 2372
BT_UUID_GATT_ILLUM (*C macro*), 2399
BT_UUID_GATT_ILLUM_VAL (*C macro*), 2399
BT_UUID_GATT_INCLUDE (*C macro*), 2366
BT_UUID_GATT_INCLUDE_VAL (*C macro*), 2366
BT_UUID_GATT_IPC (*C macro*), 2389
BT_UUID_GATT_IPC_VAL (*C macro*), 2389
BT_UUID_GATT_LANG (*C macro*), 2387
BT_UUID_GATT_LANG_VAL (*C macro*), 2387
BT_UUID_GATT_LAST_NAME (*C macro*), 2385
BT_UUID_GATT_LAST_NAME_VAL (*C macro*), 2385
BT_UUID_GATT_LAT (*C macro*), 2389
BT_UUID_GATT_LAT_VAL (*C macro*), 2389
BT_UUID_GATT_LD (*C macro*), 2422
BT_UUID_GATT_LD_VAL (*C macro*), 2422
BT_UUID_GATT_LECOORD (*C macro*), 2389
BT_UUID_GATT_LECOORD_VAL (*C macro*), 2389
BT_UUID_GATT_LLAT (*C macro*), 2373
BT_UUID_GATT_LLAT_VAL (*C macro*), 2373
BT_UUID_GATT_LLON (*C macro*), 2373
BT_UUID_GATT_LLON_VAL (*C macro*), 2373
BT_UUID_GATT_LNCOORD (*C macro*), 2389
BT_UUID_GATT_LNCOORD_VAL (*C macro*), 2389
BT_UUID_GATT_LNCP (*C macro*), 2381
BT_UUID_GATT_LNCP_VAL (*C macro*), 2381
BT_UUID_GATT_LNF (*C macro*), 2380
BT_UUID_GATT_LNF_VAL (*C macro*), 2380
BT_UUID_GATT_LO (*C macro*), 2423
BT_UUID_GATT_LO_VAL (*C macro*), 2423
BT_UUID_GATT_LOC_NAME (*C macro*), 2390
BT_UUID_GATT_LOC_NAME_VAL (*C macro*), 2390
BT_UUID_GATT_LOC_SPD (*C macro*), 2380
BT_UUID_GATT_LOC_SPD_VAL (*C macro*), 2380
BT_UUID_GATT_LON (*C macro*), 2389
BT_UUID_GATT_LON_VAL (*C macro*), 2389
BT_UUID_GATT_LST (*C macro*), 2423
BT_UUID_GATT_LST_VAL (*C macro*), 2423
BT_UUID_GATT_LTI (*C macro*), 2369
BT_UUID_GATT_LTI_VAL (*C macro*), 2369
BT_UUID_GATT_LUMEFF (*C macro*), 2399
BT_UUID_GATT_LUMEFF_VAL (*C macro*), 2399
BT_UUID_GATT_LUMEXP (*C macro*), 2399
BT_UUID_GATT_LUMEXP_VAL (*C macro*), 2399
BT_UUID_GATT_LUMFLX (*C macro*), 2399
BT_UUID_GATT_LUMFLX_VAL (*C macro*), 2399

BT_UUID_GATT_LUMFLXR (*C macro*), 2399
BT_UUID_GATT_LUMFLXR_VAL (*C macro*), 2399
BT_UUID_GATT_LUMINT (*C macro*), 2399
BT_UUID_GATT_LUMINT_VAL (*C macro*), 2399
BT_UUID_GATT_LUMNRG (*C macro*), 2399
BT_UUID_GATT_LUMNRG_VAL (*C macro*), 2399
BT_UUID_GATT_MASSFLOW (*C macro*), 2399
BT_UUID_GATT_MASSFLOW_VAL (*C macro*), 2399
BT_UUID_GATT_MID_NAME (*C macro*), 2408
BT_UUID_GATT_MID_NAME_VAL (*C macro*), 2408
BT_UUID_GATT_MRHR (*C macro*), 2385
BT_UUID_GATT_MRHR_VAL (*C macro*), 2385
BT_UUID_GATT_NALRT (*C macro*), 2376
BT_UUID_GATT_NALRT_VAL (*C macro*), 2376
BT_UUID_GATT_NAV (*C macro*), 2380
BT_UUID_GATT_NAV_VAL (*C macro*), 2380
BT_UUID_GATT_NETA (*C macro*), 2375
BT_UUID_GATT_NETA_VAL (*C macro*), 2375
BT_UUID_GATT_NH4CONC (*C macro*), 2420
BT_UUID_GATT_NH4CONC_VAL (*C macro*), 2420
BT_UUID_GATT_NNN (*C macro*), 2395
BT_UUID_GATT_NNN_VAL (*C macro*), 2395
BT_UUID_GATT_NO2CONC (*C macro*), 2421
BT_UUID_GATT_NO2CONC_VAL (*C macro*), 2421
BT_UUID_GATT_NOISE (*C macro*), 2423
BT_UUID_GATT_NOISE_VAL (*C macro*), 2423
BT_UUID_GATT_NONCH4CONC (*C macro*), 2421
BT_UUID_GATT_NONCH4CONC_VAL (*C macro*), 2421
BT_UUID_GATT_O3CONC (*C macro*), 2421
BT_UUID_GATT_O3CONC_VAL (*C macro*), 2421
BT_UUID_GATT_PER8 (*C macro*), 2400
BT_UUID_GATT_PER8_VAL (*C macro*), 2400
BT_UUID_GATT_PERLGHT (*C macro*), 2400
BT_UUID_GATT_PERLGHT_VAL (*C macro*), 2400
BT_UUID_GATT_PHY_AMCP (*C macro*), 2408
BT_UUID_GATT_PHY_AMCP_VAL (*C macro*), 2408
BT_UUID_GATT_PHY_AMF (*C macro*), 2407
BT_UUID_GATT_PHY_AMF_VAL (*C macro*), 2407
BT_UUID_GATT_PHY_ASDESC (*C macro*), 2408
BT_UUID_GATT_PHY_ASDESC_VAL (*C macro*), 2408
BT_UUID_GATT_PLX_CM (*C macro*), 2379
BT_UUID_GATT_PLX_CM_VAL (*C macro*), 2379
BT_UUID_GATT_PLX_F (*C macro*), 2379
BT_UUID_GATT_PLX_F_VAL (*C macro*), 2379
BT_UUID_GATT_PLX_SCM (*C macro*), 2379
BT_UUID_GATT_PLX_SCM_VAL (*C macro*), 2379
BT_UUID_GATT_PM1CONC (*C macro*), 2421
BT_UUID_GATT_PM1CONC_VAL (*C macro*), 2421
BT_UUID_GATT_PM10CONC (*C macro*), 2421
BT_UUID_GATT_PM10CONC_VAL (*C macro*), 2421
BT_UUID_GATT_PM25CONC (*C macro*), 2421
BT_UUID_GATT_PM25CONC_VAL (*C macro*), 2421
BT_UUID_GATT_POCP (*C macro*), 2379
BT_UUID_GATT_POCP_VAL (*C macro*), 2379
BT_UUID_GATT_POPE (*C macro*), 2379
BT_UUID_GATT_POPE_VAL (*C macro*), 2379
BT_UUID_GATT_POS_2D (*C macro*), 2373
BT_UUID_GATT_POS_2D_VAL (*C macro*), 2373

BT_UUID_GATT_POS_3D (*C macro*), 2373
BT_UUID_GATT_POS_3D_VAL (*C macro*), 2373
BT_UUID_GATT_PQ (*C macro*), 2380
BT_UUID_GATT_PQ_VAL (*C macro*), 2380
BT_UUID_GATT_PREF_U (*C macro*), 2408
BT_UUID_GATT_PREF_U_VAL (*C macro*), 2408
BT_UUID_GATT_PRIMARY (*C macro*), 2365
BT_UUID_GATT_PRIMARY_VAL (*C macro*), 2365
BT_UUID_GATT_PWR (*C macro*), 2400
BT_UUID_GATT_PWR_VAL (*C macro*), 2400
BT_UUID_GATT_PWRSPEC (*C macro*), 2400
BT_UUID_GATT_PWRSPEC_VAL (*C macro*), 2400
BT_UUID_GATT_RCCP (*C macro*), 2403
BT_UUID_GATT_RCCP_VAL (*C macro*), 2403
BT_UUID_GATT_RCF (*C macro*), 2403
BT_UUID_GATT_RCF_VAL (*C macro*), 2403
BT_UUID_GATT_RCP (*C macro*), 2375
BT_UUID_GATT_RCP_VAL (*C macro*), 2375
BT_UUID_GATT_RCSET (*C macro*), 2403
BT_UUID_GATT_RCSET_VAL (*C macro*), 2403
BT_UUID_GATT_RD (*C macro*), 2393
BT_UUID_GATT_RD_VAL (*C macro*), 2393
BT_UUID_GATT_REM (*C macro*), 2374
BT_UUID_GATT_REM_VAL (*C macro*), 2374
BT_UUID_GATT_RHR (*C macro*), 2385
BT_UUID_GATT_RHR_VAL (*C macro*), 2385
BT_UUID_GATT_RPAO (*C macro*), 2392
BT_UUID_GATT_RPAO_VAL (*C macro*), 2392
BT_UUID_GATT_RRCCTP_VAL (*C macro*), 2423
BT_UUID_GATT_RRCCTR (*C macro*), 2423
BT_UUID_GATT_RRICR (*C macro*), 2400
BT_UUID_GATT_RRICR_VAL (*C macro*), 2400
BT_UUID_GATT_RRIGLR (*C macro*), 2400
BT_UUID_GATT_RRIGLR_VAL (*C macro*), 2400
BT_UUID_GATT_RS (*C macro*), 2375
BT_UUID_GATT_RS_VAL (*C macro*), 2375
BT_UUID_GATT_RTI (*C macro*), 2370
BT_UUID_GATT_RTI_VAL (*C macro*), 2370
BT_UUID_GATT_RU (*C macro*), 2406
BT_UUID_GATT_RU_VAL (*C macro*), 2406
BT_UUID_GATT_RVIIR (*C macro*), 2400
BT_UUID_GATT_RVIIR_VAL (*C macro*), 2400
BT_UUID_GATT_RVIPOD (*C macro*), 2401
BT_UUID_GATT_RVIPOD_VAL (*C macro*), 2401
BT_UUID_GATT_RVITR (*C macro*), 2401
BT_UUID_GATT_RVITR_VAL (*C macro*), 2401
BT_UUID_GATT_RVIVR (*C macro*), 2400
BT_UUID_GATT_RVIVR_VAL (*C macro*), 2400
BT_UUID_GATT_SC (*C macro*), 2368
BT_UUID_GATT_SC_ASD (*C macro*), 2407
BT_UUID_GATT_SC_ASD_VAL (*C macro*), 2407
BT_UUID_GATT_SC_TEMP_C (*C macro*), 2375
BT_UUID_GATT_SC_TEMP_C_VAL (*C macro*), 2375
BT_UUID_GATT_SC_VAL (*C macro*), 2368
BT_UUID_GATT_SCC (*C macro*), 2366
BT_UUID_GATT_SCC_VAL (*C macro*), 2366
BT_UUID_GATT_SECONDARY (*C macro*), 2365
BT_UUID_GATT_SECONDARY_VAL (*C macro*), 2365

BT_UUID_GATT_SERVER_FEATURES (*C macro*), 2406
BT_UUID_GATT_SERVER_FEATURES_VAL (*C macro*), 2406
BT_UUID_GATT_SF6CONC (*C macro*), 2422
BT_UUID_GATT_SF6CONC_VAL (*C macro*), 2422
BT_UUID_GATT_SHRR (*C macro*), 2394
BT_UUID_GATT_SHRR_VAL (*C macro*), 2394
BT_UUID_GATT_SIN (*C macro*), 2409
BT_UUID_GATT_SIN_VAL (*C macro*), 2409
BT_UUID_GATT_SIR (*C macro*), 2394
BT_UUID_GATT_SIR_VAL (*C macro*), 2394
BT_UUID_GATT_SIW (*C macro*), 2377
BT_UUID_GATT_SIW_VAL (*C macro*), 2377
BT_UUID_GATT_SL (*C macro*), 2424
BT_UUID_GATT_SL_VAL (*C macro*), 2424
BT_UUID_GATT_SLP_AID (*C macro*), 2407
BT_UUID_GATT_SLP_AID_VAL (*C macro*), 2407
BT_UUID_GATT_SLP_ASD (*C macro*), 2407
BT_UUID_GATT_SLP_ASD_VAL (*C macro*), 2407
BT_UUID_GATT_SNALRTC (*C macro*), 2376
BT_UUID_GATT_SNALRTC_VAL (*C macro*), 2376
BT_UUID_GATT_SO2CONC (*C macro*), 2421
BT_UUID_GATT_SO2CONC_VAL (*C macro*), 2421
BT_UUID_GATT_SPR (*C macro*), 2394
BT_UUID_GATT_SPR_VAL (*C macro*), 2394
BT_UUID_GATT_SR (*C macro*), 2373
BT_UUID_GATT_SR_VAL (*C macro*), 2373
BT_UUID_GATT_SRLR (*C macro*), 2394
BT_UUID_GATT_SRLR_VAL (*C macro*), 2394
BT_UUID_GATT_SRVREQ (*C macro*), 2375
BT_UUID_GATT_SRVREQ_VAL (*C macro*), 2375
BT_UUID_GATT_SSR (*C macro*), 2394
BT_UUID_GATT_SSR_VAL (*C macro*), 2394
BT_UUID_GATT_STPCD (*C macro*), 2393
BT_UUID_GATT_STPCD_VAL (*C macro*), 2393
BT_UUID_GATT_STRCD (*C macro*), 2393
BT_UUID_GATT_STRCD_VAL (*C macro*), 2393
BT_UUID_GATT_STRDLEN (*C macro*), 2408
BT_UUID_GATT_STRDLEN_VAL (*C macro*), 2408
BT_UUID_GATT_STRING (*C macro*), 2375
BT_UUID_GATT_STRING_VAL (*C macro*), 2375
BT_UUID_GATT_SUALRTC (*C macro*), 2376
BT_UUID_GATT_SUALRTC_VAL (*C macro*), 2376
BT_UUID_GATT_TA (*C macro*), 2370
BT_UUID_GATT_TA_VAL (*C macro*), 2370
BT_UUID_GATT_TCLD (*C macro*), 2413
BT_UUID_GATT_TCLD_VAL (*C macro*), 2413
BT_UUID_GATT_TD (*C macro*), 2393
BT_UUID_GATT_TD_VAL (*C macro*), 2393
BT_UUID_GATT_TDS_CP (*C macro*), 2391
BT_UUID_GATT_TDS_CP_VAL (*C macro*), 2391
BT_UUID_GATT_TDST (*C macro*), 2369
BT_UUID_GATT_TDST_VAL (*C macro*), 2369
BT_UUID_GATT_TEMP8 (*C macro*), 2401
BT_UUID_GATT_TEMP8_IPOD (*C macro*), 2401
BT_UUID_GATT_TEMP8_IPOD_VAL (*C macro*), 2401
BT_UUID_GATT_TEMP8_STAT (*C macro*), 2401
BT_UUID_GATT_TEMP8_STAT_VAL (*C macro*), 2401
BT_UUID_GATT_TEMP8_VAL (*C macro*), 2401

BT_UUID_GATT_TEMP_RNG (*C macro*), 2401
BT_UUID_GATT_TEMP_RNG_VAL (*C macro*), 2401
BT_UUID_GATT_TEMP_STAT (*C macro*), 2401
BT_UUID_GATT_TEMP_STAT_VAL (*C macro*), 2401
BT_UUID_GATT_TIM_DC8 (*C macro*), 2401
BT_UUID_GATT_TIM_DC8_VAL (*C macro*), 2401
BT_UUID_GATT_TIM_EXP8 (*C macro*), 2402
BT_UUID_GATT_TIM_EXP8_VAL (*C macro*), 2402
BT_UUID_GATT_TIM_H24 (*C macro*), 2402
BT_UUID_GATT_TIM_H24_VAL (*C macro*), 2402
BT_UUID_GATT_TIM_MS24 (*C macro*), 2402
BT_UUID_GATT_TIM_MS24_VAL (*C macro*), 2402
BT_UUID_GATT_TIM_S8 (*C macro*), 2402
BT_UUID_GATT_TIM_S8_VAL (*C macro*), 2402
BT_UUID_GATT_TIM_S16 (*C macro*), 2402
BT_UUID_GATT_TIM_S16_VAL (*C macro*), 2402
BT_UUID_GATT_TIM_S32 (*C macro*), 2423
BT_UUID_GATT_TIM_S32_VAL (*C macro*), 2423
BT_UUID_GATT_TMAPR (*C macro*), 2409
BT_UUID_GATT_TMAPR_VAL (*C macro*), 2409
BT_UUID_GATT_TREND (*C macro*), 2383
BT_UUID_GATT_TREND_VAL (*C macro*), 2383
BT_UUID_GATT_TRSTAT (*C macro*), 2394
BT_UUID_GATT_TRSTAT_VAL (*C macro*), 2394
BT_UUID_GATT_TS (*C macro*), 2370
BT_UUID_GATT_TS_VAL (*C macro*), 2370
BT_UUID_GATT_TUCP (*C macro*), 2370
BT_UUID_GATT_TUCP_VAL (*C macro*), 2370
BT_UUID_GATT_TUS (*C macro*), 2370
BT_UUID_GATT_TUS_VAL (*C macro*), 2370
BT_UUID_GATT_TZ (*C macro*), 2369
BT_UUID_GATT_TZ_VAL (*C macro*), 2369
BT_UUID_GATT_UALRTS (*C macro*), 2376
BT_UUID_GATT_UALRTS_VAL (*C macro*), 2376
BT_UUID_GATT_UNCERTAINTY (*C macro*), 2390
BT_UUID_GATT_UNCERTAINTY_VAL (*C macro*), 2390
BT_UUID_GATT_USRCP (*C macro*), 2387
BT_UUID_GATT_USRCP_VAL (*C macro*), 2387
BT_UUID_GATT_USRIDX (*C macro*), 2386
BT_UUID_GATT_USRIDX_VAL (*C macro*), 2386
BT_UUID_GATT_V (*C macro*), 2402
BT_UUID_GATT_V_SPEC (*C macro*), 2402
BT_UUID_GATT_V_SPEC_VAL (*C macro*), 2402
BT_UUID_GATT_V_STAT (*C macro*), 2402
BT_UUID_GATT_V_STAT_VAL (*C macro*), 2402
BT_UUID_GATT_V_VAL (*C macro*), 2402
BT_UUID_GATT_VAL (*C macro*), 2357
BT_UUID_GATT_VF (*C macro*), 2423
BT_UUID_GATT_VF_VAL (*C macro*), 2423
BT_UUID_GATT_VO2_MAX (*C macro*), 2386
BT_UUID_GATT_VO2_MAX_VAL (*C macro*), 2386
BT_UUID_GATT_VOCCONC (*C macro*), 2423
BT_UUID_GATT_VOCCONC_VAL (*C macro*), 2423
BT_UUID_GATT_VOLF (*C macro*), 2403
BT_UUID_GATT_VOLF_VAL (*C macro*), 2403
BT_UUID_GATT_WC (*C macro*), 2386
BT_UUID_GATT_WC_VAL (*C macro*), 2386
BT_UUID_GATT_WEIGHT (*C macro*), 2386

BT_UUID_GATT_WEIGHT_VAL (*C macro*), 2386
BT_UUID_GATT_WM (*C macro*), 2387
BT_UUID_GATT_WM_VAL (*C macro*), 2387
BT_UUID_GATT_WSF (*C macro*), 2387
BT_UUID_GATT_WSF_VAL (*C macro*), 2387
BT_UUID_GMAP_BGR_FEAT (*C macro*), 2425
BT_UUID_GMAP_BGR_FEAT_VAL (*C macro*), 2425
BT_UUID_GMAP_BGS_FEAT (*C macro*), 2425
BT_UUID_GMAP_BGS_FEAT_VAL (*C macro*), 2425
BT_UUID_GMAP_ROLE (*C macro*), 2425
BT_UUID_GMAP_ROLE_VAL (*C macro*), 2425
BT_UUID_GMAP_UGG_FEAT (*C macro*), 2425
BT_UUID_GMAP_UGG_FEAT_VAL (*C macro*), 2425
BT_UUID_GMAP_UGT_FEAT (*C macro*), 2425
BT_UUID_GMAP_UGT_FEAT_VAL (*C macro*), 2425
BT_UUID_GMAS (*C macro*), 2425
BT_UUID_GMAS_VAL (*C macro*), 2425
BT_UUID_GMCS (*C macro*), 2364
BT_UUID_GMCS_VAL (*C macro*), 2364
BT_UUID_GS (*C macro*), 2358
BT_UUID_GS_VAL (*C macro*), 2358
BT_UUID_GTBS (*C macro*), 2364
BT_UUID_GTBS_VAL (*C macro*), 2364
BT_UUID_GUST_FACTOR (*C macro*), 2382
BT_UUID_GUST_FACTOR_VAL (*C macro*), 2382
BT_UUID_HAS (*C macro*), 2365
BT_UUID_HAS_ACTIVE_PRESET_INDEX (*C macro*), 2422
BT_UUID_HAS_ACTIVE_PRESET_INDEX_VAL (*C macro*), 2422
BT_UUID_HAS_HEARING_AID_FEATURES (*C macro*), 2422
BT_UUID_HAS_HEARING_AID_FEATURES_VAL (*C macro*), 2422
BT_UUID_HAS_PRESET_CONTROL_POINT (*C macro*), 2422
BT_UUID_HAS_PRESET_CONTROL_POINT_VAL (*C macro*), 2422
BT_UUID_HAS_VAL (*C macro*), 2365
BT_UUID_HCRP_CTRL (*C macro*), 2427
BT_UUID_HCRP_CTRL_VAL (*C macro*), 2427
BT_UUID_HCRP_DATA (*C macro*), 2427
BT_UUID_HCRP_DATA_VAL (*C macro*), 2427
BT_UUID_HCRP_NOTE (*C macro*), 2427
BT_UUID_HCRP_NOTE_VAL (*C macro*), 2427
BT_UUID_HEAT_INDEX (*C macro*), 2382
BT_UUID_HEAT_INDEX_VAL (*C macro*), 2382
BT_UUID_HIDP (*C macro*), 2427
BT_UUID_HIDP_VAL (*C macro*), 2426
BT_UUID_HIDS (*C macro*), 2359
BT_UUID_HIDS_BOOT_KB_IN_REPORT (*C macro*), 2371
BT_UUID_HIDS_BOOT_KB_IN_REPORT_VAL (*C macro*), 2371
BT_UUID_HIDS_BOOT_KB_OUT_REPORT (*C macro*), 2373
BT_UUID_HIDS_BOOT_KB_OUT_REPORT_VAL (*C macro*), 2373
BT_UUID_HIDS_BOOT_MOUSE_IN_REPORT (*C macro*), 2374
BT_UUID_HIDS_BOOT_MOUSE_IN_REPORT_VAL (*C macro*), 2374
BT_UUID_HIDS_CTRL_POINT (*C macro*), 2377
BT_UUID_HIDS_CTRL_POINT_VAL (*C macro*), 2377
BT_UUID_HIDS_EXT_REPORT (*C macro*), 2367
BT_UUID_HIDS_EXT_REPORT_VAL (*C macro*), 2367
BT_UUID_HIDS_INFO (*C macro*), 2376
BT_UUID_HIDS_INFO_VAL (*C macro*), 2376
BT_UUID_HIDS_PROTOCOL_MODE (*C macro*), 2377
BT_UUID_HIDS_PROTOCOL_MODE_VAL (*C macro*), 2377

BT_UUID_HIDS_REPORT (*C macro*), 2377
BT_UUID_HIDS_REPORT_MAP (*C macro*), 2377
BT_UUID_HIDS_REPORT_MAP_VAL (*C macro*), 2377
BT_UUID_HIDS_REPORT_REF (*C macro*), 2367
BT_UUID_HIDS_REPORT_REF_VAL (*C macro*), 2367
BT_UUID_HIDS_REPORT_VAL (*C macro*), 2377
BT_UUID_HIDS_VAL (*C macro*), 2359
BT_UUID_HPS (*C macro*), 2361
BT_UUID_HPS_VAL (*C macro*), 2361
BT_UUID_HRS (*C macro*), 2359
BT_UUID_HRS_BODY_SENSOR (*C macro*), 2374
BT_UUID_HRS_BODY_SENSOR_VAL (*C macro*), 2374
BT_UUID_HRS_CONTROL_POINT (*C macro*), 2374
BT_UUID_HRS_CONTROL_POINT_VAL (*C macro*), 2374
BT_UUID_HRS_MEASUREMENT (*C macro*), 2374
BT_UUID_HRS_MEASUREMENT_VAL (*C macro*), 2374
BT_UUID_HRS_VAL (*C macro*), 2359
BT_UUID_HTS (*C macro*), 2358
BT_UUID_HTS_INTERVAL (*C macro*), 2371
BT_UUID_HTS_INTERVAL_VAL (*C macro*), 2371
BT_UUID_HTS_MEASUREMENT (*C macro*), 2371
BT_UUID_HTS_MEASUREMENT_VAL (*C macro*), 2371
BT_UUID_HTS_TEMP_C (*C macro*), 2371
BT_UUID_HTS_TEMP_C_VAL (*C macro*), 2371
BT_UUID_HTS_TEMP_F (*C macro*), 2371
BT_UUID_HTS_TEMP_F_VAL (*C macro*), 2371
BT_UUID_HTS_TEMP_INT (*C macro*), 2371
BT_UUID_HTS_TEMP_INT_VAL (*C macro*), 2371
BT_UUID_HTS_TEMP_TYP (*C macro*), 2371
BT_UUID_HTS_TEMP_TYP_VAL (*C macro*), 2371
BT_UUID_HTS_VAL (*C macro*), 2358
BT_UUID_HTTP (*C macro*), 2426
BT_UUID_HTTP_CONTROL_POINT (*C macro*), 2390
BT_UUID_HTTP_CONTROL_POINT_VAL (*C macro*), 2390
BT_UUID_HTTP_ENTITY_BODY (*C macro*), 2390
BT_UUID_HTTP_ENTITY_BODY_VAL (*C macro*), 2390
BT_UUID_HTTP_HEADERS (*C macro*), 2390
BT_UUID_HTTP_HEADERS_VAL (*C macro*), 2390
BT_UUID_HTTP_STATUS_CODE (*C macro*), 2390
BT_UUID_HTTP_STATUS_CODE_VAL (*C macro*), 2390
BT_UUID_HTTP_VAL (*C macro*), 2426
BT_UUID_HTTPS_SECURITY (*C macro*), 2391
BT_UUID_HTTPS_SECURITY_VAL (*C macro*), 2391
BT_UUID_HUMIDITY (*C macro*), 2381
BT_UUID_HUMIDITY_VAL (*C macro*), 2381
BT_UUID_IAS (*C macro*), 2357
BT_UUID_IAS_VAL (*C macro*), 2357
BT_UUID_IDS (*C macro*), 2362
BT_UUID_IDS_VAL (*C macro*), 2362
BT_UUID_INIT_16 (*C macro*), 2355
BT_UUID_INIT_32 (*C macro*), 2355
BT_UUID_INIT_128 (*C macro*), 2355
BT_UUID_IP (*C macro*), 2426
BT_UUID_IP_VAL (*C macro*), 2426
BT_UUID_IPS (*C macro*), 2361
BT_UUID_IPS_VAL (*C macro*), 2361
BT_UUID_IPSS (*C macro*), 2361
BT_UUID_IPSS_VAL (*C macro*), 2361

BT_UUID_IRRADIANCE (*C macro*), 2382
BT_UUID_IRRADIANCE_VAL (*C macro*), 2382
BT_UUID_L2CAP (*C macro*), 2427
BT_UUID_L2CAP_VAL (*C macro*), 2427
BT_UUID_LLS (*C macro*), 2357
BT_UUID_LLS_VAL (*C macro*), 2357
BT_UUID_LNS (*C macro*), 2360
BT_UUID_LNS_VAL (*C macro*), 2360
BT_UUID_MAGN_DECLINATION (*C macro*), 2373
BT_UUID_MAGN_DECLINATION_VAL (*C macro*), 2373
BT_UUID_MAGN_FLUX_DENSITY_2D (*C macro*), 2387
BT_UUID_MAGN_FLUX_DENSITY_2D_VAL (*C macro*), 2387
BT_UUID_MAGN_FLUX_DENSITY_3D (*C macro*), 2387
BT_UUID_MAGN_FLUX_DENSITY_3D_VAL (*C macro*), 2387
BT_UUID_MCAP_CTRL (*C macro*), 2427
BT_UUID_MCAP_CTRL_VAL (*C macro*), 2427
BT_UUID_MCAP_DATA (*C macro*), 2427
BT_UUID_MCAP_DATA_VAL (*C macro*), 2427
BT_UUID_MCS (*C macro*), 2363
BT_UUID_MCS_CURRENT_GROUP_OBJ_ID (*C macro*), 2414
BT_UUID_MCS_CURRENT_GROUP_OBJ_ID_VAL (*C macro*), 2414
BT_UUID_MCS_CURRENT_TRACK_OBJ_ID (*C macro*), 2414
BT_UUID_MCS_CURRENT_TRACK_OBJ_ID_VAL (*C macro*), 2414
BT_UUID_MCS_ICON_OBJ_ID (*C macro*), 2413
BT_UUID_MCS_ICON_OBJ_ID_VAL (*C macro*), 2413
BT_UUID_MCS_ICON_URL (*C macro*), 2413
BT_UUID_MCS_ICON_URL_VAL (*C macro*), 2413
BT_UUID_MCS_MEDIA_CONTROL_OPCODES (*C macro*), 2415
BT_UUID_MCS_MEDIA_CONTROL_OPCODES_VAL (*C macro*), 2415
BT_UUID_MCS_MEDIA_CONTROL_POINT (*C macro*), 2415
BT_UUID_MCS_MEDIA_CONTROL_POINT_VAL (*C macro*), 2415
BT_UUID_MCS_MEDIA_STATE (*C macro*), 2415
BT_UUID_MCS_MEDIA_STATE_VAL (*C macro*), 2415
BT_UUID_MCS_NEXT_TRACK_OBJ_ID (*C macro*), 2414
BT_UUID_MCS_NEXT_TRACK_OBJ_ID_VAL (*C macro*), 2414
BT_UUID_MCS_PARENT_GROUP_OBJ_ID (*C macro*), 2414
BT_UUID_MCS_PARENT_GROUP_OBJ_ID_VAL (*C macro*), 2414
BT_UUID_MCS_PLAYBACK_SPEED (*C macro*), 2414
BT_UUID_MCS_PLAYBACK_SPEED_VAL (*C macro*), 2414
BT_UUID_MCS_PLAYER_NAME (*C macro*), 2413
BT_UUID_MCS_PLAYER_NAME_VAL (*C macro*), 2413
BT_UUID_MCS_PLAYING_ORDER (*C macro*), 2415
BT_UUID_MCS_PLAYING_ORDER_VAL (*C macro*), 2415
BT_UUID_MCS_PLAYING_ORDERS (*C macro*), 2415
BT_UUID_MCS_PLAYING_ORDERS_VAL (*C macro*), 2415
BT_UUID_MCS_SEARCH_CONTROL_POINT (*C macro*), 2415
BT_UUID_MCS_SEARCH_CONTROL_POINT_VAL (*C macro*), 2415
BT_UUID_MCS_SEARCH_RESULTS_OBJ_ID (*C macro*), 2415
BT_UUID_MCS_SEARCH_RESULTS_OBJ_ID_VAL (*C macro*), 2415
BT_UUID_MCS_SEEKING_SPEED (*C macro*), 2414
BT_UUID_MCS_SEEKING_SPEED_VAL (*C macro*), 2414
BT_UUID_MCS_TRACK_CHANGED (*C macro*), 2413
BT_UUID_MCS_TRACK_CHANGED_VAL (*C macro*), 2413
BT_UUID_MCS_TRACK_DURATION (*C macro*), 2413
BT_UUID_MCS_TRACK_DURATION_VAL (*C macro*), 2413
BT_UUID_MCS_TRACK_POSITION (*C macro*), 2414
BT_UUID_MCS_TRACK_POSITION_VAL (*C macro*), 2414
BT_UUID_MCS_TRACK_SEGMENTS_OBJ_ID (*C macro*), 2414

BT_UUID_MCS_TRACK_SEGMENTS_OBJ_ID_VAL (*C macro*), 2414
BT_UUID_MCS_TRACK_TITLE (*C macro*), 2413
BT_UUID_MCS_TRACK_TITLE_VAL (*C macro*), 2413
BT_UUID_MCS_VAL (*C macro*), 2363
BT_UUID_MESH_PROV (*C macro*), 2362
BT_UUID_MESH_PROV_DATA_IN (*C macro*), 2395
BT_UUID_MESH_PROV_DATA_IN_VAL (*C macro*), 2395
BT_UUID_MESH_PROV_DATA_OUT (*C macro*), 2395
BT_UUID_MESH_PROV_DATA_OUT_VAL (*C macro*), 2395
BT_UUID_MESH_PROV_VAL (*C macro*), 2362
BT_UUID_MESH_PROXY (*C macro*), 2362
BT_UUID_MESH_PROXY_DATA_IN (*C macro*), 2395
BT_UUID_MESH_PROXY_DATA_IN_VAL (*C macro*), 2395
BT_UUID_MESH_PROXY_DATA_OUT (*C macro*), 2395
BT_UUID_MESH_PROXY_DATA_OUT_VAL (*C macro*), 2395
BT_UUID_MESH_PROXY_SOLICITATION_VAL (*C macro*), 2362
BT_UUID_MESH_PROXY_VAL (*C macro*), 2362
BT_UUID_MICS (*C macro*), 2364
BT_UUID_MICS_MUTE (*C macro*), 2419
BT_UUID_MICS_MUTE_VAL (*C macro*), 2419
BT_UUID_MICS_VAL (*C macro*), 2364
BT_UUID_NAS (*C macro*), 2358
BT_UUID_NAS_VAL (*C macro*), 2358
BT_UUID_NDSTS (*C macro*), 2358
BT_UUID_NDSTS_VAL (*C macro*), 2358
BT_UUID_OBEX (*C macro*), 2426
BT_UUID_OBEX_VAL (*C macro*), 2426
BT_UUID_OTS (*C macro*), 2362
BT_UUID_OTS_ACTION_CP (*C macro*), 2392
BT_UUID_OTS_ACTION_CP_VAL (*C macro*), 2392
BT_UUID_OTS_CHANGED (*C macro*), 2392
BT_UUID_OTS_CHANGED_VAL (*C macro*), 2392
BT_UUID_OTS_DIRECTORY_LISTING (*C macro*), 2393
BT_UUID_OTS_DIRECTORY_LISTING_VAL (*C macro*), 2393
BT_UUID_OTS_FEATURE (*C macro*), 2391
BT_UUID_OTS_FEATURE_VAL (*C macro*), 2391
BT_UUID_OTS_FIRST_CREATED (*C macro*), 2391
BT_UUID_OTS_FIRST_CREATED_VAL (*C macro*), 2391
BT_UUID_OTS_ID (*C macro*), 2392
BT_UUID_OTS_ID_VAL (*C macro*), 2392
BT_UUID_OTS_LAST_MODIFIED (*C macro*), 2391
BT_UUID_OTS_LAST_MODIFIED_VAL (*C macro*), 2391
BT_UUID_OTS_LIST_CP (*C macro*), 2392
BT_UUID_OTS_LIST_CP_VAL (*C macro*), 2392
BT_UUID_OTS_LIST_FILTER (*C macro*), 2392
BT_UUID_OTS_LIST_FILTER_VAL (*C macro*), 2392
BT_UUID_OTS_NAME (*C macro*), 2391
BT_UUID_OTS_NAME_VAL (*C macro*), 2391
BT_UUID_OTS_PROPERTIES (*C macro*), 2392
BT_UUID_OTS_PROPERTIES_VAL (*C macro*), 2392
BT_UUID_OTS_SIZE (*C macro*), 2391
BT_UUID_OTS_SIZE_VAL (*C macro*), 2391
BT_UUID_OTS_TYPE (*C macro*), 2391
BT_UUID_OTS_TYPE_GROUP (*C macro*), 2416
BT_UUID_OTS_TYPE_GROUP_VAL (*C macro*), 2416
BT_UUID_OTS_TYPE_MPL_ICON (*C macro*), 2416
BT_UUID_OTS_TYPE_MPL_ICON_VAL (*C macro*), 2416
BT_UUID_OTS_TYPE_TRACK (*C macro*), 2416

BT_UUID_OTS_TYPE_TRACK_SEGMENT (*C macro*), 2416
BT_UUID_OTS_TYPE_TRACK_SEGMENT_VAL (*C macro*), 2416
BT_UUID_OTS_TYPE_TRACK_VAL (*C macro*), 2416
BT_UUID_OTS_TYPE_UNSPECIFIED (*C macro*), 2392
BT_UUID_OTS_TYPE_UNSPECIFIED_VAL (*C macro*), 2392
BT_UUID_OTS_TYPE_VAL (*C macro*), 2391
BT_UUID_OTS_VAL (*C macro*), 2361
BT_UUID_PACS (*C macro*), 2364
BT_UUID_PACS_AVAILABLE_CONTEXT (*C macro*), 2420
BT_UUID_PACS_AVAILABLE_CONTEXT_VAL (*C macro*), 2420
BT_UUID_PACS_SNK (*C macro*), 2420
BT_UUID_PACS_SNK_LOC (*C macro*), 2420
BT_UUID_PACS_SNK_LOC_VAL (*C macro*), 2420
BT_UUID_PACS_SNK_VAL (*C macro*), 2420
BT_UUID_PACS_SRC (*C macro*), 2420
BT_UUID_PACS_SRC_LOC (*C macro*), 2420
BT_UUID_PACS_SRC_LOC_VAL (*C macro*), 2420
BT_UUID_PACS_SRC_VAL (*C macro*), 2420
BT_UUID_PACS_SUPPORTED_CONTEXT (*C macro*), 2420
BT_UUID_PACS_SUPPORTED_CONTEXT_VAL (*C macro*), 2420
BT_UUID_PACS_VAL (*C macro*), 2364
BT_UUID_PAMS (*C macro*), 2363
BT_UUID_PAMS_VAL (*C macro*), 2363
BT_UUID_PAS (*C macro*), 2359
BT_UUID_PAS_VAL (*C macro*), 2359
BT_UUID_PBA (*C macro*), 2365
BT_UUID_PBA_VAL (*C macro*), 2365
BT_UUID_POLLEN_CONCENTRATION (*C macro*), 2382
BT_UUID_POLLEN_CONCENTRATION_VAL (*C macro*), 2382
BT_UUID_POS (*C macro*), 2361
BT_UUID_POS_VAL (*C macro*), 2361
BT_UUID_PRESSURE (*C macro*), 2381
BT_UUID_PRESSURE_VAL (*C macro*), 2381
BT_UUID_RAINFALL (*C macro*), 2382
BT_UUID_RAINFALL_VAL (*C macro*), 2382
BT_UUID_RCSRV (*C macro*), 2362
BT_UUID_RCSRV_VAL (*C macro*), 2362
BT_UUID_RECORD_ACCESS_CONTROL_POINT (*C macro*), 2377
BT_UUID_RECORD_ACCESS_CONTROL_POINT_VAL (*C macro*), 2377
BT_UUID_RFCOMM (*C macro*), 2425
BT_UUID_RFCOMM_VAL (*C macro*), 2425
BT_UUID_RSC_FEATURE (*C macro*), 2378
BT_UUID_RSC_FEATURE_VAL (*C macro*), 2378
BT_UUID_RSC_MEASUREMENT (*C macro*), 2378
BT_UUID_RSC_MEASUREMENT_VAL (*C macro*), 2378
BT_UUID_RSCS (*C macro*), 2360
BT_UUID_RSCS_VAL (*C macro*), 2359
BT_UUID_RTUS (*C macro*), 2358
BT_UUID_RTUS_VAL (*C macro*), 2358
BT_UUID_SC_CONTROL_POINT (*C macro*), 2378
BT_UUID_SC_CONTROL_POINT_VAL (*C macro*), 2378
BT_UUID_SDP (*C macro*), 2425
BT_UUID_SDP_VAL (*C macro*), 2425
BT_UUID_SENSOR_LOCATION (*C macro*), 2379
BT_UUID_SENSOR_LOCATION_VAL (*C macro*), 2379
BT_UUID_SIZE_16 (*C macro*), 2355
BT_UUID_SIZE_32 (*C macro*), 2355
BT_UUID_SIZE_128 (*C macro*), 2355

BT_UUID_SPS (*C macro*), 2359
BT_UUID_SPS_VAL (*C macro*), 2359
BT_UUID_STR_LEN (*C macro*), 2357
BT_UUID_TBS (*C macro*), 2364
BT_UUID_TBS_CALL_CONTROL_POINT (*C macro*), 2418
BT_UUID_TBS_CALL_CONTROL_POINT_VAL (*C macro*), 2418
BT_UUID_TBS_CALL_STATE (*C macro*), 2418
BT_UUID_TBS_CALL_STATE_VAL (*C macro*), 2418
BT_UUID_TBS_FRIENDLY_NAME (*C macro*), 2419
BT_UUID_TBS_FRIENDLY_NAME_VAL (*C macro*), 2419
BT_UUID_TBS_INCOMING_CALL (*C macro*), 2419
BT_UUID_TBS_INCOMING_CALL_VAL (*C macro*), 2419
BT_UUID_TBS_INCOMING_URI (*C macro*), 2418
BT_UUID_TBS_INCOMING_URI_VAL (*C macro*), 2418
BT_UUID_TBS_LIST_CURRENT_CALLS (*C macro*), 2418
BT_UUID_TBS_LIST_CURRENT_CALLS_VAL (*C macro*), 2418
BT_UUID_TBS_OPTIONAL_OPCODES (*C macro*), 2418
BT_UUID_TBS_OPTIONAL_OPCODES_VAL (*C macro*), 2418
BT_UUID_TBS_PROVIDER_NAME (*C macro*), 2417
BT_UUID_TBS_PROVIDER_NAME_VAL (*C macro*), 2417
BT_UUID_TBS_SIGNAL_INTERVAL (*C macro*), 2417
BT_UUID_TBS_SIGNAL_INTERVAL_VAL (*C macro*), 2417
BT_UUID_TBS_SIGNAL_STRENGTH (*C macro*), 2417
BT_UUID_TBS_SIGNAL_STRENGTH_VAL (*C macro*), 2417
BT_UUID_TBS_STATUS_FLAGS (*C macro*), 2418
BT_UUID_TBS_STATUS_FLAGS_VAL (*C macro*), 2418
BT_UUID_TBS_TECHNOLOGY (*C macro*), 2417
BT_UUID_TBS_TECHNOLOGY_VAL (*C macro*), 2417
BT_UUID_TBS_TERMINATE_REASON (*C macro*), 2418
BT_UUID_TBS_TERMINATE_REASON_VAL (*C macro*), 2418
BT_UUID_TBS_UCI (*C macro*), 2417
BT_UUID_TBS_UCI_VAL (*C macro*), 2417
BT_UUID_TBS_URI_LIST (*C macro*), 2417
BT_UUID_TBS_URI_LIST_VAL (*C macro*), 2417
BT_UUID_TBS_VAL (*C macro*), 2364
BT_UUID_TCP (*C macro*), 2426
BT_UUID_TCP_VAL (*C macro*), 2426
BT_UUID_TCS_AT (*C macro*), 2426
BT_UUID_TCS_AT_VAL (*C macro*), 2426
BT_UUID_TCS_BIN (*C macro*), 2426
BT_UUID_TCS_BIN_VAL (*C macro*), 2426
BT_UUID_TDS (*C macro*), 2361
BT_UUID_TDS_VAL (*C macro*), 2361
BT_UUID_TEMPERATURE (*C macro*), 2381
BT_UUID_TEMPERATURE_VAL (*C macro*), 2381
BT_UUID_TM_TRIGGER_SETTING (*C macro*), 2367
BT_UUID_TM_TRIGGER_SETTING_VAL (*C macro*), 2367
BT_UUID_TMAS (*C macro*), 2365
BT_UUID_TMAS_VAL (*C macro*), 2365
bt_uuid_to_str (*C function*), 2428
BT_UUID_TPS (*C macro*), 2358
BT_UUID_TPS_TX_POWER_LEVEL (*C macro*), 2368
BT_UUID_TPS_TX_POWER_LEVEL_VAL (*C macro*), 2368
BT_UUID_TPS_VAL (*C macro*), 2357
BT_UUID_TRUE_WIND_DIR (*C macro*), 2381
BT_UUID_TRUE_WIND_DIR_VAL (*C macro*), 2381
BT_UUID_TRUE_WIND_SPEED (*C macro*), 2381
BT_UUID_TRUE_WIND_SPEED_VAL (*C macro*), 2381

BT_UUID_UDI (*C macro*), 2427
BT_UUID_UDI_VAL (*C macro*), 2427
BT_UUID_UDP (*C macro*), 2425
BT_UUID_UDP_VAL (*C macro*), 2425
BT_UUID_UDS (*C macro*), 2360
BT_UUID_UDS_VAL (*C macro*), 2360
BT_UUID_UPNP (*C macro*), 2426
BT_UUID_UPNP_VAL (*C macro*), 2426
BT_UUID_URI (*C macro*), 2390
BT_UUID_URI_VAL (*C macro*), 2390
BT_UUID_UV_INDEX (*C macro*), 2382
BT_UUID_UV_INDEX_VAL (*C macro*), 2382
BT_UUID_VAL_TRIGGER_SETTING (*C macro*), 2367
BT_UUID_VAL_TRIGGER_SETTING_VAL (*C macro*), 2367
BT_UUID_VALID_RANGE (*C macro*), 2367
BT_UUID_VALID_RANGE_VAL (*C macro*), 2367
BT_UUID_VCS (*C macro*), 2363
BT_UUID_VCS_CONTROL (*C macro*), 2410
BT_UUID_VCS_CONTROL_VAL (*C macro*), 2410
BT_UUID_VCS_FLAGS (*C macro*), 2410
BT_UUID_VCS_FLAGS_VAL (*C macro*), 2410
BT_UUID_VCS_STATE (*C macro*), 2410
BT_UUID_VCS_STATE_VAL (*C macro*), 2410
BT_UUID_VCS_VAL (*C macro*), 2363
BT_UUID_VOCS (*C macro*), 2363
BT_UUID_VOCS_CONTROL (*C macro*), 2411
BT_UUID_VOCS_CONTROL_VAL (*C macro*), 2411
BT_UUID_VOCS_DESCRIPTION (*C macro*), 2411
BT_UUID_VOCS_DESCRIPTION_VAL (*C macro*), 2411
BT_UUID_VOCS_LOCATION (*C macro*), 2411
BT_UUID_VOCS_LOCATION_VAL (*C macro*), 2411
BT_UUID_VOCS_STATE (*C macro*), 2411
BT_UUID_VOCS_STATE_VAL (*C macro*), 2411
BT_UUID_VOCS_VAL (*C macro*), 2363
BT_UUID_WDS (*C macro*), 2359
BT_UUID_WDS_VAL (*C macro*), 2358
BT_UUID_WIND_CHILL (*C macro*), 2382
BT_UUID_WIND_CHILL_VAL (*C macro*), 2382
BT_UUID_WSP (*C macro*), 2426
BT_UUID_WSP_VAL (*C macro*), 2426
BT_UUID_WSS (*C macro*), 2361
BT_UUID_WSS_VAL (*C macro*), 2360
BT_VCP_ERR_INVALID_COUNTER (*C macro*), 1895
BT_VCP_ERR_OP_NOT_SUPPORTED (*C macro*), 1895
bt_vcp_included (*C struct*), 1900
bt_vcp_included.aics (*C var*), 1900
bt_vcp_included.aics_cnt (*C var*), 1900
bt_vcp_included.vocs (*C var*), 1900
bt_vcp_included.vocs_cnt (*C var*), 1900
BT_VCP_STATE_MUTED (*C macro*), 1895
BT_VCP_STATE_UNMUTED (*C macro*), 1895
bt_vcp_vol_ctlr_cb (*C struct*), 1901
bt_vcp_vol_ctlr_cb_register (*C function*), 1897
bt_vcp_vol_ctlr_cb_unregister (*C function*), 1897
bt_vcp_vol_ctlr_cb.aics_cb (*C var*), 1903
bt_vcp_vol_ctlr_cb.discover (*C var*), 1902
bt_vcp_vol_ctlr_cb.flags (*C var*), 1901
bt_vcp_vol_ctlr_cb.mute (*C var*), 1902

bt_vcp_vol_ctlr_cb.state (*C var*), 1901
bt_vcp_vol_ctlr_cb.unmute (*C var*), 1902
bt_vcp_vol_ctlr_cb.vocs_cb (*C var*), 1903
bt_vcp_vol_ctlr_cb.vol_down (*C var*), 1902
bt_vcp_vol_ctlr_cb.vol_down_unmute (*C var*), 1903
bt_vcp_vol_ctlr_cb.vol_set (*C var*), 1903
bt_vcp_vol_ctlr_cb.vol_up (*C var*), 1902
bt_vcp_vol_ctlr_cb.vol_up_unmute (*C var*), 1903
bt_vcp_vol_ctlr_conn_get (*C function*), 1897
bt_vcp_vol_ctlr_discover (*C function*), 1897
bt_vcp_vol_ctlr_get_by_conn (*C function*), 1897
bt_vcp_vol_ctlr_included_get (*C function*), 1898
bt_vcp_vol_ctlr_mute (*C function*), 1899
bt_vcp_vol_ctlr_read_flags (*C function*), 1898
bt_vcp_vol_ctlr_read_state (*C function*), 1898
bt_vcp_vol_ctlr_set_vol (*C function*), 1899
bt_vcp_vol_ctlr_unmute (*C function*), 1899
bt_vcp_vol_ctlr_unmute_vol_down (*C function*), 1899
bt_vcp_vol_ctlr_unmute_vol_up (*C function*), 1899
bt_vcp_vol_ctlr_vol_down (*C function*), 1898
bt_vcp_vol_ctlr_vol_up (*C function*), 1898
BT_VCP_VOL_REND_AICS_CNT (*C macro*), 1895
bt_vcp_vol_rend_cb (*C struct*), 1900
bt_vcp_vol_rend_cb.flags (*C var*), 1901
bt_vcp_vol_rend_cb.state (*C var*), 1900
bt_vcp_vol_rend_get_flags (*C function*), 1896
bt_vcp_vol_rend_get_state (*C function*), 1896
bt_vcp_vol_rend_included_get (*C function*), 1895
bt_vcp_vol_rend_mute (*C function*), 1896
bt_vcp_vol_rend_register (*C function*), 1895
bt_vcp_vol_rend_register_param (*C struct*), 1899
bt_vcp_vol_rend_register_param.aics_param (*C var*), 1900
bt_vcp_vol_rend_register_param.cb (*C var*), 1900
bt_vcp_vol_rend_register_param.mute (*C var*), 1899
bt_vcp_vol_rend_register_param.step (*C var*), 1899
bt_vcp_vol_rend_register_param.vocs_param (*C var*), 1900
bt_vcp_vol_rend_register_param.volume (*C var*), 1900
bt_vcp_vol_rend_set_step (*C function*), 1895
bt_vcp_vol_rend_set_vol (*C function*), 1896
bt_vcp_vol_rend_unmute (*C function*), 1896
bt_vcp_vol_rend_unmute_vol_down (*C function*), 1896
bt_vcp_vol_rend_unmute_vol_up (*C function*), 1896
BT_VCP_VOL_REND_VOCS_CNT (*C macro*), 1895
bt_vcp_vol_rend_vol_down (*C function*), 1896
bt_vcp_vol_rend_vol_up (*C function*), 1896
build_conf (*runners.core.ZephyrBinaryRunner property*), 199
build_dir (*runners.core.RunnerConfig attribute*), 197
BuildConfiguration (*class in runners.core*), 195
bytecpy (*C function*), 696
byteswp (*C function*), 696

C

call() (*runners.core.ZephyrBinaryRunner method*), 199
can_add_rx_filter (*C function*), 3237
can_add_rx_filter_msgq (*C function*), 3238
can_bus_err_cnt (*C struct*), 3250
can_bus_err_cnt.rx_err_cnt (*C var*), 3250
can_bus_err_cnt.tx_err_cnt (*C var*), 3250

can_bytes_to_dlc (*C function*), 3243
can_calc_prescaler (*C function*), 3233
can_calc_timing (*C function*), 3230
can_calc_timing_data (*C function*), 3231
CAN_DEVICE_DT_DEFINE (*C macro*), 3247
CAN_DEVICE_DT_INST_DEFINE (*C macro*), 3247
can_device_state (*C struct*), 3251
can_device_state.devstate (*C var*), 3251
can_device_state.stats (*C var*), 3251
can_dlc_to_bytes (*C function*), 3243
CAN_EXT_ID_MASK (*C macro*), 3243
can_filter (*C struct*), 3250
CAN_FILTER_IDE (*C macro*), 3245
can_filter.flags (*C var*), 3250
can_filter.id (*C var*), 3250
can_filter.mask (*C var*), 3250
can_frame (*C struct*), 3249
CAN_FRAME_BRS (*C macro*), 3245
CAN_FRAME_ESI (*C macro*), 3245
CAN_FRAME_FDF (*C macro*), 3245
CAN_FRAME_IDE (*C macro*), 3244
can_frame_matches_filter (*C function*), 3243
CAN_FRAME_RTR (*C macro*), 3244
can_frame.data (*C var*), 3249
can_frame.data_32 (*C var*), 3250
can_frame.dlc (*C var*), 3249
can_frame.flags (*C var*), 3249
can_frame.id (*C var*), 3249
can_frame.timestamp (*C var*), 3249
can_get_bitrate_max (*C function*), 3229
can_get_bitrate_min (*C function*), 3229
can_get_capabilities (*C function*), 3234
can_get_core_clock (*C function*), 3228
can_get_max_bitrate (*C function*), 3229
can_get_max_filters (*C function*), 3238
can_get_min_bitrate (*C function*), 3229
can_get_mode (*C function*), 3235
can_get_state (*C function*), 3239
can_get_timing_data_max (*C function*), 3231
can_get_timing_data_min (*C function*), 3230
can_get_timing_max (*C function*), 3230
can_get_timing_min (*C function*), 3230
can_get_transceiver (*C function*), 3234
CAN_MAX_DLC (*C macro*), 3244
CAN_MAX_EXT_ID (*C macro*), 3244
CAN_MAX_STD_ID (*C macro*), 3243
CAN_MODE_3_SAMPLES (*C macro*), 3244
CAN_MODE_FD (*C macro*), 3244
CAN_MODE_LISTENONLY (*C macro*), 3244
CAN_MODE_LOOPBACK (*C macro*), 3244
CAN_MODE_MANUAL_RECOVERY (*C macro*), 3244
CAN_MODE_NORMAL (*C macro*), 3244
CAN_MODE_ONE_SHOT (*C macro*), 3244
can_mode_t (*C type*), 3248
CAN_MSGQ_DEFINE (*C macro*), 3239
can_recover (*C function*), 3239
can_remove_rx_filter (*C function*), 3238
can_rx_callback_t (*C type*), 3248

can_send (*C function*), 3236
can_set_bitrate (*C function*), 3235
can_set_bitrate_data (*C function*), 3232
can_set_mode (*C function*), 3235
can_set_state_change_callback (*C function*), 3240
can_set_timing (*C function*), 3233
can_set_timing_data (*C function*), 3232
can_start (*C function*), 3234
can_state (*C enum*), 3248
can_state_change_callback_t (*C type*), 3248
can_state.CAN_STATE_BUS_OFF (*C enumerator*), 3249
can_state.CAN_STATE_ERROR_ACTIVE (*C enumerator*), 3249
can_state.CAN_STATE_ERROR_PASSIVE (*C enumerator*), 3249
can_state.CAN_STATE_ERROR_WARNING (*C enumerator*), 3249
can_state.CAN_STATE_STOPPED (*C enumerator*), 3249
CAN_STATS_ACK_ERROR_INC (*C macro*), 3246
CAN_STATS_BIT0_ERROR_INC (*C macro*), 3246
CAN_STATS_BIT1_ERROR_INC (*C macro*), 3246
CAN_STATS_BIT_ERROR_INC (*C macro*), 3245
CAN_STATS_CRC_ERROR_INC (*C macro*), 3246
CAN_STATS_FORM_ERROR_INC (*C macro*), 3246
can_stats_get_ack_errors (*C function*), 3242
can_stats_get_bit0_errors (*C function*), 3240
can_stats_get_bit1_errors (*C function*), 3241
can_stats_get_bit_errors (*C function*), 3240
can_stats_get_crc_errors (*C function*), 3241
can_stats_get_form_errors (*C function*), 3242
can_stats_get_rx_overruns (*C function*), 3242
can_stats_get_stuff_errors (*C function*), 3241
CAN_STATS_RESET (*C macro*), 3247
CAN_STATS_RX_OVERRUN_INC (*C macro*), 3247
CAN_STATS_STUFF_ERROR_INC (*C macro*), 3246
CAN_STD_ID_MASK (*C macro*), 3243
can_stop (*C function*), 3235
can_timing (*C struct*), 3250
can_timing.phase_seg1 (*C var*), 3251
can_timing.phase_seg2 (*C var*), 3251
can_timing.prescaler (*C var*), 3251
can_timing.prop_seg (*C var*), 3251
can_timing.sjw (*C var*), 3251
can_transceiver_disable (*C function*), 3253
can_transceiver_enable (*C function*), 3252
can_tx_callback_t (*C type*), 3248
CANFD_MAX_DLC (*C macro*), 3244
CAP_ASYNC_OPS (*C macro*), 720
CAP_AUTONONCE (*C macro*), 720
CAP_INPLACE_OPS (*C macro*), 720
CAP_KEY_LOADING_API (*C macro*), 720
CAP_NO_IV_PREFIX (*C macro*), 720
CAP_OPAQUE_KEY_HNDL (*C macro*), 720
CAP_RAW_KEY (*C macro*), 720
CAP_SEPARATE_IO_BUFS (*C macro*), 720
CAP_SYNC_OPS (*C macro*), 720
capabilities() (*runners.core.ZephyrBinaryRunner class method*), 199
cbc_op_t (*C type*), 721
cbprintf (*C function*), 869
cbprintf_external (*C function*), 867
cbprintf (*C function*), 867

cbprintf_cb (*C type*), 863
cbprintf_cb_local (*C type*), 863
cbprintf_convert_cb (*C type*), 863
cbprintf_fsc_package (*C function*), 866
CBPRINTF_MUST_RUNTIME_PACKAGE (*C macro*), 862
cbprintf_package (*C function*), 864
CBPRINTF_PACKAGE_ALIGNMENT (*C macro*), 862
cbprintf_package_convert (*C function*), 865
cbprintf_package_copy (*C function*), 866
CBPRINTF_STATIC_PACKAGE (*C macro*), 862
cbvprintf (*C function*), 868
cbvprintf_external_formatter_func (*C type*), 864
cbvprintf_package (*C function*), 865
cbvprintf_tagged_args (*C function*), 869
ccm_op_t (*C type*), 721
ccm_params (*C struct*), 724
ceiling_fraction (*C macro*), 683
cfb_display_param (*C enum*), 3302
cfb_display_param.CFB_DISPLAY_COLS (*C enumerator*), 3302
cfb_display_param.CFB_DISPLAY_HEIGH (*C enumerator*), 3302
cfb_display_param.CFB_DISPLAY_PPT (*C enumerator*), 3302
cfb_display_param.CFB_DISPLAY_ROWS (*C enumerator*), 3302
cfb_display_param.CFB_DISPLAY_WIDTH (*C enumerator*), 3302
cfb_draw_line (*C function*), 3303
cfb_draw_point (*C function*), 3303
cfb_draw_rect (*C function*), 3303
cfb_draw_text (*C function*), 3303
cfb_font (*C struct*), 3305
cfb_font_caps (*C enum*), 3302
cfb_font_caps.CFB_FONT_MONO_HPACKED (*C enumerator*), 3302
cfb_font_caps.CFB_FONT_MONO_VPACKED (*C enumerator*), 3302
cfb_font_caps.CFB_FONT_MSB_FIRST (*C enumerator*), 3302
cfb_framebuffer_clear (*C function*), 3303
cfb_framebuffer_deinit (*C function*), 3305
cfb_framebuffer_finalize (*C function*), 3304
cfb_framebuffer_init (*C function*), 3305
cfb_framebuffer_invert (*C function*), 3304
cfb_framebuffer_set_font (*C function*), 3304
cfb_get_display_parameter (*C function*), 3304
cfb_get_font_size (*C function*), 3305
cfb_get_numof_fonts (*C function*), 3305
cfb_invert_area (*C function*), 3304
cfb_position (*C struct*), 3305
cfb_print (*C function*), 3302
cfb_set_kerning (*C function*), 3305
cfg (*runners.core.ZephyrBinaryRunner attribute*), 199
char2hex (*C function*), 697
charger_charge_enable (*C function*), 3264
charger_charge_enable_t (*C type*), 3259
charger_charge_type (*C enum*), 3261
charger_charge_type.CHARGER_CHARGE_TYPE_ADAPTIVE (*C enumerator*), 3262
charger_charge_type.CHARGER_CHARGE_TYPE_BYPASS (*C enumerator*), 3262
charger_charge_type.CHARGER_CHARGE_TYPE_FAST (*C enumerator*), 3262
charger_charge_type.CHARGER_CHARGE_TYPE_LONGLIFE (*C enumerator*), 3262
charger_charge_type.CHARGER_CHARGE_TYPE_NONE (*C enumerator*), 3261
charger_charge_type.CHARGER_CHARGE_TYPE_STANDARD (*C enumerator*), 3262
charger_charge_type.CHARGER_CHARGE_TYPE_TRICKLE (*C enumerator*), 3261
charger_charge_type.CHARGER_CHARGE_TYPE_UNKNOWN (*C enumerator*), 3261

charger_current_notifier (*C struct*), 3264
charger_current_notifier.current_ua (*C var*), 3264
charger_current_notifier.duration_us (*C var*), 3264
charger_current_notifier.severity (*C var*), 3264
charger_driver_api (*C struct*), 3265
charger_get_prop (*C function*), 3263
charger_get_property_t (*C type*), 3259
charger_health (*C enum*), 3262
charger_health.CHARGER_HEALTH_CALIBRATION_REQUIRED (*C enumerator*), 3262
charger_health.CHARGER_HEALTH_COLD (*C enumerator*), 3262
charger_health.CHARGER_HEALTH_COOL (*C enumerator*), 3263
charger_health.CHARGER_HEALTH_GOOD (*C enumerator*), 3262
charger_health.CHARGER_HEALTH_HOT (*C enumerator*), 3263
charger_health.CHARGER_HEALTH_NO_BATTERY (*C enumerator*), 3263
charger_health.CHARGER_HEALTH_OVERHEAT (*C enumerator*), 3262
charger_health.CHARGER_HEALTH_OVERVOLTAGE (*C enumerator*), 3262
charger_health.CHARGER_HEALTH_SAFETY_TIMER_EXPIRE (*C enumerator*), 3262
charger_health.CHARGER_HEALTH_UNKNOWN (*C enumerator*), 3262
charger_health.CHARGER_HEALTH_UNSPEC_FAILURE (*C enumerator*), 3262
charger_health.CHARGER_HEALTH_WARM (*C enumerator*), 3262
charger_health.CHARGER_HEALTH_WATCHDOG_TIMER_EXPIRE (*C enumerator*), 3262
charger_notification_severity (*C enum*), 3263
charger_notification_severity.CHARGER_SEVERITY_CRITICAL (*C enumerator*), 3263
charger_notification_severity.CHARGER_SEVERITY_PEAK (*C enumerator*), 3263
charger_notification_severity.CHARGER_SEVERITY_WARNING (*C enumerator*), 3263
charger_online (*C enum*), 3261
charger_online_notifier_t (*C type*), 3258
charger_online.CHARGER_ONLINE_FIXED (*C enumerator*), 3261
charger_online.CHARGER_ONLINE_OFFLINE (*C enumerator*), 3261
charger_online.CHARGER_ONLINE_PROGRAMMABLE (*C enumerator*), 3261
charger_prop_t (*C type*), 3258
charger_property (*C enum*), 3259
charger_property.CHARGER_PROP_CHARGE_TERM_CURRENT_UA (*C enumerator*), 3260
charger_property.CHARGER_PROP_CHARGE_TYPE (*C enumerator*), 3259
charger_property.CHARGER_PROP_COMMON_COUNT (*C enumerator*), 3260
charger_property.CHARGER_PROP_CONSTANT_CHARGE_CURRENT_UA (*C enumerator*), 3259
charger_property.CHARGER_PROP_CONSTANT_CHARGE_VOLTAGE_UV (*C enumerator*), 3260
charger_property.CHARGER_PROP_CUSTOM_BEGIN (*C enumerator*), 3260
charger_property.CHARGER_PROP_DISCHARGE_CURRENT_NOTIFICATION (*C enumerator*), 3260
charger_property.CHARGER_PROP_HEALTH (*C enumerator*), 3259
charger_property.CHARGER_PROP_INPUT_CURRENT_NOTIFICATION (*C enumerator*), 3260
charger_property.CHARGER_PROP_INPUT_REGULATION_CURRENT_UA (*C enumerator*), 3260
charger_property.CHARGER_PROP_INPUT_REGULATION_VOLTAGE_UV (*C enumerator*), 3260
charger_property.CHARGER_PROP_MAX (*C enumerator*), 3261
charger_property.CHARGER_PROP_ONLINE (*C enumerator*), 3259
charger_property.CHARGER_PROP_ONLINE_NOTIFICATION (*C enumerator*), 3260
charger_property.CHARGER_PROP_PRECHARGE_CURRENT_UA (*C enumerator*), 3260
charger_property.CHARGER_PROP_PRESENT (*C enumerator*), 3259
charger_property.CHARGER_PROP_STATUS (*C enumerator*), 3259
charger_property.CHARGER_PROP_STATUS_NOTIFICATION (*C enumerator*), 3260
charger_property.CHARGER_PROP_SYSTEM_VOLTAGE_NOTIFICATION_UV (*C enumerator*), 3260
charger_propval (*C union*), 3264
charger_propval.charge_term_current_ua (*C var*), 3265
charger_propval.charge_type (*C var*), 3264
charger_propval.const_charge_current_ua (*C var*), 3265
charger_propval.const_charge_voltage_uv (*C var*), 3265
charger_propval.discharge_current_notification (*C var*), 3265
charger_propval.health (*C var*), 3264

charger_propval.input_current_notification (*C var*), 3265
charger_propval.input_current_regulation_current_ua (*C var*), 3265
charger_propval.input_voltage_regulation_voltage_uv (*C var*), 3265
charger_propval.online (*C var*), 3264
charger_propval.online_notification (*C var*), 3265
charger_propval.precharge_current_ua (*C var*), 3265
charger_propval.present (*C var*), 3264
charger_propval.status (*C var*), 3264
charger_propval.status_notification (*C var*), 3265
charger_propval.system_voltage_notification (*C var*), 3265
charger_set_prop (*C function*), 3263
charger_set_property_t (*C type*), 3259
charger_status (*C enum*), 3261
charger_status_notifier_t (*C type*), 3258
charger_status.CHARGER_STATUS_CHARGING (*C enumerator*), 3261
charger_status.CHARGER_STATUS_DISCHARGING (*C enumerator*), 3261
charger_status.CHARGER_STATUS_FULL (*C enumerator*), 3261
charger_status.CHARGER_STATUS_NOT_CHARGING (*C enumerator*), 3261
charger_status.CHARGER_STATUS_UNKNOWN (*C enumerator*), 3261
check_call() (*runners.core.ZephyrBinaryRunner method*), 199
check_output() (*runners.core.ZephyrBinaryRunner method*), 199
cipher_aead_pkt (*C struct*), 726
cipher_aead_pkt.ad (*C var*), 726
cipher_aead_pkt.ad_len (*C var*), 726
cipher_aead_pkt.tag (*C var*), 726
cipher_algo (*C enum*), 721
cipher_algo.CRYPTO_CIPHER_ALGO_AES (*C enumerator*), 721
cipher_begin_session (*C function*), 722
cipher_block_op (*C function*), 723
cipher_callback_set (*C function*), 723
cipher_cbc_op (*C function*), 723
cipher_ccm_op (*C function*), 724
cipher_completion_cb (*C type*), 721
cipher_ctr_op (*C function*), 723
cipher_ctx (*C struct*), 724
cipher_ctx.app_sessn_state (*C var*), 725
cipher_ctx.device (*C var*), 725
cipher_ctx.drv_sessn_state (*C var*), 725
cipher_ctx.flags (*C var*), 725
cipher_ctx.key (*C var*), 725
cipher_ctx.keylen (*C var*), 725
cipher_ctx.mode_params (*C var*), 725
cipher_ctx.ops (*C var*), 725
cipher_free_session (*C function*), 722
cipher_gcm_op (*C function*), 724
cipher_mode (*C enum*), 722
cipher_mode.CRYPTO_CIPHER_MODE_CBC (*C enumerator*), 722
cipher_mode.CRYPTO_CIPHER_MODE_CCM (*C enumerator*), 722
cipher_mode.CRYPTO_CIPHER_MODE_CTR (*C enumerator*), 722
cipher_mode.CRYPTO_CIPHER_MODE_ECB (*C enumerator*), 722
cipher_mode.CRYPTO_CIPHER_MODE_GCM (*C enumerator*), 722
cipher_op (*C enum*), 721
cipher_op.CRYPTO_CIPHER_OP_DECRYPT (*C enumerator*), 722
cipher_op.CRYPTO_CIPHER_OP_ENCRYPT (*C enumerator*), 722
cipher_ops (*C struct*), 724
cipher_pkt (*C struct*), 725
cipher_pkt.ctx (*C var*), 726
cipher_pkt.in_buf (*C var*), 726

cipher_pkt.in_len (*C var*), 726
cipher_pkt.out_buf (*C var*), 726
cipher_pkt.out_buf_max (*C var*), 726
cipher_pkt.out_len (*C var*), 726
CLAMP (*C macro*), 684
clock_control (*C type*), 3220
clock_control_async_on (*C function*), 3221
clock_control_async_on_fn (*C type*), 3220
clock_control_cb_t (*C type*), 3220
clock_control_configure (*C function*), 3223
clock_control_configure_fn (*C type*), 3221
clock_control_driver_api (*C struct*), 3223
clock_control_get (*C type*), 3220
clock_control_get_rate (*C function*), 3222
clock_control_get_status (*C function*), 3222
clock_control_get_status_fn (*C type*), 3220
clock_control_off (*C function*), 3221
clock_control_on (*C function*), 3221
clock_control_set (*C type*), 3220
clock_control_set_rate (*C function*), 3222
clock_control_status (*C enum*), 3221
clock_control_status.CLOCK_CONTROL_STATUS_OFF (*C enumerator*), 3221
clock_control_status.CLOCK_CONTROL_STATUS_ON (*C enumerator*), 3221
clock_control_status.CLOCK_CONTROL_STATUS_STARTING (*C enumerator*), 3221
clock_control_status.CLOCK_CONTROL_STATUS_UNKNOWN (*C enumerator*), 3221
CLOCK_CONTROL_SUBSYS_ALL (*C macro*), 3220
clock_control_subsys_rate_t (*C type*), 3220
clock_control_subsys_t (*C type*), 3220
close (*C function*), 2488
close() (*twister_harness.DeviceAdapter method*), 268
CMSG_DATA (*C macro*), 2509
CMSG_FIRSTHDR (*C macro*), 2509
CMSG_LEN (*C macro*), 2509
CMSG_NXTHDR (*C macro*), 2509
CMSG_SPACE (*C macro*), 2509
cmsghdr (*C struct*), 2528
cmsghdr.cmsg_data (*C var*), 2528
cmsghdr.cmsg_len (*C var*), 2528
cmsghdr.cmsg_level (*C var*), 2528
cmsghdr.cmsg_type (*C var*), 2528
coap_ack_init (*C function*), 2758
coap_append_block1_option (*C function*), 2762
coap_append_block2_option (*C function*), 2762
coap_append_descriptive_block_option (*C function*), 2761
coap_append_option_int (*C function*), 2759
coap_append_size1_option (*C function*), 2762
coap_append_size2_option (*C function*), 2763
coap_block_context (*C struct*), 2773
coap_block_context.block_size (*C var*), 2773
coap_block_context.current (*C var*), 2773
coap_block_context.total_size (*C var*), 2773
coap_block_has_more (*C function*), 2762
coap_block_size (*C enum*), 2755
coap_block_size_to_bytes (*C function*), 2761
coap_block_size.COAP_BLOCK_16 (*C enumerator*), 2755
coap_block_size.COAP_BLOCK_32 (*C enumerator*), 2755
coap_block_size.COAP_BLOCK_64 (*C enumerator*), 2755
coap_block_size.COAP_BLOCK_128 (*C enumerator*), 2755

coap_block_size.COAP_BLOCK_256 (*C enumerator*), 2755
coap_block_size.COAP_BLOCK_512 (*C enumerator*), 2755
coap_block_size.COAP_BLOCK_1024 (*C enumerator*), 2755
coap_block_transfer_init (*C function*), 2761
coap_bytes_to_block_size (*C function*), 2761
coap_client_cancel_requests (*C function*), 2776
coap_client_init (*C function*), 2775
coap_client_option (*C struct*), 2777
coap_client_option.code (*C var*), 2777
coap_client_option.len (*C var*), 2777
coap_client_option.value (*C var*), 2777
coap_client_req (*C function*), 2775
coap_client_request (*C struct*), 2776
coap_client_request.cb (*C var*), 2777
coap_client_request.confirmable (*C var*), 2776
coap_client_request.fmt (*C var*), 2776
coap_client_request.len (*C var*), 2776
coap_client_request.method (*C var*), 2776
coap_client_request.num_options (*C var*), 2777
coap_client_request.options (*C var*), 2777
coap_client_request.path (*C var*), 2776
coap_client_request.payload (*C var*), 2776
coap_client_request.user_data (*C var*), 2777
coap_client_response_cb_t (*C type*), 2775
coap_content_format (*C enum*), 2754
coap_content_format.COAP_CONTENT_FORMAT_APP_CBOR (*C enumerator*), 2755
coap_content_format.COAP_CONTENT_FORMAT_APP_EXI (*C enumerator*), 2755
coap_content_format.COAP_CONTENT_FORMAT_APP_JSON (*C enumerator*), 2755
coap_content_format.COAP_CONTENT_FORMAT_APP_JSON_PATCH_JSON (*C enumerator*), 2755
coap_content_format.COAP_CONTENT_FORMAT_APP_LINK_FORMAT (*C enumerator*), 2754
coap_content_format.COAP_CONTENT_FORMAT_APP_MERGE_PATCH_JSON (*C enumerator*), 2755
coap_content_format.COAP_CONTENT_FORMAT_APP_OCTET_STREAM (*C enumerator*), 2754
coap_content_format.COAP_CONTENT_FORMAT_APP_XML (*C enumerator*), 2754
coap_content_format.COAP_CONTENT_FORMAT_TEXT_PLAIN (*C enumerator*), 2754
coap_core_metadata (*C struct*), 2773
coap_core_metadata.attributes (*C var*), 2773
coap_core_metadata.user_data (*C var*), 2773
coap_find_observer (*C function*), 2765
coap_find_observer_by_addr (*C function*), 2765
coap_find_observer_by_token (*C function*), 2765
coap_find_options (*C function*), 2758
coap_get_block1_option (*C function*), 2763
coap_get_block2_option (*C function*), 2763
coap_get_option_int (*C function*), 2763
coap_get_transmission_parameters (*C function*), 2769
coap_handle_request (*C function*), 2760
coap_handle_request_len (*C function*), 2760
coap_has_descriptive_block_option (*C function*), 2762
coap_header_get_code (*C function*), 2756
coap_header_get_id (*C function*), 2756
coap_header_get_token (*C function*), 2756
coap_header_get_type (*C function*), 2756
coap_header_get_version (*C function*), 2756
coap_header_set_code (*C function*), 2756
COAP_MAKE_RESPONSE_CODE (*C macro*), 2749
coap_method (*C enum*), 2751
coap_method_t (*C type*), 2750
coap_method.COAP_METHOD_DELETE (*C enumerator*), 2752

coap_method.COAP_METHOD_FETCH (*C enumerator*), 2752
coap_method.COAP_METHOD_GET (*C enumerator*), 2751
coap_method.COAP_METHOD_IPATCH (*C enumerator*), 2752
coap_method.COAP_METHOD_PATCH (*C enumerator*), 2752
coap_method.COAP_METHOD_POST (*C enumerator*), 2752
coap_method.COAP_METHOD_PUT (*C enumerator*), 2752
coap_msgtype (*C enum*), 2752
coap_msgtype.COAP_TYPE_ACK (*C enumerator*), 2752
coap_msgtype.COAP_TYPE_CON (*C enumerator*), 2752
coap_msgtype.COAP_TYPE_NON_CON (*C enumerator*), 2752
coap_msgtype.COAP_TYPE_RESET (*C enumerator*), 2752
coap_next_block (*C function*), 2764
coap_next_block_for_option (*C function*), 2764
coap_next_id (*C function*), 2758
coap_next_token (*C function*), 2758
coap_notify_t (*C type*), 2750
coap_observer (*C struct*), 2770
coap_observer_init (*C function*), 2764
coap_observer_next_unused (*C function*), 2766
coap_observer.addr (*C var*), 2770
coap_observer.list (*C var*), 2770
coap_observer.tkl (*C var*), 2770
coap_observer.token (*C var*), 2770
coap_option (*C struct*), 2771
coap_option_num (*C enum*), 2750
coap_option_num.COAP_OPTION_ACCEPT (*C enumerator*), 2751
coap_option_num.COAP_OPTION_BLOCK1 (*C enumerator*), 2751
coap_option_num.COAP_OPTION_BLOCK2 (*C enumerator*), 2751
coap_option_num.COAP_OPTION_CONTENT_FORMAT (*C enumerator*), 2751
coap_option_num.COAP_OPTION_ECHO (*C enumerator*), 2751
coap_option_num.COAP_OPTION_ETAG (*C enumerator*), 2750
coap_option_num.COAP_OPTION_IF_MATCH (*C enumerator*), 2750
coap_option_num.COAP_OPTION_IF_NONE_MATCH (*C enumerator*), 2750
coap_option_num.COAP_OPTION_LOCATION_PATH (*C enumerator*), 2750
coap_option_num.COAP_OPTION_LOCATION_QUERY (*C enumerator*), 2751
coap_option_num.COAP_OPTION_MAX_AGE (*C enumerator*), 2751
coap_option_num.COAP_OPTION_OBSERVE (*C enumerator*), 2750
coap_option_num.COAP_OPTION_PROXY_SCHEME (*C enumerator*), 2751
coap_option_num.COAP_OPTION_PROXY_URI (*C enumerator*), 2751
coap_option_num.COAP_OPTION_REQUEST_TAG (*C enumerator*), 2751
coap_option_num.COAP_OPTION_SIZE1 (*C enumerator*), 2751
coap_option_num.COAP_OPTION_SIZE2 (*C enumerator*), 2751
coap_option_num.COAP_OPTION_URI_HOST (*C enumerator*), 2750
coap_option_num.COAP_OPTION_URI_PATH (*C enumerator*), 2750
coap_option_num.COAP_OPTION_URI_PORT (*C enumerator*), 2750
coap_option_num.COAP_OPTION_URI_QUERY (*C enumerator*), 2751
coap_option_value_to_int (*C function*), 2759
coap_option.delta (*C var*), 2771
coap_option.len (*C var*), 2771
coap_option.value (*C var*), 2771
coap_packet (*C struct*), 2770
coap_packet_append_option (*C function*), 2759
coap_packet_append_payload (*C function*), 2760
coap_packet_append_payload_marker (*C function*), 2759
coap_packet_get_payload (*C function*), 2756
coap_packet_init (*C function*), 2757
coap_packet_is_request (*C function*), 2760
coap_packet_parse (*C function*), 2757

coap_packet_remove_option (*C function*), 2759
coap_packet_set_path (*C function*), 2757
coap_packet.data (*C var*), 2771
coap_packet.delta (*C var*), 2771
coap_packet.hdr_len (*C var*), 2771
coap_packet.max_len (*C var*), 2771
coap_packet.offset (*C var*), 2771
coap_packet.opt_len (*C var*), 2771
coap_pending (*C struct*), 2772
coap_pending_clear (*C function*), 2768
coap_pending_cycle (*C function*), 2767
coap_pending_init (*C function*), 2766
coap_pending_next_to_expire (*C function*), 2767
coap_pending_next_unused (*C function*), 2766
coap_pending_received (*C function*), 2767
coap_pending.addr (*C var*), 2772
coap_pending.data (*C var*), 2772
coap_pending.id (*C var*), 2772
coap_pending.len (*C var*), 2772
coap_pending.params (*C var*), 2772
coap_pending.retries (*C var*), 2772
coap_pendings_clear (*C function*), 2768
coap_pendings_count (*C function*), 2768
coap_pending.t0 (*C var*), 2772
coap_pending.timeout (*C var*), 2772
coap_register_observer (*C function*), 2764
coap_remove_descriptive_block_option (*C function*), 2762
coap_remove_observer (*C function*), 2764
coap_replies_clear (*C function*), 2768
coap_reply (*C struct*), 2772
coap_reply_clear (*C function*), 2768
coap_reply_init (*C function*), 2766
coap_reply_next_unused (*C function*), 2767
coap_reply_t (*C type*), 2750
coap_reply.age (*C var*), 2772
coap_reply.id (*C var*), 2772
coap_reply.reply (*C var*), 2772
coap_reply.tkl (*C var*), 2773
coap_reply.token (*C var*), 2773
coap_reply.user_data (*C var*), 2772
coap_request_is_observe (*C function*), 2768
coap_resource (*C struct*), 2769
COAP_RESOURCE_DEFINE (*C macro*), 2782
COAP_RESOURCE_FOREACH (*C macro*), 2784
coap_resource_notify (*C function*), 2768
coap_resource_parse_observe (*C function*), 2786
coap_resource_remove_observer_by_addr (*C function*), 2786
coap_resource_remove_observer_by_token (*C function*), 2787
coap_resource_send (*C function*), 2785
coap_resource.age (*C var*), 2770
coap_resource.get (*C var*), 2770
coap_resource.notify (*C var*), 2770
coap_resource.observers (*C var*), 2770
coap_resource.path (*C var*), 2770
coap_resource.user_data (*C var*), 2770
coap_response_code (*C enum*), 2752
coap_response_code.COAP_RESPONSE_CODE_BAD_GATEWAY (*C enumerator*), 2754
coap_response_code.COAP_RESPONSE_CODE_BAD_OPTION (*C enumerator*), 2753

coap_response_code.COAP_RESPONSE_CODE_BAD_REQUEST (*C enumerator*), 2753
coap_response_code.COAP_RESPONSE_CODE_CHANGED (*C enumerator*), 2753
coap_response_code.COAP_RESPONSE_CODE_CONFLICT (*C enumerator*), 2753
coap_response_code.COAP_RESPONSE_CODE_CONTENT (*C enumerator*), 2753
coap_response_code.COAP_RESPONSE_CODE_CONTINUE (*C enumerator*), 2753
coap_response_code.COAP_RESPONSE_CODE_CREATED (*C enumerator*), 2753
coap_response_code.COAP_RESPONSE_CODE_DELETED (*C enumerator*), 2753
coap_response_code.COAP_RESPONSE_CODE_FORBIDDEN (*C enumerator*), 2753
coap_response_code.COAP_RESPONSE_CODE_GATEWAY_TIMEOUT (*C enumerator*), 2754
coap_response_code.COAP_RESPONSE_CODE_INCOMPLETE (*C enumerator*), 2753
coap_response_code.COAP_RESPONSE_CODE_INTERNAL_ERROR (*C enumerator*), 2754
coap_response_code.COAP_RESPONSE_CODE_NOT_ACCEPTABLE (*C enumerator*), 2753
coap_response_code.COAP_RESPONSE_CODE_NOT_ALLOWED (*C enumerator*), 2753
coap_response_code.COAP_RESPONSE_CODE_NOT_FOUND (*C enumerator*), 2753
coap_response_code.COAP_RESPONSE_CODE_NOT_IMPLEMENTED (*C enumerator*), 2754
coap_response_code.COAP_RESPONSE_CODE_OK (*C enumerator*), 2752
coap_response_code.COAP_RESPONSE_CODE_PRECONDITION_FAILED (*C enumerator*), 2753
coap_response_code.COAP_RESPONSE_CODE_PROXYING_NOT_SUPPORTED (*C enumerator*), 2754
coap_response_code.COAP_RESPONSE_CODE_REQUEST_TOO_LARGE (*C enumerator*), 2754
coap_response_code.COAP_RESPONSE_CODE_SERVICE_UNAVAILABLE (*C enumerator*), 2754
coap_response_code.COAP_RESPONSE_CODE_TOO_MANY_REQUESTS (*C enumerator*), 2754
coap_response_code.COAP_RESPONSE_CODE_UNAUTHORIZED (*C enumerator*), 2753
coap_response_code.COAP_RESPONSE_CODE_UNPROCESSABLE_ENTITY (*C enumerator*), 2754
coap_response_code.COAP_RESPONSE_CODE_UNSUPPORTED_CONTENT_FORMAT (*C enumerator*), 2754
coap_response_code.COAP_RESPONSE_CODE_VALID (*C enumerator*), 2753
coap_response_received (*C function*), 2767
COAP_SERVICE_AUTOSTART (*C macro*), 2782
COAP_SERVICE_COUNT (*C macro*), 2783
COAP_SERVICE_DEFINE (*C macro*), 2783
COAP_SERVICE_FOREACH (*C macro*), 2783
COAP_SERVICE_FOREACH_RESOURCE (*C macro*), 2784
COAP_SERVICE_HAS_RESOURCE (*C macro*), 2783
coap_service_is_running (*C function*), 2785
COAP_SERVICE_RESOURCE_COUNT (*C macro*), 2783
coap_service_send (*C function*), 2785
coap_service_start (*C function*), 2784
coap_service_stop (*C function*), 2784
coap_set_transmission_parameters (*C function*), 2769
coap_transmission_parameters (*C struct*), 2771
coap_transmission_parameters.ack_timeout (*C var*), 2771
coap_transmission_parameters.coap_backoff_percent (*C var*), 2771
coap_transmission_parameters.max_retransmission (*C var*), 2771
coap_update_from_block (*C function*), 2763
coap_uri_path_match (*C function*), 2757
coap_well_known_core_get (*C function*), 2769
coap_well_known_core_get_len (*C function*), 2769
COAP_WELL_KNOWN_CORE_PATH (*C macro*), 2749
CONCAT (*C macro*), 682
COND_CODE_0 (*C macro*), 688
COND_CODE_1 (*C macro*), 687
CONFIG_BT_MESH_BLOB_CHUNK_COUNT_MAX (*C macro*), 2196
CONFIG_BT_MESH_DFD_SRV_SLOT_MAX_SIZE (*C macro*), 2215
CONFIG_BT_MESH_DFD_SRV_SLOT_SPACE (*C macro*), 2215
CONFIG_BT_MESH_DFD_SRV_TARGETS_MAX (*C macro*), 2215
CONFIG_BT_MESH_DFU_FWID_MAXLEN (*C macro*), 2225
CONFIG_BT_MESH_DFU_METADATA_MAXLEN (*C macro*), 2225
CONFIG_BT_MESH_DFU_SLOT_CNT (*C macro*), 2225
CONFIG_BT_MESH_DFU_URI_MAXLEN (*C macro*), 2225

conn_mgr_all_if_connect (*C function*), 2982
conn_mgr_all_if_disconnect (*C function*), 2982
conn_mgr_all_if_down (*C function*), 2982
conn_mgr_all_if_up (*C function*), 2982
CONN_MGR_BIND_CONN (*C macro*), 2991
CONN_MGR_BIND_CONN_INST (*C macro*), 2991
conn_mgr_binding_get_flag (*C function*), 2992
conn_mgr_binding_lock (*C function*), 2991
conn_mgr_binding_set_flag (*C function*), 2992
conn_mgr_binding_unlock (*C function*), 2992
conn_mgr_conn_api (*C struct*), 2992
conn_mgr_conn_api.connect (*C var*), 2993
conn_mgr_conn_api.disconnect (*C var*), 2993
conn_mgr_conn_api.get_opt (*C var*), 2993
conn_mgr_conn_api.init (*C var*), 2993
conn_mgr_conn_api.set_opt (*C var*), 2993
conn_mgr_conn_binding (*C struct*), 2994
conn_mgr_conn_binding.ctx (*C var*), 2994
conn_mgr_conn_binding.flags (*C var*), 2994
conn_mgr_conn_binding.iface (*C var*), 2994
conn_mgr_conn_binding.impl (*C var*), 2994
conn_mgr_conn_binding.timeout (*C var*), 2994
CONN_MGR_CONN_DECLARE_PUBLIC (*C macro*), 2991
CONN_MGR_CONN_DEFINE (*C macro*), 2991
conn_mgr_conn_impl (*C struct*), 2994
conn_mgr_conn_impl.api (*C var*), 2994
conn_mgr_if_connect (*C function*), 2979
conn_mgr_if_disconnect (*C function*), 2979
conn_mgr_if_flag (*C enum*), 2978
conn_mgr_if_flag.CONN_MGR_IF_NO_AUTO_CONNECT (*C enumerator*), 2978
conn_mgr_if_flag.CONN_MGR_IF_NO_AUTO_DOWN (*C enumerator*), 2978
conn_mgr_if_flag.CONN_MGR_IF_PERSISTENT (*C enumerator*), 2978
conn_mgr_if_get_binding (*C function*), 2991
conn_mgr_if_get_flag (*C function*), 2981
conn_mgr_if_get_opt (*C function*), 2980
conn_mgr_if_get_timeout (*C function*), 2981
conn_mgr_if_is_bound (*C function*), 2979
CONN_MGR_IF_NO_TIMEOUT (*C macro*), 2978
conn_mgr_if_set_flag (*C function*), 2981
conn_mgr_if_set_opt (*C function*), 2979
conn_mgr_if_set_timeout (*C function*), 2981
conn_mgr_ignore_iface (*C function*), 2973
conn_mgr_ignore_l2 (*C function*), 2974
conn_mgr_is_iface_ignored (*C function*), 2973
conn_mgr_mon_resend_status (*C function*), 2973
conn_mgr_watch_iface (*C function*), 2973
conn_mgr_watch_l2 (*C function*), 2974
connect (*C function*), 2488
connect() (*twister_harness.DeviceAdapter method*), 268
console_getchar (*C function*), 713
console_getline (*C function*), 714
console_getline_init (*C function*), 714
console_init (*C function*), 713
console_putchar (*C function*), 714
console_read (*C function*), 713
console_write (*C function*), 713
CONTAINER_OF (*C macro*), 681
CONTAINER_OF_VALIDATE (*C macro*), 681

coredump (*C function*), 736
coredump_buffer_output (*C function*), 736
coredump_cmd (*C function*), 736
coredump_cmd_copy_arg (*C struct*), 737
coredump_cmd_copy_arg.buffer (*C var*), 737
coredump_cmd_copy_arg.length (*C var*), 737
coredump_cmd_copy_arg.offset (*C var*), 737
coredump_cmd_id (*C enum*), 735
coredump_cmd_id.COREDUMP_CMD_CLEAR_ERROR (*C enumerator*), 735
coredump_cmd_id.COREDUMP_CMD_COPY_STORED_DUMP (*C enumerator*), 735
coredump_cmd_id.COREDUMP_CMD_ERASE_STORED_DUMP (*C enumerator*), 735
coredump_cmd_id.COREDUMP_CMD_INVALIDATE_STORED_DUMP (*C enumerator*), 735
coredump_cmd_id.COREDUMP_CMD_MAX (*C enumerator*), 735
coredump_cmd_id.COREDUMP_CMD_VERIFY_STORED_DUMP (*C enumerator*), 735
coredump_device_register_callback (*C function*), 3266
coredump_device_register_memory (*C function*), 3266
coredump_device_unregister_memory (*C function*), 3266
coredump_dump_callback_t (*C type*), 3266
coredump_mem_region_node (*C struct*), 3267
coredump_mem_region_node.node (*C var*), 3267
coredump_mem_region_node.size (*C var*), 3267
coredump_mem_region_node.start (*C var*), 3267
coredump_memory_dump (*C function*), 736
coredump_query (*C function*), 736
coredump_query_id (*C enum*), 734
coredump_query_id.COREDUMP_QUERY_GET_ERROR (*C enumerator*), 734
coredump_query_id.COREDUMP_QUERY_GET_STORED_DUMP_SIZE (*C enumerator*), 734
coredump_query_id.COREDUMP_QUERY_HAS_STORED_DUMP (*C enumerator*), 734
coredump_query_id.COREDUMP_QUERY_MAX (*C enumerator*), 735
counter_alarm_callback_t (*C type*), 3269
counter_alarm_cfg (*C struct*), 3274
COUNTER_ALARM_CFG_ABSOLUTE (*C macro*), 3268
COUNTER_ALARM_CFG_EXPIRE_WHEN_LATE (*C macro*), 3268
counter_alarm_cfg.callback (*C var*), 3274
counter_alarm_cfg.flags (*C var*), 3275
counter_alarm_cfg.ticks (*C var*), 3274
counter_alarm_cfg.user_data (*C var*), 3275
counter_api_cancel_alarm (*C type*), 3269
counter_api_get_freq (*C type*), 3269
counter_api_get_guard_period (*C type*), 3269
counter_api_get_pending_int (*C type*), 3269
counter_api_get_top_value (*C type*), 3269
counter_api_get_value (*C type*), 3269
counter_api_get_value_64 (*C type*), 3269
counter_api_set_alarm (*C type*), 3269
counter_api_set_guard_period (*C type*), 3269
counter_api_set_top_value (*C type*), 3269
counter_api_start (*C type*), 3269
counter_api_stop (*C type*), 3269
counter_cancel_channel_alarm (*C function*), 3272
counter_config_info (*C struct*), 3275
COUNTER_CONFIG_INFO_COUNT_UP (*C macro*), 3268
counter_config_info.channels (*C var*), 3275
counter_config_info.flags (*C var*), 3275
counter_config_info.freq (*C var*), 3275
counter_config_info.max_top_value (*C var*), 3275
counter_driver_api (*C struct*), 3276
counter_get_frequency (*C function*), 3270

counter_get_guard_period (*C function*), 3274
counter_get_max_top_value (*C function*), 3270
counter_get_num_of_channels (*C function*), 3270
counter_get_pending_int (*C function*), 3272
counter_get_top_value (*C function*), 3273
counter_get_value (*C function*), 3271
counter_get_value_64 (*C function*), 3271
COUNTER_GUARD_PERIOD_LATE_TO_SET (*C macro*), 3268
counter_is_counting_up (*C function*), 3270
counter_set_channel_alarm (*C function*), 3271
counter_set_guard_period (*C function*), 3273
counter_set_top_value (*C function*), 3272
counter_start (*C function*), 3270
counter_stop (*C function*), 3271
counter_ticks_to_us (*C function*), 3270
counter_top_callback_t (*C type*), 3269
counter_top_cfg (*C struct*), 3275
COUNTER_TOP_CFG_DONT_RESET (*C macro*), 3268
COUNTER_TOP_CFG_RESET_WHEN_LATE (*C macro*), 3268
counter_top_cfg.callback (*C var*), 3275
counter_top_cfg.flags (*C var*), 3275
counter_top_cfg.ticks (*C var*), 3275
counter_top_cfg.user_data (*C var*), 3275
counter_us_to_ticks (*C function*), 3270
CPU cluster, **3936**
CPU core, **3936**
crc4 (*C function*), 1293
crc4_ti (*C function*), 1293
crc7_be (*C function*), 1292
crc8 (*C function*), 1290
crc8_ccitt (*C function*), 1292
crc16 (*C function*), 1289
crc16_ansi (*C function*), 1291
crc16_ccitt (*C function*), 1290
crc16_itu_t (*C function*), 1291
crc16_reflect (*C function*), 1289
crc24_pgp (*C function*), 1293
crc24_pgp_update (*C function*), 1293
crc32_c (*C function*), 1292
crc32_ieee (*C function*), 1291
crc32_ieee_update (*C function*), 1292
crc_by_type (*C function*), 1294
crc_type (*C enum*), 1288
crc_type.CRC4 (*C enumerator*), 1288
crc_type.CRC4_TI (*C enumerator*), 1288
crc_type.CRC7_BE (*C enumerator*), 1288
crc_type.CRC8 (*C enumerator*), 1288
crc_type.CRC8_CCITT (*C enumerator*), 1288
crc_type.CRC16 (*C enumerator*), 1288
crc_type.CRC16_ANSI (*C enumerator*), 1288
crc_type.CRC16_CCITT (*C enumerator*), 1288
crc_type.CRC16_ITU_T (*C enumerator*), 1288
crc_type.CRC24_PGP (*C enumerator*), 1288
crc_type.CRC32_C (*C enumerator*), 1288
crc_type.CRC32_IEEE (*C enumerator*), 1288
create() (*runners.core.ZephyrBinaryRunner class method*), 199
crypto_driver_api (*C struct*), 721
crypto_query_hwcaps (*C function*), 721

ctr_op_t (*C type*), 721
ctr_params (*C struct*), 724

D

dac_channel_cfg (*C struct*), 3277
dac_channel_cfg.buffered (*C var*), 3277
dac_channel_cfg.channel_id (*C var*), 3277
dac_channel_cfg.resolution (*C var*), 3277
dac_channel_setup (*C function*), 3276
dac_write_value (*C function*), 3277
dai_clock_inversion (*C enum*), 3202
dai_clock_inversion.DAI_INVERSION_IB_IF (*C enumerator*), 3203
dai_clock_inversion.DAI_INVERSION_IB_NF (*C enumerator*), 3203
dai_clock_inversion.DAI_INVERSION_NB_IF (*C enumerator*), 3203
dai_clock_inversion.DAI_INVERSION_NB_NF (*C enumerator*), 3203
dai_clock_provider (*C enum*), 3202
dai_clock_provider.DAI_CBC_CFC (*C enumerator*), 3202
dai_clock_provider.DAI_CBC_CFP (*C enumerator*), 3202
dai_clock_provider.DAI_CBP_CFC (*C enumerator*), 3202
dai_clock_provider.DAI_CBP_CFP (*C enumerator*), 3202
dai_config (*C struct*), 3209
dai_config_get (*C function*), 3207
dai_config_set (*C function*), 3207
dai_config.block_size (*C var*), 3210
dai_config.channels (*C var*), 3210
dai_config.dai_index (*C var*), 3210
dai_config.format (*C var*), 3210
dai_config.link_config (*C var*), 3210
dai_config.options (*C var*), 3210
dai_config.rate (*C var*), 3210
dai_config.type (*C var*), 3210
dai_config.word_size (*C var*), 3210
dai_dir (*C enum*), 3204
dai_dir.DAI_DIR_BOTH (*C enumerator*), 3204
dai_dir.DAI_DIR_RX (*C enumerator*), 3204
dai_dir.DAI_DIR_TX (*C enumerator*), 3204
DAI_FORMAT_CLOCK_INVERSION_MASK (*C macro*), 3201
DAI_FORMAT_CLOCK_PROVIDER_MASK (*C macro*), 3201
DAI_FORMAT_PROTOCOL_MASK (*C macro*), 3201
dai_get_properties (*C function*), 3207
dai_probe (*C function*), 3206
dai_properties (*C struct*), 3209
dai_properties.dma_hs_id (*C var*), 3209
dai_properties.fifo_address (*C var*), 3209
dai_properties.fifo_depth (*C var*), 3209
dai_properties.reg_init_delay (*C var*), 3209
dai_properties.stream_id (*C var*), 3209
dai_protocol (*C enum*), 3202
dai_protocol.DAI_PROTO_DSP_A (*C enumerator*), 3202
dai_protocol.DAI_PROTO_DSP_B (*C enumerator*), 3202
dai_protocol.DAI_PROTO_I2S (*C enumerator*), 3202
dai_protocol.DAI_PROTO_LEFT_J (*C enumerator*), 3202
dai_protocol.DAI_PROTO_PDM (*C enumerator*), 3202
dai_protocol.DAI_PROTO_RIGHT_J (*C enumerator*), 3202
dai_remove (*C function*), 3206
dai_state (*C enum*), 3204
dai_state.DAI_STATE_ERROR (*C enumerator*), 3205
dai_state.DAI_STATE_NOT_READY (*C enumerator*), 3204

dai_state.DAI_STATE_PAUSED (*C enumerator*), 3205
dai_state.DAI_STATE_PRE_RUNNING (*C enumerator*), 3205
dai_state.DAI_STATE_READY (*C enumerator*), 3204
dai_state.DAI_STATE_RUNNING (*C enumerator*), 3205
dai_state.DAI_STATE_STOPPING (*C enumerator*), 3205
dai_trigger (*C function*), 3208
dai_trigger_cmd (*C enum*), 3205
dai_trigger_cmd.DAI_TRIGGER_COPY (*C enumerator*), 3206
dai_trigger_cmd.DAI_TRIGGER_DRAIN (*C enumerator*), 3206
dai_trigger_cmd.DAI_TRIGGER_DROP (*C enumerator*), 3206
dai_trigger_cmd.DAI_TRIGGER_PAUSE (*C enumerator*), 3205
dai_trigger_cmd.DAI_TRIGGER_POST_STOP (*C enumerator*), 3205
dai_trigger_cmd.DAI_TRIGGER_PRE_START (*C enumerator*), 3205
dai_trigger_cmd.DAI_TRIGGER_PREPARE (*C enumerator*), 3206
dai_trigger_cmd.DAI_TRIGGER_RESET (*C enumerator*), 3206
dai_trigger_cmd.DAI_TRIGGER_START (*C enumerator*), 3205
dai_trigger_cmd.DAI_TRIGGER_STOP (*C enumerator*), 3205
dai_ts_cfg (*C struct*), 3210
dai_ts_cfg.direction (*C var*), 3210
dai_ts_cfg.dma_chan_count (*C var*), 3211
dai_ts_cfg.dma_chan_index (*C var*), 3211
dai_ts_cfg.dma_id (*C var*), 3210
dai_ts_cfg.index (*C var*), 3210
dai_ts_cfg.type (*C var*), 3210
dai_ts_cfg.walclk_rate (*C var*), 3210
dai_ts_config (*C function*), 3208
dai_ts_data (*C struct*), 3211
dai_ts_data.sample (*C var*), 3211
dai_ts_data.walclk (*C var*), 3211
dai_ts_data.walclk_rate (*C var*), 3211
dai_ts_get (*C function*), 3209
dai_ts_start (*C function*), 3208
dai_ts_stop (*C function*), 3208
dai_type (*C enum*), 3203
dai_type.DAI_AMD_BT (*C enumerator*), 3203
dai_type.DAI_AMD_DMIC (*C enumerator*), 3204
dai_type.DAI_AMD_SP (*C enumerator*), 3204
dai_type.DAI_IMX_ESAI (*C enumerator*), 3203
dai_type.DAI_IMX_SAI (*C enumerator*), 3203
dai_type.DAI_INTEL_ALH (*C enumerator*), 3203
dai_type.DAI_INTEL_ALH_NHLT (*C enumerator*), 3204
dai_type.DAI_INTEL_DMIC (*C enumerator*), 3203
dai_type.DAI_INTEL_DMIC_NHLT (*C enumerator*), 3204
dai_type.DAI_INTEL_HDA (*C enumerator*), 3203
dai_type.DAI_INTEL_HDA_NHLT (*C enumerator*), 3204
dai_type.DAI_INTEL_SSP (*C enumerator*), 3203
dai_type.DAI_INTEL_SSP_NHLT (*C enumerator*), 3204
dai_type.DAI_LEGACY_I2S (*C enumerator*), 3203
dai_type.DAI_MEDIATEK_AFE (*C enumerator*), 3204
DeprecatedAction (*class in runners.core*), 196
dev_id_help() (*runners.core.ZephyrBinaryRunner class method*), 199
device (*C struct*), 530
device runtime power management, **3936**
DEVICE_DECLARE (*C macro*), 524
DEVICE_DEFINE (*C macro*), 521
DEVICE_DT_DEFER (*C macro*), 522
DEVICE_DT_DEFINE (*C macro*), 522
DEVICE_DT_GET (*C macro*), 523

DEVICE_DT_GET_ANY (*C macro*), 523
DEVICE_DT_GET_ONE (*C macro*), 524
DEVICE_DT_GET_OR_NULL (*C macro*), 524
DEVICE_DT_INST_DEFINE (*C macro*), 523
DEVICE_DT_INST_GET (*C macro*), 523
DEVICE_DT_NAME (*C macro*), 522
DEVICE_DT_NAME_GET (*C macro*), 523
device_from_handle (*C function*), 526
DEVICE_GET (*C macro*), 524
device_get_binding (*C function*), 529
device_handle_get (*C function*), 526
DEVICE_HANDLE_NULL (*C macro*), 521
device_handle_t (*C type*), 525
device_init (*C function*), 529
DEVICE_INIT_DT_GET (*C macro*), 525
DEVICE_INIT_GET (*C macro*), 525
device_injected_handles_get (*C function*), 527
device_is_ready (*C function*), 529
DEVICE_NAME_GET (*C macro*), 521
DEVICE_PCIE_DECLARE (*C macro*), 3507
DEVICE_PCIE_INIT (*C macro*), 3507
DEVICE_PCIE_INST_DECLARE (*C macro*), 3507
DEVICE_PCIE_INST_INIT (*C macro*), 3508
device_required_foreach (*C function*), 527
device_required_handles_get (*C function*), 526
device_state (*C struct*), 529
device_state.init_res (*C var*), 530
device_state.initialized (*C var*), 530
device_supported_foreach (*C function*), 528
device_supported_handles_get (*C function*), 527
device_visitor_callback_t (*C type*), 525
DeviceAdapter (*class in twister_harness*), 268
device.api (*C var*), 530
device.config (*C var*), 530
device.data (*C var*), 530
device.deps (*C var*), 530
device.name (*C var*), 530
device.state (*C var*), 530
disconnect() (*twister_harness.DeviceAdapter method*), 268
disk_access_init (*C function*), 1183
disk_access_ioctl (*C function*), 1184
disk_access_read (*C function*), 1184
disk_access_register (*C function*), 1186
disk_access_status (*C function*), 1183
disk_access_unregister (*C function*), 1186
disk_access_write (*C function*), 1184
disk_info (*C struct*), 1186
disk_info.dev (*C var*), 1186
disk_info.name (*C var*), 1186
disk_info.node (*C var*), 1186
disk_info.ops (*C var*), 1186
disk_info.refcnt (*C var*), 1186
DISK_IOCTL_CTRL_DEINIT (*C macro*), 1185
DISK_IOCTL_CTRL_INIT (*C macro*), 1185
DISK_IOCTL_CTRL_SYNC (*C macro*), 1185
DISK_IOCTL_GET_ERASE_BLOCK_SZ (*C macro*), 1185
DISK_IOCTL_GET_SECTOR_COUNT (*C macro*), 1185
DISK_IOCTL_GET_SECTOR_SIZE (*C macro*), 1185

DISK_IOCTL_RESERVED (*C macro*), 1185
disk_operations (*C struct*), 1187
DISK_STATUS_NOMEDIA (*C macro*), 1186
DISK_STATUS_OK (*C macro*), 1185
DISK_STATUS_UNINIT (*C macro*), 1186
DISK_STATUS_WR_PROTECT (*C macro*), 1186
DISPLAY_BITS_PER_PIXEL (*C macro*), 3291
display_blanking_off (*C function*), 3294
display_blanking_off_api (*C type*), 3291
display_blanking_on (*C function*), 3294
display_blanking_on_api (*C type*), 3291
display_buffer_descriptor (*C struct*), 3296
display_buffer_descriptor.buf_size (*C var*), 3296
display_buffer_descriptor.height (*C var*), 3296
display_buffer_descriptor.pitch (*C var*), 3296
display_buffer_descriptor.width (*C var*), 3296
display_capabilities (*C struct*), 3295
display_capabilities.current_orientation (*C var*), 3296
display_capabilities.current_pixel_format (*C var*), 3296
display_capabilities.screen_info (*C var*), 3296
display_capabilities.supported_pixel_formats (*C var*), 3296
display_capabilities.x_resolution (*C var*), 3296
display_capabilities.y_resolution (*C var*), 3296
display_driver_api (*C struct*), 3296
display_get_capabilities (*C function*), 3295
display_get_capabilities_api (*C type*), 3291
display_get_framebuffer (*C function*), 3294
display_get_framebuffer_api (*C type*), 3291
display_orientation (*C enum*), 3293
display_orientation.DISPLAY_ORIENTATION_NORMAL (*C enumerator*), 3293
display_orientation.DISPLAY_ORIENTATION_ROTATED_90 (*C enumerator*), 3293
display_orientation.DISPLAY_ORIENTATION_ROTATED_180 (*C enumerator*), 3293
display_orientation.DISPLAY_ORIENTATION_ROTATED_270 (*C enumerator*), 3293
display_pixel_format (*C enum*), 3292
display_pixel_format.PIXEL_FORMAT_ARGB_8888 (*C enumerator*), 3292
display_pixel_format.PIXEL_FORMAT_BGR_565 (*C enumerator*), 3292
display_pixel_format.PIXEL_FORMAT_MONO01 (*C enumerator*), 3292
display_pixel_format.PIXEL_FORMAT_MONO10 (*C enumerator*), 3292
display_pixel_format.PIXEL_FORMAT_RGB_565 (*C enumerator*), 3292
display_pixel_format.PIXEL_FORMAT_RGB_888 (*C enumerator*), 3292
display_read (*C function*), 3293
display_read_api (*C type*), 3291
display_screen_info (*C enum*), 3292
display_screen_info.SCREEN_INFO_DOUBLE_BUFFER (*C enumerator*), 3292
display_screen_info.SCREEN_INFO_EPD (*C enumerator*), 3292
display_screen_info.SCREEN_INFO_MONO_MSB_FIRST (*C enumerator*), 3292
display_screen_info.SCREEN_INFO_MONO_VTILED (*C enumerator*), 3292
display_screen_info.SCREEN_INFO_X_ALIGNMENT_WIDTH (*C enumerator*), 3293
display_set_brightness (*C function*), 3294
display_set_brightness_api (*C type*), 3291
display_set_contrast (*C function*), 3295
display_set_contrast_api (*C type*), 3291
display_set_orientation (*C function*), 3295
display_set_orientation_api (*C type*), 3291
display_set_pixel_format (*C function*), 3295
display_set_pixel_format_api (*C type*), 3291
display_write (*C function*), 3293
display_write_api (*C type*), 3291

DIV_ROUND_CLOSEST (*C macro*), 683
DIV_ROUND_UP (*C macro*), 682
dma_addr_adj (*C enum*), 3281
dma_addr_adj.DMA_ADDR_ADJ_DECREMENT (*C enumerator*), 3281
dma_addr_adj.DMA_ADDR_ADJ_INCREMENT (*C enumerator*), 3281
dma_addr_adj.DMA_ADDR_ADJ_NO_CHANGE (*C enumerator*), 3281
dma_attribute_type (*C enum*), 3281
dma_attribute_type.DMA_ATTR_BUFFER_ADDRESS_ALIGNMENT (*C enumerator*), 3281
dma_attribute_type.DMA_ATTR_BUFFER_SIZE_ALIGNMENT (*C enumerator*), 3281
dma_attribute_type.DMA_ATTR_COPY_ALIGNMENT (*C enumerator*), 3281
dma_attribute_type.DMA_ATTR_MAX_BLOCK_COUNT (*C enumerator*), 3281
dma_block_config (*C struct*), 3286
dma_block_config.block_size (*C var*), 3286
dma_block_config.dest_addr_adj (*C var*), 3287
dma_block_config.dest_address (*C var*), 3286
dma_block_config.dest_reload_en (*C var*), 3287
dma_block_config.dest_scatter_count (*C var*), 3286
dma_block_config.dest_scatter_en (*C var*), 3287
dma_block_config.dest_scatter_interval (*C var*), 3286
dma_block_config.fifo_mode_control (*C var*), 3287
dma_block_config.flow_control_mode (*C var*), 3287
dma_block_config.next_block (*C var*), 3286
dma_block_config.source_addr_adj (*C var*), 3287
dma_block_config.source_address (*C var*), 3286
dma_block_config.source_gather_count (*C var*), 3286
dma_block_config.source_gather_en (*C var*), 3287
dma_block_config.source_gather_interval (*C var*), 3286
dma_block_config.source_reload_en (*C var*), 3287
DMA_BUF_ADDR_ALIGNMENT (*C macro*), 3279
DMA_BUF_SIZE_ALIGNMENT (*C macro*), 3279
dma_burst_index (*C function*), 3286
dma_callback_t (*C type*), 3280
dma_chan_filter (*C function*), 3284
dma_channel_direction (*C enum*), 3280
dma_channel_direction.DMA_CHANNEL_DIRECTION_COMMON_COUNT (*C enumerator*), 3280
dma_channel_direction.DMA_CHANNEL_DIRECTION_MAX (*C enumerator*), 3281
dma_channel_direction.DMA_CHANNEL_DIRECTION_PRIV_START (*C enumerator*), 3281
dma_channel_direction.HOST_TO_MEMORY (*C enumerator*), 3280
dma_channel_direction.MEMORY_TO_HOST (*C enumerator*), 3280
dma_channel_direction.MEMORY_TO_MEMORY (*C enumerator*), 3280
dma_channel_direction.MEMORY_TO_PERIPHERAL (*C enumerator*), 3280
dma_channel_direction.PERIPHERAL_TO_MEMORY (*C enumerator*), 3280
dma_channel_direction.PERIPHERAL_TO_PERIPHERAL (*C enumerator*), 3280
dma_channel_filter (*C enum*), 3281
dma_channel_filter.DMA_CHANNEL_NORMAL (*C enumerator*), 3281
dma_channel_filter.DMA_CHANNEL_PERIODIC (*C enumerator*), 3281
dma_config (*C function*), 3282
dma_config (*C struct*), 3287
dma_config.block_count (*C var*), 3289
dma_config.channel_direction (*C var*), 3287
dma_config.channel_priority (*C var*), 3288
dma_config.complete_callback_en (*C var*), 3288
dma_config.cyclic (*C var*), 3288
dma_config.dest_burst_length (*C var*), 3289
dma_config.dest_chaining_en (*C var*), 3288
dma_config.dest_data_size (*C var*), 3289
dma_config.dest_handshake (*C var*), 3288
dma_config.dma_callback (*C var*), 3289

dma_config.dma_slot (*C var*), 3287
dma_config.error_callback_dis (*C var*), 3288
dma_config.head_block (*C var*), 3289
dma_config.linked_channel (*C var*), 3288
dma_config.source_burst_length (*C var*), 3289
dma_config.source_chaining_en (*C var*), 3288
dma_config.source_data_size (*C var*), 3288
dma_config.source_handshake (*C var*), 3288
dma_config.user_data (*C var*), 3289
dma_context (*C struct*), 3290
dma_context.atomic (*C var*), 3290
dma_context.dma_channels (*C var*), 3290
dma_context.magic (*C var*), 3290
DMA_COPY_ALIGNMENT (*C macro*), 3279
dma_get_attribute (*C function*), 3285
dma_get_status (*C function*), 3284
DMA_MAGIC (*C macro*), 3279
dma_release_channel (*C function*), 3284
dma_reload (*C function*), 3282
dma_request_channel (*C function*), 3284
dma_resume (*C function*), 3283
dma_start (*C function*), 3282
dma_status (*C struct*), 3289
DMA_STATUS_BLOCK (*C macro*), 3279
DMA_STATUS_COMPLETE (*C macro*), 3279
dma_status.busy (*C var*), 3289
dma_status.dir (*C var*), 3289
dma_status.free (*C var*), 3289
dma_status.pending_length (*C var*), 3289
dma_status.read_position (*C var*), 3289
dma_status.total_copied (*C var*), 3289
dma_status.write_position (*C var*), 3289
dma_stop (*C function*), 3282
dma_suspend (*C function*), 3283
dma_width_index (*C function*), 3285
dmic_build_channel_map (*C function*), 3188
dmic_build_clk_skew_map (*C function*), 3188
dmic_cfg (*C struct*), 3191
dmic_cfg.streams (*C var*), 3191
dmic_configure (*C function*), 3188
dmic_parse_channel_map (*C function*), 3188
dmic_read (*C function*), 3189
dmic_state (*C enum*), 3187
dmic_state.DMIC_STATE_ACTIVE (*C enumerator*), 3187
dmic_state.DMIC_STATE_CONFIGURED (*C enumerator*), 3187
dmic_state.DMIC_STATE_ERROR (*C enumerator*), 3187
dmic_state.DMIC_STATE_INITIALIZED (*C enumerator*), 3187
dmic_state.DMIC_STATE_PAUSED (*C enumerator*), 3187
dmic_state.DMIC_STATE_UNINIT (*C enumerator*), 3187
dmic_trigger (*C enum*), 3187
dmic_trigger (*C function*), 3189
dmic_trigger.DMIC_TRIGGER_PAUSE (*C enumerator*), 3187
dmic_trigger.DMIC_TRIGGER_RELEASE (*C enumerator*), 3187
dmic_trigger.DMIC_TRIGGER_RESET (*C enumerator*), 3187
dmic_trigger.DMIC_TRIGGER_START (*C enumerator*), 3187
dmic_trigger.DMIC_TRIGGER_STOP (*C enumerator*), 3187
dns_addrinfo (*C struct*), 2535
dns_addrinfo.ai_addr (*C var*), 2535

dns_addrinfo.ai_addrlen (*C var*), 2535
dns_addrinfo.ai_canonname (*C var*), 2535
dns_addrinfo.ai_family (*C var*), 2535
dns_cancel_addr_info (*C function*), 2534
dns_get_addr_info (*C function*), 2534
DNS_MAX_NAME_SIZE (*C macro*), 2530
dns_query_type (*C enum*), 2530
dns_query_type.DNS_QUERY_TYPE_A (*C enumerator*), 2530
dns_query_type.DNS_QUERY_TYPE_AAAA (*C enumerator*), 2530
dns_resolve_cancel (*C function*), 2533
dns_resolve_cancel_with_name (*C function*), 2533
dns_resolve_cb_t (*C type*), 2530
dns_resolve_close (*C function*), 2532
dns_resolve_context (*C struct*), 2535
dns_resolve_context.buf_timeout (*C var*), 2535
dns_resolve_context.dns_pending_query (*C struct*), 2535
dns_resolve_context.dns_pending_query.cb (*C var*), 2536
dns_resolve_context.dns_pending_query.ctx (*C var*), 2535
dns_resolve_context.dns_pending_query.id (*C var*), 2536
dns_resolve_context.dns_pending_query.query (*C var*), 2536
dns_resolve_context.dns_pending_query.query_hash (*C var*), 2536
dns_resolve_context.dns_pending_query.query_type (*C var*), 2536
dns_resolve_context.dns_pending_query.timeout (*C var*), 2536
dns_resolve_context.dns_pending_query.timer (*C var*), 2535
dns_resolve_context.dns_pending_query.user_data (*C var*), 2536
dns_resolve_context.dns_server (*C struct*), 2536
dns_resolve_context.dns_server.dns_server (*C var*), 2536
dns_resolve_context.dns_server.is_llmnr (*C var*), 2536
dns_resolve_context.dns_server.is_mdns (*C var*), 2536
dns_resolve_context.dns_server.sock (*C var*), 2536
dns_resolve_context.lock (*C var*), 2535
dns_resolve_context.state (*C var*), 2535
dns_resolve_get_default (*C function*), 2534
dns_resolve_init (*C function*), 2532
dns_resolve_init_default (*C function*), 2532
dns_resolve_name (*C function*), 2533
dns_resolve_reconfigure (*C function*), 2532
dns_resolve_status (*C enum*), 2530
dns_resolve_status.DNS_EAI_ADDRFAMILY (*C enumerator*), 2531
dns_resolve_status.DNS_EAI_AGAIN (*C enumerator*), 2531
dns_resolve_status.DNS_EAI_ALLDONE (*C enumerator*), 2531
dns_resolve_status.DNS_EAI_BADFLAGS (*C enumerator*), 2530
dns_resolve_status.DNS_EAI_CANCELED (*C enumerator*), 2531
dns_resolve_status.DNS_EAI_FAIL (*C enumerator*), 2531
dns_resolve_status.DNS_EAI_FAMILY (*C enumerator*), 2531
dns_resolve_status.DNS_EAI_IDN_ENCODE (*C enumerator*), 2531
dns_resolve_status.DNS_EAI_INPROGRESS (*C enumerator*), 2531
dns_resolve_status.DNS_EAI_MEMORY (*C enumerator*), 2531
dns_resolve_status.DNS_EAI_NODATA (*C enumerator*), 2531
dns_resolve_status.DNS_EAI_NONAME (*C enumerator*), 2530
dns_resolve_status.DNS_EAI_NOTCANCELED (*C enumerator*), 2531
dns_resolve_status.DNS_EAI_OVERFLOW (*C enumerator*), 2531
dns_resolve_status.DNS_EAI_SERVICE (*C enumerator*), 2531
dns_resolve_status.DNS_EAI_SOCKTYPE (*C enumerator*), 2531
dns_resolve_status.DNS_EAI_SYSTEM (*C enumerator*), 2531
do_add_parser() (*runners.core.ZephyrBinaryRunner class method*), 199
do_create() (*runners.core.ZephyrBinaryRunner class method*), 200
do_run() (*runners.core.ZephyrBinaryRunner method*), 200

DT_ALIAS (*C macro*), 1381
DT_ANY_INST_HAS_PROP_STATUS_OKAY (*C macro*), 1448
DT_ANY_INST_ON_BUS_STATUS_OKAY (*C macro*), 1447
DT_BUS (*C macro*), 1431
DT_CAN_TRANSCEIVER_MAX_BITRATE (*C macro*), 1454
DT_CAN_TRANSCEIVER_MIN_BITRATE (*C macro*), 1453
DT_CHILD (*C macro*), 1384
DT_CHILD_NUM (*C macro*), 1386
DT_CHILD_NUM_STATUS_OKAY (*C macro*), 1386
DT_CHOSEN (*C macro*), 1512
DT_CLOCKS_CELL (*C macro*), 1459
DT_CLOCKS_CELL_BY_IDX (*C macro*), 1458
DT_CLOCKS_CELL_BY_NAME (*C macro*), 1459
DT_CLOCKS_CTLR (*C macro*), 1457
DT_CLOCKS_CTLR_BY_IDX (*C macro*), 1457
DT_CLOCKS_CTLR_BY_NAME (*C macro*), 1458
DT_CLOCKS_HAS_IDX (*C macro*), 1455
DT_CLOCKS_HAS_NAME (*C macro*), 1456
DT_COMPAT_GET_ANY_STATUS_OKAY (*C macro*), 1384
DT_DEP_ORD (*C macro*), 1429
DT_DEP_ORD_STR_SORTABLE (*C macro*), 1429
DT_DMAS_CELL_BY_IDX (*C macro*), 1464
DT_DMAS_CELL_BY_NAME (*C macro*), 1466
DT_DMAS_CTLR (*C macro*), 1463
DT_DMAS_CTLR_BY_IDX (*C macro*), 1462
DT_DMAS_CTLR_BY_NAME (*C macro*), 1463
DT_DMAS_HAS_IDX (*C macro*), 1467
DT_DMAS_HAS_NAME (*C macro*), 1467
DT_DRV_INST (*C macro*), 1432
DT_ENUM_HAS_VALUE (*C macro*), 1391
DT_ENUM_IDX (*C macro*), 1391
DT_ENUM_IDX_OR (*C macro*), 1391
DT_FIXED_PARTITION_ADDR (*C macro*), 1469
DT_FIXED_PARTITION_EXISTS (*C macro*), 1468
DT_FIXED_PARTITION_ID (*C macro*), 1468
DT_FOREACH_CHILD (*C macro*), 1417
DT_FOREACH_CHILD_SEP (*C macro*), 1418
DT_FOREACH_CHILD_SEP_VARGS (*C macro*), 1419
DT_FOREACH_CHILD_STATUS_OKAY (*C macro*), 1419
DT_FOREACH_CHILD_STATUS_OKAY_SEP (*C macro*), 1420
DT_FOREACH_CHILD_STATUS_OKAY_SEP_VARGS (*C macro*), 1420
DT_FOREACH_CHILD_STATUS_OKAY_VARGS (*C macro*), 1420
DT_FOREACH_CHILD_VARGS (*C macro*), 1419
DT_FOREACH_NODE (*C macro*), 1417
DT_FOREACH_NODE_VARGS (*C macro*), 1417
DT_FOREACH_NODELABEL (*C macro*), 1425
DT_FOREACH_NODELABEL_VARGS (*C macro*), 1425
DT_FOREACH_PROP_ELEM (*C macro*), 1421
DT_FOREACH_PROP_ELEM_SEP (*C macro*), 1422
DT_FOREACH_PROP_ELEM_SEP_VARGS (*C macro*), 1423
DT_FOREACH_PROP_ELEM_VARGS (*C macro*), 1422
DT_FOREACH_RANGE (*C macro*), 1408
DT_FOREACH_STATUS_OKAY (*C macro*), 1423
DT_FOREACH_STATUS_OKAY_NODE (*C macro*), 1417
DT_FOREACH_STATUS_OKAY_NODE_VARGS (*C macro*), 1417
DT_FOREACH_STATUS_OKAY_VARGS (*C macro*), 1424
DT_GPARENT (*C macro*), 1383
DT_GPIO_CTLR (*C macro*), 1470

DT_GPIO_CTLR_BY_IDX (*C macro*), 1470
DT_GPIO_FLAGS (*C macro*), 1472
DT_GPIO_FLAGS_BY_IDX (*C macro*), 1472
DT_GPIO_HOG_FLAGS_BY_IDX (*C macro*), 1474
DT_GPIO_HOG_PIN_BY_IDX (*C macro*), 1473
DT_GPIO_PIN (*C macro*), 1471
DT_GPIO_PIN_BY_IDX (*C macro*), 1470
DT_HAS_CHOSEN (*C macro*), 1512
DT_HAS_COMPAT_ON_BUS_STATUS_OKAY (*C macro*), 1447
DT_HAS_COMPAT_STATUS_OKAY (*C macro*), 1427
DT_HAS_FIXED_PARTITION_LABEL (*C macro*), 1468
DT_INST (*C macro*), 1382
DT_INST_BUS (*C macro*), 1446
DT_INST_CAN_TRANSCEIVER_MAX_BITRATE (*C macro*), 1455
DT_INST_CAN_TRANSCEIVER_MIN_BITRATE (*C macro*), 1454
DT_INST_CHILD (*C macro*), 1433
DT_INST_CHILD_NUM (*C macro*), 1433
DT_INST_CHILD_NUM_STATUS_OKAY (*C macro*), 1434
DT_INST_CLOCKS_CELL (*C macro*), 1462
DT_INST_CLOCKS_CELL_BY_IDX (*C macro*), 1461
DT_INST_CLOCKS_CELL_BY_NAME (*C macro*), 1461
DT_INST_CLOCKS_CTLR (*C macro*), 1461
DT_INST_CLOCKS_CTLR_BY_IDX (*C macro*), 1460
DT_INST_CLOCKS_CTLR_BY_NAME (*C macro*), 1461
DT_INST_CLOCKS_HAS_IDX (*C macro*), 1460
DT_INST_CLOCKS_HAS_NAME (*C macro*), 1460
DT_INST_DEP_ORD (*C macro*), 1430
DT_INST_DMAS_CELL_BY_IDX (*C macro*), 1465
DT_INST_DMAS_CELL_BY_NAME (*C macro*), 1466
DT_INST_DMAS_CTLR (*C macro*), 1464
DT_INST_DMAS_CTLR_BY_IDX (*C macro*), 1464
DT_INST_DMAS_CTLR_BY_NAME (*C macro*), 1464
DT_INST_DMAS_HAS_IDX (*C macro*), 1467
DT_INST_DMAS_HAS_NAME (*C macro*), 1467
DT_INST_ENUM_HAS_VALUE (*C macro*), 1437
DT_INST_ENUM_IDX (*C macro*), 1437
DT_INST_ENUM_IDX_OR (*C macro*), 1437
DT_INST_FOREACH_CHILD (*C macro*), 1434
DT_INST_FOREACH_CHILD_SEP (*C macro*), 1434
DT_INST_FOREACH_CHILD_SEP_VARGS (*C macro*), 1435
DT_INST_FOREACH_CHILD_STATUS_OKAY (*C macro*), 1436
DT_INST_FOREACH_CHILD_STATUS_OKAY_SEP (*C macro*), 1436
DT_INST_FOREACH_CHILD_STATUS_OKAY_SEP_VARGS (*C macro*), 1437
DT_INST_FOREACH_CHILD_STATUS_OKAY_VARGS (*C macro*), 1436
DT_INST_FOREACH_CHILD_VARGS (*C macro*), 1435
DT_INST_FOREACH_NODELABEL (*C macro*), 1450
DT_INST_FOREACH_NODELABEL_VARGS (*C macro*), 1450
DT_INST_FOREACH_PROP_ELEM (*C macro*), 1450
DT_INST_FOREACH_PROP_ELEM_SEP (*C macro*), 1450
DT_INST_FOREACH_PROP_ELEM_SEP_VARGS (*C macro*), 1451
DT_INST_FOREACH_PROP_ELEM_VARGS (*C macro*), 1450
DT_INST_FOREACH_STATUS_OKAY (*C macro*), 1448
DT_INST_FOREACH_STATUS_OKAY_VARGS (*C macro*), 1449
DT_INST_GPARENT (*C macro*), 1433
DT_INST_GPIO_FLAGS (*C macro*), 1476
DT_INST_GPIO_FLAGS_BY_IDX (*C macro*), 1475
DT_INST_GPIO_PIN (*C macro*), 1475
DT_INST_GPIO_PIN_BY_IDX (*C macro*), 1475

DT_INST_IO_CHANNELS_CTLR (*C macro*), 1478
DT_INST_IO_CHANNELS_CTLR_BY_IDX (*C macro*), 1478
DT_INST_IO_CHANNELS_CTLR_BY_NAME (*C macro*), 1478
DT_INST_IO_CHANNELS_INPUT (*C macro*), 1481
DT_INST_IO_CHANNELS_INPUT_BY_IDX (*C macro*), 1481
DT_INST_IO_CHANNELS_INPUT_BY_NAME (*C macro*), 1481
DT_INST_IRQ (*C macro*), 1446
DT_INST_IRQ_BY_IDX (*C macro*), 1445
DT_INST_IRQ_BY_NAME (*C macro*), 1446
DT_INST_IRQ_HAS_CELL (*C macro*), 1452
DT_INST_IRQ_HAS_CELL_AT_IDX (*C macro*), 1452
DT_INST_IRQ_HAS_IDX (*C macro*), 1452
DT_INST_IRQ_HAS_NAME (*C macro*), 1452
DT_INST_IRQ_INTC (*C macro*), 1445
DT_INST_IRQ_INTC_BY_IDX (*C macro*), 1445
DT_INST_IRQ_INTC_BY_NAME (*C macro*), 1445
DT_INST_IRQ_LEVEL (*C macro*), 1444
DT_INST_IRQN (*C macro*), 1446
DT_INST_IRQN_BY_IDX (*C macro*), 1446
DT_INST_NODE_HAS_COMPAT (*C macro*), 1451
DT_INST_NODE_HAS_PROP (*C macro*), 1451
DT_INST_NODELABEL_STRING_ARRAY (*C macro*), 1434
DT_INST_NUM_CLOCKS (*C macro*), 1460
DT_INST_NUM_NODELABELS (*C macro*), 1434
DT_INST_NUM_PINCTRL_STATES (*C macro*), 1490
DT_INST_NUM_PINCTRLS_BY_IDX (*C macro*), 1489
DT_INST_NUM_PINCTRLS_BY_NAME (*C macro*), 1489
DT_INST_ON_BUS (*C macro*), 1446
DT_INST_PARENT (*C macro*), 1432
DT_INST_PHA (*C macro*), 1441
DT_INST_PHA_BY_IDX (*C macro*), 1440
DT_INST_PHA_BY_IDX_OR (*C macro*), 1441
DT_INST_PHA_BY_NAME (*C macro*), 1441
DT_INST_PHA_BY_NAME_OR (*C macro*), 1441
DT_INST_PHA_HAS_CELL (*C macro*), 1452
DT_INST_PHA_HAS_CELL_AT_IDX (*C macro*), 1451
DT_INST_PHA_OR (*C macro*), 1441
DT_INST_PHANDLE (*C macro*), 1442
DT_INST_PHANDLE_BY_IDX (*C macro*), 1442
DT_INST_PHANDLE_BY_NAME (*C macro*), 1442
DT_INST_PINCTRL_0 (*C macro*), 1488
DT_INST_PINCTRL_BY_IDX (*C macro*), 1488
DT_INST_PINCTRL_BY_NAME (*C macro*), 1488
DT_INST_PINCTRL_HAS_IDX (*C macro*), 1490
DT_INST_PINCTRL_HAS_NAME (*C macro*), 1490
DT_INST_PINCTRL_IDX_TO_NAME_TOKEN (*C macro*), 1489
DT_INST_PINCTRL_IDX_TO_NAME_UPPER_TOKEN (*C macro*), 1489
DT_INST_PINCTRL_NAME_TO_IDX (*C macro*), 1489
DT_INST_PROP (*C macro*), 1437
DT_INST_PROP_BY_IDX (*C macro*), 1438
DT_INST_PROP_BY_PHANDLE (*C macro*), 1440
DT_INST_PROP_BY_PHANDLE_IDX (*C macro*), 1440
DT_INST_PROP_HAS_IDX (*C macro*), 1438
DT_INST_PROP_HAS_NAME (*C macro*), 1438
DT_INST_PROP_LEN (*C macro*), 1438
DT_INST_PROP_LEN_OR (*C macro*), 1439
DT_INST_PROP_OR (*C macro*), 1438
DT_INST_PWMS_CELL (*C macro*), 1498

DT_INST_PWMS_CELL_BY_IDX (*C macro*), 1498
DT_INST_PWMS_CELL_BY_NAME (*C macro*), 1498
DT_INST_PWMS_CHANNEL (*C macro*), 1499
DT_INST_PWMS_CHANNEL_BY_IDX (*C macro*), 1498
DT_INST_PWMS_CHANNEL_BY_NAME (*C macro*), 1499
DT_INST_PWMS_CTLR (*C macro*), 1497
DT_INST_PWMS_CTLR_BY_IDX (*C macro*), 1497
DT_INST_PWMS_CTLR_BY_NAME (*C macro*), 1497
DT_INST_PWMS_FLAGS (*C macro*), 1501
DT_INST_PWMS_FLAGS_BY_IDX (*C macro*), 1500
DT_INST_PWMS_FLAGS_BY_NAME (*C macro*), 1500
DT_INST_PWMS_PERIOD (*C macro*), 1500
DT_INST_PWMS_PERIOD_BY_IDX (*C macro*), 1499
DT_INST_PWMS_PERIOD_BY_NAME (*C macro*), 1499
DT_INST_REG_ADDR (*C macro*), 1444
DT_INST_REG_ADDR_BY_IDX (*C macro*), 1443
DT_INST_REG_ADDR_BY_NAME (*C macro*), 1443
DT_INST_REG_ADDR_BY_NAME_OR (*C macro*), 1443
DT_INST_REG_ADDR_BY_NAME_U64 (*C macro*), 1443
DT_INST_REG_ADDR_U64 (*C macro*), 1444
DT_INST_REG_HAS_IDX (*C macro*), 1442
DT_INST_REG_HAS_NAME (*C macro*), 1442
DT_INST_REG_SIZE (*C macro*), 1444
DT_INST_REG_SIZE_BY_IDX (*C macro*), 1443
DT_INST_REG_SIZE_BY_NAME (*C macro*), 1444
DT_INST_REG_SIZE_BY_NAME_OR (*C macro*), 1444
DT_INST_REQUIRES_DEP_ORDS (*C macro*), 1430
DT_INST_RESET_CELL (*C macro*), 1506
DT_INST_RESET_CELL_BY_IDX (*C macro*), 1505
DT_INST_RESET_CELL_BY_NAME (*C macro*), 1505
DT_INST_RESET_CTLR (*C macro*), 1504
DT_INST_RESET_CTLR_BY_IDX (*C macro*), 1504
DT_INST_RESET_CTLR_BY_NAME (*C macro*), 1505
DT_INST_RESET_ID (*C macro*), 1507
DT_INST_RESET_ID_BY_IDX (*C macro*), 1507
DT_INST_SPI_DEV_CS_GPIOS_CTLR (*C macro*), 1511
DT_INST_SPI_DEV_CS_GPIOS_FLAGS (*C macro*), 1512
DT_INST_SPI_DEV_CS_GPIOS_PIN (*C macro*), 1511
DT_INST_SPI_DEV_HAS_CS_GPIOS (*C macro*), 1511
DT_INST_STRING_TOKEN (*C macro*), 1439
DT_INST_STRING_TOKEN_BY_IDX (*C macro*), 1439
DT_INST_STRING_TOKEN_OR (*C macro*), 1447
DT_INST_STRING_UNQUOTED (*C macro*), 1439
DT_INST_STRING_UNQUOTED_BY_IDX (*C macro*), 1440
DT_INST_STRING_UNQUOTED_OR (*C macro*), 1447
DT_INST_STRING_UPPER_TOKEN (*C macro*), 1439
DT_INST_STRING_UPPER_TOKEN_BY_IDX (*C macro*), 1440
DT_INST_STRING_UPPER_TOKEN_OR (*C macro*), 1447
DT_INST_SUPPORTS_DEP_ORDS (*C macro*), 1430
DT_INVALID_NODE (*C macro*), 1379
DT_IO_CHANNELS_CTLR (*C macro*), 1477
DT_IO_CHANNELS_CTLR_BY_IDX (*C macro*), 1476
DT_IO_CHANNELS_CTLR_BY_NAME (*C macro*), 1477
DT_IO_CHANNELS_INPUT (*C macro*), 1480
DT_IO_CHANNELS_INPUT_BY_IDX (*C macro*), 1479
DT_IO_CHANNELS_INPUT_BY_NAME (*C macro*), 1480
DT_IRQ (*C macro*), 1414
DT_IRQ_BY_IDX (*C macro*), 1413

DT_IRQ_BY_NAME (*C macro*), 1413
DT_IRQ_HAS_CELL (*C macro*), 1412
DT_IRQ_HAS_CELL_AT_IDX (*C macro*), 1412
DT_IRQ_HAS_IDX (*C macro*), 1412
DT_IRQ_HAS_NAME (*C macro*), 1413
DT_IRQ_INTC (*C macro*), 1415
DT_IRQ_INTC_BY_IDX (*C macro*), 1414
DT_IRQ_INTC_BY_NAME (*C macro*), 1415
DT_IRQ_LEVEL (*C macro*), 1412
DT_IRQN (*C macro*), 1416
DT_IRQN_BY_IDX (*C macro*), 1416
DT_MBOX_CHANNEL_BY_NAME (*C macro*), 1482
DT_MBOX_CTLR_BY_NAME (*C macro*), 1482
DT_MEM_FROM_FIXED_PARTITION (*C macro*), 1469
DT_MEMORY_ATTR_FOREACH_STATUS_OKAY_NODE (*C macro*), 1014
DT_MTD_FROM_FIXED_PARTITION (*C macro*), 1469
DT_NODE_BY_FIXED_PARTITION_LABEL (*C macro*), 1468
DT_NODE_CHILD_IDX (*C macro*), 1386
DT_NODE_EXISTS (*C macro*), 1426
DT_NODE_FULL_NAME (*C macro*), 1385
DT_NODE_HAS_COMPAT (*C macro*), 1427
DT_NODE_HAS_COMPAT_STATUS (*C macro*), 1428
DT_NODE_HAS_PROP (*C macro*), 1428
DT_NODE_HAS_STATUS (*C macro*), 1426
DT_NODE_PATH (*C macro*), 1385
DT_NODELABEL (*C macro*), 1380
DT_NODELABEL_STRING_ARRAY (*C macro*), 1387
DT_NUM_CLOCKS (*C macro*), 1456
DT_NUM_CPU_POWER_STATES (*C macro*), 1060
DT_NUM_GPIO_HOGS (*C macro*), 1473
DT_NUM_INST_STATUS_OKAY (*C macro*), 1427
DT_NUM_IRQS (*C macro*), 1411
DT_NUM_NODELABELS (*C macro*), 1411
DT_NUM_PINCTRL_STATES (*C macro*), 1486
DT_NUM_PINCTRLS_BY_IDX (*C macro*), 1486
DT_NUM_PINCTRLS_BY_NAME (*C macro*), 1486
DT_NUM_RANGES (*C macro*), 1403
DT_NUM_REGS (*C macro*), 1408
DT_ON_BUS (*C macro*), 1431
DT_PARENT (*C macro*), 1383
DT_PATH (*C macro*), 1379
DT_PHA (*C macro*), 1399
DT_PHA_BY_IDX (*C macro*), 1398
DT_PHA_BY_IDX_OR (*C macro*), 1399
DT_PHA_BY_NAME (*C macro*), 1400
DT_PHA_BY_NAME_OR (*C macro*), 1400
DT_PHA_HAS_CELL (*C macro*), 1428
DT_PHA_HAS_CELL_AT_IDX (*C macro*), 1428
DT_PHA_OR (*C macro*), 1400
DT_PHANDLE (*C macro*), 1402
DT_PHANDLE_BY_IDX (*C macro*), 1402
DT_PHANDLE_BY_NAME (*C macro*), 1401
DT_PINCTRL_0 (*C macro*), 1484
DT_PINCTRL_BY_IDX (*C macro*), 1483
DT_PINCTRL_BY_NAME (*C macro*), 1484
DT_PINCTRL_HAS_IDX (*C macro*), 1487
DT_PINCTRL_HAS_NAME (*C macro*), 1487
DT_PINCTRL_IDX_TO_NAME_TOKEN (*C macro*), 1485

DT_PINCTRL_IDX_TO_NAME_UPPER_TOKEN (*C macro*), 1485
DT_PINCTRL_NAME_TO_IDX (*C macro*), 1484
DT_PROP (*C macro*), 1387
DT_PROP_BY_IDX (*C macro*), 1390
DT_PROP_BY_PHANDLE (*C macro*), 1398
DT_PROP_BY_PHANDLE_IDX (*C macro*), 1397
DT_PROP_BY_PHANDLE_IDX_OR (*C macro*), 1398
DT_PROP_HAS_IDX (*C macro*), 1389
DT_PROP_HAS_NAME (*C macro*), 1389
DT_PROP_LEN (*C macro*), 1388
DT_PROP_LEN_OR (*C macro*), 1388
DT_PROP_OR (*C macro*), 1390
DT_PWMS_CELL (*C macro*), 1493
DT_PWMS_CELL_BY_IDX (*C macro*), 1492
DT_PWMS_CELL_BY_NAME (*C macro*), 1492
DT_PWMS_CHANNEL (*C macro*), 1494
DT_PWMS_CHANNEL_BY_IDX (*C macro*), 1494
DT_PWMS_CHANNEL_BY_NAME (*C macro*), 1494
DT_PWMS_CTLR (*C macro*), 1491
DT_PWMS_CTLR_BY_IDX (*C macro*), 1490
DT_PWMS_CTLR_BY_NAME (*C macro*), 1491
DT_PWMS_FLAGS (*C macro*), 1496
DT_PWMS_FLAGS_BY_IDX (*C macro*), 1496
DT_PWMS_FLAGS_BY_NAME (*C macro*), 1496
DT_PWMS_PERIOD (*C macro*), 1495
DT_PWMS_PERIOD_BY_IDX (*C macro*), 1495
DT_PWMS_PERIOD_BY_NAME (*C macro*), 1495
DT_RANGES_CHILD_BUS_ADDRESS_BY_IDX (*C macro*), 1405
DT_RANGES_CHILD_BUS_FLAGS_BY_IDX (*C macro*), 1405
DT_RANGES_HAS_CHILD_BUS_FLAGS_AT_IDX (*C macro*), 1404
DT_RANGES_HAS_IDX (*C macro*), 1403
DT_RANGES_LENGTH_BY_IDX (*C macro*), 1407
DT_RANGES_PARENT_BUS_ADDRESS_BY_IDX (*C macro*), 1406
DT_REG_ADDR (*C macro*), 1409
DT_REG_ADDR_BY_IDX (*C macro*), 1409
DT_REG_ADDR_BY_NAME (*C macro*), 1410
DT_REG_ADDR_BY_NAME_OR (*C macro*), 1410
DT_REG_ADDR_BY_NAME_U64 (*C macro*), 1410
DT_REG_ADDR_U64 (*C macro*), 1410
DT_REG_HAS_IDX (*C macro*), 1409
DT_REG_HAS_NAME (*C macro*), 1409
DT_REG_SIZE (*C macro*), 1410
DT_REG_SIZE_BY_IDX (*C macro*), 1409
DT_REG_SIZE_BY_NAME (*C macro*), 1411
DT_REG_SIZE_BY_NAME_OR (*C macro*), 1411
DT_REQUIRES_DEP_ORDS (*C macro*), 1429
DT_RESET_CELL (*C macro*), 1504
DT_RESET_CELL_BY_IDX (*C macro*), 1503
DT_RESET_CELL_BY_NAME (*C macro*), 1503
DT_RESET_CTLR (*C macro*), 1502
DT_RESET_CTLR_BY_IDX (*C macro*), 1501
DT_RESET_CTLR_BY_NAME (*C macro*), 1502
DT_RESET_ID (*C macro*), 1507
DT_RESET_ID_BY_IDX (*C macro*), 1506
DT_ROOT (*C macro*), 1379
DT_SAME_NODE (*C macro*), 1386
DT_SPI_DEV_CS_GPIOS_CTLR (*C macro*), 1509
DT_SPI_DEV_CS_GPIOS_FLAGS (*C macro*), 1510

DT_SPI_DEV_CS_GPIOS_PIN (*C macro*), 1510
DT_SPI_DEV_HAS_CS_GPIOS (*C macro*), 1508
DT_SPI_HAS_CS_GPIOS (*C macro*), 1508
DT_SPI_NUM_CS_GPIOS (*C macro*), 1508
DT_STRING_TOKEN (*C macro*), 1392
DT_STRING_TOKEN_BY_IDX (*C macro*), 1395
DT_STRING_TOKEN_OR (*C macro*), 1393
DT_STRING_UNQUOTED (*C macro*), 1394
DT_STRING_UNQUOTED_BY_IDX (*C macro*), 1396
DT_STRING_UNQUOTED_OR (*C macro*), 1395
DT_STRING_UPPER_TOKEN (*C macro*), 1393
DT_STRING_UPPER_TOKEN_BY_IDX (*C macro*), 1395
DT_STRING_UPPER_TOKEN_OR (*C macro*), 1394
DT_SUPPORTS_DEP_ORDS (*C macro*), 1430

E

E2BIG (*C macro*), 76
EACCES (*C macro*), 76
EADDRINUSE (*C macro*), 79
EADDRNOTAVAIL (*C macro*), 80
EAFNOSUPPORT (*C macro*), 79
EAGAIN (*C macro*), 76
EAI_AGAIN (*C macro*), 2490
EAI_BADFLAGS (*C macro*), 2490
EAI_FAIL (*C macro*), 2490
EAI_FAMILY (*C macro*), 2491
EAI_MEMORY (*C macro*), 2490
EAI_NODATA (*C macro*), 2490
EAI_NONAME (*C macro*), 2490
EAI_SERVICE (*C macro*), 2491
EAI_SOCKTYPE (*C macro*), 2491
EAI_SYSTEM (*C macro*), 2491
EALREADY (*C macro*), 79
EBADF (*C macro*), 76
EBADMSG (*C macro*), 78
EBUSY (*C macro*), 76
ec_host_cmd (*C struct*), 836
ec_host_cmd_add_suppressed (*C function*), 835
ec_host_cmd_backend_api (*C struct*), 836
ec_host_cmd_backend_api_init (*C type*), 830
ec_host_cmd_backend_api_send (*C type*), 831
ec_host_cmd_backend_get_espi (*C function*), 833
ec_host_cmd_backend_get_shi_ite (*C function*), 833
ec_host_cmd_backend_get_shi_npcx (*C function*), 833
ec_host_cmd_backend_get_spi (*C function*), 834
ec_host_cmd_backend_get_uart (*C function*), 834
ec_host_cmd_get_hc (*C function*), 835
EC_HOST_CMD_HANDLER (*C macro*), 830
ec_host_cmd_handler (*C struct*), 837
ec_host_cmd_handler_args (*C struct*), 836
ec_host_cmd_handler_args.command (*C var*), 837
ec_host_cmd_handler_args.input_buf (*C var*), 837
ec_host_cmd_handler_args.input_buf_size (*C var*), 837
ec_host_cmd_handler_args.output_buf (*C var*), 837
ec_host_cmd_handler_args.output_buf_max (*C var*), 837
ec_host_cmd_handler_args.output_buf_size (*C var*), 837
ec_host_cmd_handler_args.reserved (*C var*), 837
ec_host_cmd_handler_args.version (*C var*), 837

ec_host_cmd_handler_cb (*C type*), 831
EC_HOST_CMD_HANDLER_UNBOUND (*C macro*), 830
ec_host_cmd_handler.handler (*C var*), 837
ec_host_cmd_handler.id (*C var*), 837
ec_host_cmd_handler.min_rqt_size (*C var*), 837
ec_host_cmd_handler.min_rsp_size (*C var*), 837
ec_host_cmd_handler.version_mask (*C var*), 837
ec_host_cmd_in_progress_cb_t (*C type*), 831
ec_host_cmd_init (*C function*), 834
ec_host_cmd_log_level (*C enum*), 833
ec_host_cmd_log_level.EC_HOST_CMD_DEBUG_EVERY (*C enumerator*), 833
ec_host_cmd_log_level.EC_HOST_CMD_DEBUG_MODES (*C enumerator*), 833
ec_host_cmd_log_level.EC_HOST_CMD_DEBUG_NORMAL (*C enumerator*), 833
ec_host_cmd_log_level.EC_HOST_CMD_DEBUG_OFF (*C enumerator*), 833
ec_host_cmd_log_level.EC_HOST_CMD_DEBUG_PARAMS (*C enumerator*), 833
ec_host_cmd_request_header (*C struct*), 838
ec_host_cmd_request_header.checksum (*C var*), 838
ec_host_cmd_request_header.cmd_id (*C var*), 838
ec_host_cmd_request_header.cmd_ver (*C var*), 838
ec_host_cmd_request_header.data_len (*C var*), 838
ec_host_cmd_request_header.prtcl_ver (*C var*), 838
ec_host_cmd_request_header.reserved (*C var*), 838
ec_host_cmd_response_header (*C struct*), 838
ec_host_cmd_response_header.checksum (*C var*), 838
ec_host_cmd_response_header.data_len (*C var*), 839
ec_host_cmd_response_header.prtcl_ver (*C var*), 838
ec_host_cmd_response_header.reserved (*C var*), 839
ec_host_cmd_response_header.result (*C var*), 839
ec_host_cmd_rx_ctx (*C struct*), 835
ec_host_cmd_rx_ctx.buf (*C var*), 835
ec_host_cmd_rx_ctx.len (*C var*), 835
ec_host_cmd_rx_ctx.len_max (*C var*), 835
ec_host_cmd_rx_notify (*C function*), 834
ec_host_cmd_send_response (*C function*), 834
ec_host_cmd_set_user_cb (*C function*), 835
ec_host_cmd_state (*C enum*), 833
ec_host_cmd_state.EC_HOST_CMD_STATE_DISABLED (*C enumerator*), 833
ec_host_cmd_state.EC_HOST_CMD_STATE_PROCESSING (*C enumerator*), 833
ec_host_cmd_state.EC_HOST_CMD_STATE_RECEIVING (*C enumerator*), 833
ec_host_cmd_state.EC_HOST_CMD_STATE_SENDING (*C enumerator*), 833
ec_host_cmd_status (*C enum*), 831
ec_host_cmd_status.EC_HOST_CMD_ACCESS_DENIED (*C enumerator*), 831
ec_host_cmd_status.EC_HOST_CMD_BUS_ERROR (*C enumerator*), 832
ec_host_cmd_status.EC_HOST_CMD_BUSY (*C enumerator*), 832
ec_host_cmd_status.EC_HOST_CMD_DUP_UNAVAILABLE (*C enumerator*), 833
ec_host_cmd_status.EC_HOST_CMD_ERROR (*C enumerator*), 831
ec_host_cmd_status.EC_HOST_CMD_IN_PROGRESS (*C enumerator*), 832
ec_host_cmd_status.EC_HOST_CMD_INVALID_CHECKSUM (*C enumerator*), 832
ec_host_cmd_status.EC_HOST_CMD_INVALID_COMMAND (*C enumerator*), 831
ec_host_cmd_status.EC_HOST_CMD_INVALID_DATA_CRC (*C enumerator*), 832
ec_host_cmd_status.EC_HOST_CMD_INVALID_HEADER (*C enumerator*), 832
ec_host_cmd_status.EC_HOST_CMD_INVALID_HEADER_CRC (*C enumerator*), 832
ec_host_cmd_status.EC_HOST_CMD_INVALID_HEADER_VERSION (*C enumerator*), 832
ec_host_cmd_status.EC_HOST_CMD_INVALID_PARAM (*C enumerator*), 831
ec_host_cmd_status.EC_HOST_CMD_INVALID_RESPONSE (*C enumerator*), 832
ec_host_cmd_status.EC_HOST_CMD_INVALID_VERSION (*C enumerator*), 832
ec_host_cmd_status.EC_HOST_CMD_MAX (*C enumerator*), 833
ec_host_cmd_status.EC_HOST_CMD_OVERFLOW (*C enumerator*), 832

ec_host_cmd_status.EC_HOST_CMD_REQUEST_TRUNCATED (*C enumerator*), 832
ec_host_cmd_status.EC_HOST_CMD_RESPONSE_TOO_BIG (*C enumerator*), 832
ec_host_cmd_status.EC_HOST_CMD_SUCCESS (*C enumerator*), 831
ec_host_cmd_status.EC_HOST_CMD_TIMEOUT (*C enumerator*), 832
ec_host_cmd_status.EC_HOST_CMD_UNAVAILABLE (*C enumerator*), 832
ec_host_cmd_task (*C function*), 835
ec_host_cmd_tx_buf (*C struct*), 836
ec_host_cmd_tx_buf.buf (*C var*), 836
ec_host_cmd_tx_buf.len (*C var*), 836
ec_host_cmd_tx_buf.len_max (*C var*), 836
ec_host_cmd_user_cb_t (*C type*), 831
ec_host_cmd.rx_ready (*C var*), 836
ec_host_cmd.rx_status (*C var*), 836
ec_host_cmd.user_cb (*C var*), 836
ECANCELED (*C macro*), 80
ECHILD (*C macro*), 76
ECONNABORTED (*C macro*), 79
ECONNREFUSED (*C macro*), 79
ECONNRESET (*C macro*), 78
edac_ecc_error_log_clear (*C function*), 3332
edac_ecc_error_log_get (*C function*), 3332
edac_error_type (*C enum*), 3334
edac_error_type.EDAC_ERROR_TYPE_DRAM_COR (*C enumerator*), 3334
edac_error_type.EDAC_ERROR_TYPE_DRAM_UC (*C enumerator*), 3334
edac_errors_cor_get (*C function*), 3333
edac_errors_uc_get (*C function*), 3333
edac_inject_error_trigger (*C function*), 3332
edac_inject_get_error_type (*C function*), 3331
edac_inject_get_param1 (*C function*), 3330
edac_inject_get_param2 (*C function*), 3331
edac_inject_set_error_type (*C function*), 3331
edac_inject_set_param1 (*C function*), 3330
edac_inject_set_param2 (*C function*), 3331
edac_notify_callback_set (*C function*), 3333
edac_parity_error_log_clear (*C function*), 3333
edac_parity_error_log_get (*C function*), 3332
EDEADLK (*C macro*), 78
EDESTADDRREQ (*C macro*), 80
EDOM (*C macro*), 77
eeprom_get_size (*C function*), 3307
eeprom_read (*C function*), 3306
eeprom_target_program (*C function*), 3387
eeprom_target_read (*C function*), 3387
eeprom_target_set_addr (*C function*), 3387
eeprom_write (*C function*), 3306
EEXIST (*C macro*), 76
EFAULT (*C macro*), 76
EFBIG (*C macro*), 77
EHOSTDOWN (*C macro*), 79
EHOSTUNREACH (*C macro*), 79
EILSEQ (*C macro*), 80
EINPROGRESS (*C macro*), 79
EINTR (*C macro*), 76
EINVAL (*C macro*), 77
EIO (*C macro*), 76
EISCONN (*C macro*), 80
EISDIR (*C macro*), 77
elf_file (*runners.core.RunnerConfig attribute*), 197

ELOOP (*C macro*), 78
EMFILE (*C macro*), 77
EMLINK (*C macro*), 77
EMPTY (*C macro*), 690
EMSGSIZE (*C macro*), 80
emul (*C struct*), 3146
emul_bus_type (*C enum*), 3145
emul_bus_type.EMUL_BUS_TYPE_ESPI (*C enumerator*), 3145
emul_bus_type.EMUL_BUS_TYPE_I2C (*C enumerator*), 3145
emul_bus_type.EMUL_BUS_TYPE_MSPI (*C enumerator*), 3145
emul_bus_type.EMUL_BUS_TYPE_NONE (*C enumerator*), 3145
emul_bus_type.EMUL_BUS_TYPE_SPI (*C enumerator*), 3145
EMUL_DT_DEFINE (*C macro*), 3143
EMUL_DT_GET (*C macro*), 3144
EMUL_DT_GET_OR_NULL (*C macro*), 3144
EMUL_DT_INST_DEFINE (*C macro*), 3144
EMUL_DT_NAME_GET (*C macro*), 3143
emul_fuel_gauge_is_battery_cutoff (*C function*), 3350
emul_fuel_gauge_set_battery_charging (*C function*), 3350
emul_get_binding (*C function*), 3145
emul_init_for_bus (*C function*), 3145
emul_init_t (*C type*), 3144
emul_link_for_bus (*C struct*), 3145
emul_list_for_bus (*C struct*), 3145
emul_list_for_bus.children (*C var*), 3146
emul_list_for_bus.num_children (*C var*), 3146
emul_sensor_backend_get_attribute_metadata (*C function*), 3605
emul_sensor_backend_get_sample_range (*C function*), 3604
emul_sensor_backend_is_supported (*C function*), 3604
emul_sensor_backend_set_attribute (*C function*), 3605
emul_sensor_backend_set_channel (*C function*), 3604
emul.backend_api (*C var*), 3146
emul.bus (*C union*), 3146
emul.bus_type (*C var*), 3146
emul.bus.espi (*C var*), 3146
emul.bus.i2c (*C var*), 3146
emul.bus.mspi (*C var*), 3147
emul.bus.none (*C var*), 3147
emul.bus.spi (*C var*), 3146
emul.cfg (*C var*), 3146
emul.data (*C var*), 3146
emul.dev (*C var*), 3146
emul.init (*C var*), 3146
ENAMETOOLONG (*C macro*), 78
energy_scan_done_cb_t (*C type*), 2682
ENETDOWN (*C macro*), 79
ENETRESET (*C macro*), 80
ENETUNREACH (*C macro*), 79
ENFILE (*C macro*), 77
ENOBUFS (*C macro*), 79
ENODATA (*C macro*), 78
ENODEV (*C macro*), 77
ENOENT (*C macro*), 75
ENOEXEC (*C macro*), 76
ENOLCK (*C macro*), 78
ENOMEM (*C macro*), 76
ENOMSG (*C macro*), 78
ENOPROTOOPT (*C macro*), 79

ENOSPC (*C macro*), 77
ENOSR (*C macro*), 78
ENOSTR (*C macro*), 78
ENOSYS (*C macro*), 78
ENOTBLK (*C macro*), 76
ENOTCONN (*C macro*), 80
ENOTDIR (*C macro*), 77
ENOTEMPTY (*C macro*), 78
ENOTSOCK (*C macro*), 79
ENOTSUP (*C macro*), 80
ENOTTY (*C macro*), 77
ensure_output() (*runners.core.ZephyrBinaryRunner method*), 200
ENTROPY_BUSYWAIT (*C macro*), 3327
entropy_driver_api (*C struct*), 3328
entropy_get_entropy (*C function*), 3328
entropy_get_entropy_isr (*C function*), 3328
entropy_get_entropy_isr_t (*C type*), 3328
entropy_get_entropy_t (*C type*), 3328
environment variable
    ARCMWDT_TOOLCHAIN_PATH, 290
    ARM_PRODUCT_DEF, 288
    ARMCLANG_TOOLCHAIN_PATH, 288
    ARMLMD_LICENSE_FILE, 288
    BOARD, 27, 186
    CONF_FILE, 27
    EXTRA_ZEPHYR_MODULES, 27
    GNUARMEMB_TOOLCHAIN_PATH, 290, 291
    METAWARE_ROOT, 290
    PATH, 7, 8, 16, 27, 28, 96, 131, 136, 137, 284
    QEMU_EXTRA_FLAGS, 48
    SHIELD, 27
    T32_DIR, 98
    {TOOLCHAIN}_TOOLCHAIN_PATH, 22, 28
    TOOLCHAIN_VER, 289
    XCC_NO_G_FLAG, 289
    XTENSA_CORE, 289
    XTENSA_TOOLCHAIN_PATH, 289
    XTOOLS_TOOLCHAIN_PATH, 292
    ZEPHYR_BASE, 27, 138140, 181, 739, 1613
    ZEPHYR_BOARD_ALIASES, 23, 28
    ZEPHYR_MODULES, 28
    ZEPHYR_SCA_VARIANT, 281
    ZEPHYR_SDK_INSTALL_DIR, 15, 20, 22, 28, 285
    ZEPHYR_TOOLCHAIN_VARIANT, 15, 22, 28, 242, 285, 288293
ENXIO (*C macro*), 76
EOPNOTSUPP (*C macro*), 78
EOVERFLOW (*C macro*), 80
EPERM (*C macro*), 75
EPFNOSUPPORT (*C macro*), 78
EPIPE (*C macro*), 77
EPROTO (*C macro*), 78
EPROTONOSUPPORT (*C macro*), 80
EPROTOTYPE (*C macro*), 79
ERANGE (*C macro*), 77
EROFS (*C macro*), 77
errno (*C macro*), 75
ESHUTDOWN (*C macro*), 79
ESOCKTNOSUPPORT (*C macro*), 80

espi_add_callback (*C function*), 3320
espi_bus_event (*C enum*), 3311
espi_bus_event.ESPI_BUS_EVENT_CHANNEL_READY (*C enumerator*), 3311
espi_bus_event.ESPI_BUS_EVENT_OOB_RECEIVED (*C enumerator*), 3311
espi_bus_event.ESPI_BUS_EVENT_VWIRE_RECEIVED (*C enumerator*), 3311
espi_bus_event.ESPI_BUS_PERIPHERAL_NOTIFICATION (*C enumerator*), 3311
espi_bus_event.ESPI_BUS_RESET (*C enumerator*), 3311
espi_bus_event.ESPI_BUS_TAF_NOTIFICATION (*C enumerator*), 3311
espi_callback_handler_t (*C type*), 3309
espi_cfg (*C struct*), 3326
espi_cfg.channel_caps (*C var*), 3326
espi_cfg.io_caps (*C var*), 3326
espi_cfg.max_freq (*C var*), 3326
espi_channel (*C enum*), 3310
espi_channel.ESPI_CHANNEL_FLASH (*C enumerator*), 3311
espi_channel.ESPI_CHANNEL_OOB (*C enumerator*), 3311
espi_channel.ESPI_CHANNEL_PERIPHERAL (*C enumerator*), 3311
espi_channel.ESPI_CHANNEL_VWIRE (*C enumerator*), 3311
espi_config (*C function*), 3315
espi_cycle_type (*C enum*), 3312
espi_cycle_type.ESPI_CYCLE_MEMORY_READ32 (*C enumerator*), 3312
espi_cycle_type.ESPI_CYCLE_MEMORY_READ64 (*C enumerator*), 3312
espi_cycle_type.ESPI_CYCLE_MEMORY_WRITE32 (*C enumerator*), 3312
espi_cycle_type.ESPI_CYCLE_MEMORY_WRITE64 (*C enumerator*), 3312
espi_cycle_type.ESPI_CYCLE_MESSAGE_DATA (*C enumerator*), 3312
espi_cycle_type.ESPI_CYCLE_MESSAGE_NODATA (*C enumerator*), 3312
espi_cycle_type.ESPI_CYCLE_NOK_COMPLETION_NODATA (*C enumerator*), 3312
espi_cycle_type.ESPI_CYCLE_OK_COMPLETION_NODATA (*C enumerator*), 3312
espi_cycle_type.ESPI_CYCLE_OKCOMPLETION_DATA (*C enumerator*), 3312
espi_event (*C struct*), 3326
espi_event.evt_data (*C var*), 3326
espi_event.evt_details (*C var*), 3326
espi_event.evt_type (*C var*), 3326
espi_evt_data_acpi (*C struct*), 3326
espi_evt_data_kbc (*C struct*), 3326
espi_flash_erase (*C function*), 3319
espi_flash_packet (*C struct*), 3327
espi_get_channel_status (*C function*), 3316
espi_init_callback (*C function*), 3319
espi_io_mode (*C enum*), 3309
espi_io_mode.ESPI_IO_MODE_DUAL_LINES (*C enumerator*), 3310
espi_io_mode.ESPI_IO_MODE_QUAD_LINES (*C enumerator*), 3310
espi_io_mode.ESPI_IO_MODE_SINGLE_LINE (*C enumerator*), 3309
espi_oob_packet (*C struct*), 3327
espi_pc_event (*C enum*), 3311
espi_pc_event.ESPI_PC_EVT_BUS_CHANNEL_READY (*C enumerator*), 3311
espi_pc_event.ESPI_PC_EVT_BUS_MASTER_ENABLE (*C enumerator*), 3311
espi_read_flash (*C function*), 3318
espi_read_lpc_request (*C function*), 3317
espi_read_request (*C function*), 3316
espi_receive_oob (*C function*), 3318
espi_receive_vwire (*C function*), 3318
espi_remove_callback (*C function*), 3321
espi_request_packet (*C struct*), 3326
espi_saf_activate (*C function*), 3322
espi_saf_add_callback (*C function*), 3325
espi_saf_cfg (*C struct*), 3327
espi_saf_config (*C function*), 3321

espi_saf_flash_erase (*C function*), 3323
espi_saf_flash_read (*C function*), 3323
espi_saf_flash_unsuccess (*C function*), 3324
espi_saf_flash_write (*C function*), 3323
espi_saf_get_channel_status (*C function*), 3323
espi_saf_init_callback (*C function*), 3324
espi_saf_packet (*C struct*), 3327
espi_saf_remove_callback (*C function*), 3325
espi_saf_set_protection_regions (*C function*), 3322
espi_send_oob (*C function*), 3318
espi_send_vwire (*C function*), 3317
espi_virtual_peripheral (*C enum*), 3312
espi_virtual_peripheral.ESPI_PERIPHERAL_8042_KBC (*C enumerator*), 3312
espi_virtual_peripheral.ESPI_PERIPHERAL_DEBUG_PORT80 (*C enumerator*), 3312
espi_virtual_peripheral.ESPI_PERIPHERAL_HOST_IO (*C enumerator*), 3312
espi_virtual_peripheral.ESPI_PERIPHERAL_HOST_IO_PVT (*C enumerator*), 3312
espi_virtual_peripheral.ESPI_PERIPHERAL_UART (*C enumerator*), 3312
espi_vwire_signal (*C enum*), 3312
ESPI_VWIRE_SIGNAL_OCB_0 (*C macro*), 3309
ESPI_VWIRE_SIGNAL_OCB_1 (*C macro*), 3309
ESPI_VWIRE_SIGNAL_OCB_2 (*C macro*), 3309
ESPI_VWIRE_SIGNAL_OCB_3 (*C macro*), 3309
espi_vwire_signal.ESPI_VWIRE_SIGNAL_COUNT (*C enumerator*), 3314
espi_vwire_signal.ESPI_VWIRE_SIGNAL_DNX_ACK (*C enumerator*), 3314
espi_vwire_signal.ESPI_VWIRE_SIGNAL_DNX_WARN (*C enumerator*), 3313
espi_vwire_signal.ESPI_VWIRE_SIGNAL_ERR_FATAL (*C enumerator*), 3313
espi_vwire_signal.ESPI_VWIRE_SIGNAL_ERR_NON_FATAL (*C enumerator*), 3313
espi_vwire_signal.ESPI_VWIRE_SIGNAL_HOST_C10 (*C enumerator*), 3313
espi_vwire_signal.ESPI_VWIRE_SIGNAL_HOST_RST_ACK (*C enumerator*), 3313
espi_vwire_signal.ESPI_VWIRE_SIGNAL_HOST_RST_WARN (*C enumerator*), 3313
espi_vwire_signal.ESPI_VWIRE_SIGNAL_NMIOUT (*C enumerator*), 3313
espi_vwire_signal.ESPI_VWIRE_SIGNAL_OOB_RST_ACK (*C enumerator*), 3313
espi_vwire_signal.ESPI_VWIRE_SIGNAL_OOB_RST_WARN (*C enumerator*), 3313
espi_vwire_signal.ESPI_VWIRE_SIGNAL_PLTRST (*C enumerator*), 3313
espi_vwire_signal.ESPI_VWIRE_SIGNAL_PME (*C enumerator*), 3313
espi_vwire_signal.ESPI_VWIRE_SIGNAL_RST_CPU_INIT (*C enumerator*), 3313
espi_vwire_signal.ESPI_VWIRE_SIGNAL_SCI (*C enumerator*), 3314
espi_vwire_signal.ESPI_VWIRE_SIGNAL_SLP_A (*C enumerator*), 3313
espi_vwire_signal.ESPI_VWIRE_SIGNAL_SLP_LAN (*C enumerator*), 3313
espi_vwire_signal.ESPI_VWIRE_SIGNAL_SLP_S3 (*C enumerator*), 3312
espi_vwire_signal.ESPI_VWIRE_SIGNAL_SLP_S4 (*C enumerator*), 3312
espi_vwire_signal.ESPI_VWIRE_SIGNAL_SLP_S5 (*C enumerator*), 3313
espi_vwire_signal.ESPI_VWIRE_SIGNAL_SLP_WLAN (*C enumerator*), 3313
espi_vwire_signal.ESPI_VWIRE_SIGNAL_SMI (*C enumerator*), 3314
espi_vwire_signal.ESPI_VWIRE_SIGNAL_SMIOUT (*C enumerator*), 3313
espi_vwire_signal.ESPI_VWIRE_SIGNAL_SUS_ACK (*C enumerator*), 3314
espi_vwire_signal.ESPI_VWIRE_SIGNAL_SUS_PWRDN_ACK (*C enumerator*), 3313
espi_vwire_signal.ESPI_VWIRE_SIGNAL_SUS_STAT (*C enumerator*), 3313
espi_vwire_signal.ESPI_VWIRE_SIGNAL_SUS_WARN (*C enumerator*), 3313
espi_vwire_signal.ESPI_VWIRE_SIGNAL_TARGET_BOOT_DONE (*C enumerator*), 3313
espi_vwire_signal.ESPI_VWIRE_SIGNAL_TARGET_BOOT_STS (*C enumerator*), 3313
espi_vwire_signal.ESPI_VWIRE_SIGNAL_TARGET_GPIO_0 (*C enumerator*), 3314
espi_vwire_signal.ESPI_VWIRE_SIGNAL_TARGET_GPIO_1 (*C enumerator*), 3314
espi_vwire_signal.ESPI_VWIRE_SIGNAL_TARGET_GPIO_2 (*C enumerator*), 3314
espi_vwire_signal.ESPI_VWIRE_SIGNAL_TARGET_GPIO_3 (*C enumerator*), 3314
espi_vwire_signal.ESPI_VWIRE_SIGNAL_TARGET_GPIO_4 (*C enumerator*), 3314
espi_vwire_signal.ESPI_VWIRE_SIGNAL_TARGET_GPIO_5 (*C enumerator*), 3314
espi_vwire_signal.ESPI_VWIRE_SIGNAL_TARGET_GPIO_6 (*C enumerator*), 3314

espi_vwire_signal.ESPI_VWIRE_SIGNAL_TARGET_GPIO_7 (*C enumerator*), 3314
espi_vwire_signal.ESPI_VWIRE_SIGNAL_TARGET_GPIO_8 (*C enumerator*), 3314
espi_vwire_signal.ESPI_VWIRE_SIGNAL_TARGET_GPIO_9 (*C enumerator*), 3314
espi_vwire_signal.ESPI_VWIRE_SIGNAL_TARGET_GPIO_10 (*C enumerator*), 3314
espi_vwire_signal.ESPI_VWIRE_SIGNAL_TARGET_GPIO_11 (*C enumerator*), 3314
espi_vwire_signal.ESPI_VWIRE_SIGNAL_WAKE (*C enumerator*), 3313
espi_write_flash (*C function*), 3319
espi_write_lpc_request (*C function*), 3317
espi_write_request (*C function*), 3316
ESPIPE (*C macro*), 77
ESRCH (*C macro*), 76
ETH_NET_DEVICE_DT_DEFINE (*C macro*), 2642
ETH_NET_DEVICE_DT_INST_DEFINE (*C macro*), 2642
ETH_NET_DEVICE_INIT (*C macro*), 2641
ETH_NET_DEVICE_INIT_INSTANCE (*C macro*), 2641
ethernet_api (*C struct*), 2653
ethernet_api.get_capabilities (*C var*), 2653
ethernet_api.get_config (*C var*), 2654
ethernet_api.iface_api (*C var*), 2653
ethernet_api.send (*C var*), 2654
ethernet_api.set_config (*C var*), 2654
ethernet_api.start (*C var*), 2653
ethernet_api.stop (*C var*), 2653
ethernet_checksum_support (*C enum*), 2644
ethernet_checksum_support.ETHERNET_CHECKSUM_SUPPORT_IPV4_HEADER (*C enumerator*), 2644
ethernet_checksum_support.ETHERNET_CHECKSUM_SUPPORT_IPV4_ICMP (*C enumerator*), 2644
ethernet_checksum_support.ETHERNET_CHECKSUM_SUPPORT_IPV6_HEADER (*C enumerator*), 2644
ethernet_checksum_support.ETHERNET_CHECKSUM_SUPPORT_IPV6_ICMP (*C enumerator*), 2644
ethernet_checksum_support.ETHERNET_CHECKSUM_SUPPORT_NONE (*C enumerator*), 2644
ethernet_checksum_support.ETHERNET_CHECKSUM_SUPPORT_TCP (*C enumerator*), 2644
ethernet_checksum_support.ETHERNET_CHECKSUM_SUPPORT_UDP (*C enumerator*), 2644
ethernet_filter (*C struct*), 2652
ethernet_filter.mac_address (*C var*), 2653
ethernet_filter.set (*C var*), 2653
ethernet_filter.type (*C var*), 2653
ethernet_hw_caps (*C enum*), 2642
ethernet_hw_caps.ETHERNET_AUTO_NEGOTIATION_SET (*C enumerator*), 2642
ethernet_hw_caps.ETHERNET_DSA_MASTER_PORT (*C enumerator*), 2643
ethernet_hw_caps.ETHERNET_DSA_SLAVE_PORT (*C enumerator*), 2643
ethernet_hw_caps.ETHERNET_DUPLEX_SET (*C enumerator*), 2643
ethernet_hw_caps.ETHERNET_HW_FILTERING (*C enumerator*), 2643
ethernet_hw_caps.ETHERNET_HW_RX_CHKSUM_OFFLOAD (*C enumerator*), 2642
ethernet_hw_caps.ETHERNET_HW_TX_CHKSUM_OFFLOAD (*C enumerator*), 2642
ethernet_hw_caps.ETHERNET_HW_VLAN (*C enumerator*), 2642
ethernet_hw_caps.ETHERNET_HW_VLAN_TAG_STRIP (*C enumerator*), 2643
ethernet_hw_caps.ETHERNET_LINK_10BASE_T (*C enumerator*), 2642
ethernet_hw_caps.ETHERNET_LINK_100BASE_T (*C enumerator*), 2643
ethernet_hw_caps.ETHERNET_LINK_1000BASE_T (*C enumerator*), 2643
ethernet_hw_caps.ETHERNET_LLDP (*C enumerator*), 2643
ethernet_hw_caps.ETHERNET_PRIORITY_QUEUES (*C enumerator*), 2643
ethernet_hw_caps.ETHERNET_PROMISC_MODE (*C enumerator*), 2643
ethernet_hw_caps.ETHERNET_PTP (*C enumerator*), 2643
ethernet_hw_caps.ETHERNET_QAV (*C enumerator*), 2643
ethernet_hw_caps.ETHERNET_QBU (*C enumerator*), 2643
ethernet_hw_caps.ETHERNET_QBV (*C enumerator*), 2643
ethernet_hw_caps.ETHERNET_TXINJECTION_MODE (*C enumerator*), 2643
ethernet_hw_caps.ETHERNET_TXTIME (*C enumerator*), 2643
ethernet_if_types (*C enum*), 2644

ethernet_if_types.L2_ETH_IF_TYPE_ETHERNET (*C enumerator*), 2644
ethernet_if_types.L2_ETH_IF_TYPE_WIFI (*C enumerator*), 2644
ethernet_lldp (*C struct*), 2654
ethernet_lldp.cb (*C var*), 2654
ethernet_lldp.iface (*C var*), 2654
ethernet_lldp.lldpdu (*C var*), 2654
ethernet_lldp.node (*C var*), 2654
ethernet_lldp.optional_du (*C var*), 2654
ethernet_lldp.optional_len (*C var*), 2654
ethernet_lldp.tx_timer_start (*C var*), 2654
ethernet_lldp.tx_timer_timeout (*C var*), 2654
ethernet_mgmt_raise_carrier_off_event (*C function*), 2945
ethernet_mgmt_raise_carrier_on_event (*C function*), 2945
ethernet_mgmt_raise_vlan_disabled_event (*C function*), 2945
ethernet_mgmt_raise_vlan_enabled_event (*C function*), 2945
ethernet_qav_param (*C struct*), 2650
ethernet_qav_param.delta_bandwidth (*C var*), 2651
ethernet_qav_param.enabled (*C var*), 2651
ethernet_qav_param.idle_slope (*C var*), 2651
ethernet_qav_param.oper_idle_slope (*C var*), 2651
ethernet_qav_param.queue_id (*C var*), 2650
ethernet_qav_param.traffic_class (*C var*), 2651
ethernet_qav_param.type (*C var*), 2651
ethernet_qbu_param (*C struct*), 2652
ethernet_qbu_param.additional_fragment_size (*C var*), 2652
ethernet_qbu_param.enabled (*C var*), 2652
ethernet_qbu_param.frame_preempt_statuses (*C var*), 2652
ethernet_qbu_param.hold_advance (*C var*), 2652
ethernet_qbu_param.link_partner_status (*C var*), 2652
ethernet_qbu_param.port_id (*C var*), 2652
ethernet_qbu_param.release_advance (*C var*), 2652
ethernet_qbu_param.type (*C var*), 2652
ethernet_qbv_param (*C struct*), 2651
ethernet_qbv_param.base_time (*C var*), 2652
ethernet_qbv_param.cycle_time (*C var*), 2652
ethernet_qbv_param.enabled (*C var*), 2651
ethernet_qbv_param.extension_time (*C var*), 2652
ethernet_qbv_param.gate_control (*C var*), 2652
ethernet_qbv_param.gate_control_list_len (*C var*), 2652
ethernet_qbv_param.gate_status (*C var*), 2651
ethernet_qbv_param.operation (*C var*), 2651
ethernet_qbv_param.port_id (*C var*), 2651
ethernet_qbv_param.row (*C var*), 2651
ethernet_qbv_param.state (*C var*), 2651
ethernet_qbv_param.time_interval (*C var*), 2651
ethernet_qbv_param.type (*C var*), 2651
ethernet_t1s_param (*C struct*), 2649
ethernet_t1s_param.burst_count (*C var*), 2650
ethernet_t1s_param.burst_timer (*C var*), 2650
ethernet_t1s_param.enable (*C var*), 2650
ethernet_t1s_param.node_count (*C var*), 2650
ethernet_t1s_param.node_id (*C var*), 2650
ethernet_t1s_param.plca (*C var*), 2650
ethernet_t1s_param.to_timer (*C var*), 2650
ethernet_t1s_param.type (*C var*), 2650
ethernet_txtime_param (*C struct*), 2653
ethernet_txtime_param.enable_txtime (*C var*), 2653
ethernet_txtime_param.queue_id (*C var*), 2653

ethernet_txtime_param.type (*C var*), 2653
ETIME (*C macro*), 78
ETIMEDOUT (*C macro*), 79
ETOOMANYREFS (*C macro*), 80
ETXTBSY (*C macro*), 77
EWOULDBLOCK (*C macro*), 80
EXDEV (*C macro*), 76
exe_file (*runners.core.RunnerConfig attribute*), 197
exec_command() (*twister_harness.Shell method*), 268
EXPORT_SYMBOL (*C macro*), 928
extload_help() (*runners.core.ZephyrBinaryRunner class method*), 200

F

fcb (*C struct*), 1196
fcb_append (*C function*), 1198
fcb_append_finish (*C function*), 1198
fcb_append_to_scratch (*C function*), 1199
fcb_clear (*C function*), 1200
fcb_entry (*C struct*), 1195
fcb_entry_ctx (*C struct*), 1196
fcb_entry_ctx.fap (*C var*), 1196
fcb_entry_ctx.loc (*C var*), 1196
FCB_ENTRY_FA_DATA_OFF (*C macro*), 1195
fcb_entry.fe_data_len (*C var*), 1196
fcb_entry.fe_data_off (*C var*), 1196
fcb_entry.fe_elem_off (*C var*), 1196
fcb_entry.fe_sector (*C var*), 1196
FCB_FLAGS_CRC_DISABLED (*C macro*), 1195
fcb_free_sector_cnt (*C function*), 1199
fcb_getnext (*C function*), 1199
fcb_init (*C function*), 1198
fcb_is_empty (*C function*), 1199
FCB_MAX_LEN (*C macro*), 1195
fcb_offset_last_n (*C function*), 1200
fcb_rotate (*C function*), 1199
fcb_walk (*C function*), 1198
fcb_walk_cb (*C type*), 1197
fcb.f_active (*C var*), 1197
fcb.f_active_id (*C var*), 1197
fcb.f_align (*C var*), 1197
fcb.f_erase_value (*C var*), 1197
fcb.f_magic (*C var*), 1196
fcb.f_mtx (*C var*), 1197
fcb.f_oldest (*C var*), 1197
fcb.f_scratch_cnt (*C var*), 1197
fcb.f_sector_cnt (*C var*), 1197
fcb.f_sectors (*C var*), 1197
fcb.f_version (*C var*), 1196
fcb.fap (*C var*), 1197
FIELD_GET (*C macro*), 680
FIELD_PREP (*C macro*), 680
file (*runners.core.RunnerConfig attribute*), 197
file_type (*runners.core.RunnerConfig attribute*), 197
FileType (*class in runners.core*), 196
FIXED_5V_100MA_RDO (*C macro*), 3087
FIXED_PARTITION_DEVICE (*C macro*), 1190
FIXED_PARTITION_EXISTS (*C macro*), 1189
FIXED_PARTITION_ID (*C macro*), 1189

FIXED_PARTITION_NODE_DEVICE (*C macro*), 1190
FIXED_PARTITION_NODE_OFFSET (*C macro*), 1189
FIXED_PARTITION_NODE_SIZE (*C macro*), 1190
FIXED_PARTITION_OFFSET (*C macro*), 1189
FIXED_PARTITION_SIZE (*C macro*), 1190
flash_address_from_build_conf() (*runners.core.ZephyrBinaryRunner static method*), 200
flash_api_erase (*C type*), 3341
flash_api_ex_op (*C type*), 3342
flash_api_get_parameters (*C type*), 3341
flash_api_pages_layout (*C type*), 3341
flash_api_read (*C type*), 3341
flash_api_read_jedec_id (*C type*), 3342
flash_api_sfdp_read (*C type*), 3342
flash_api_write (*C type*), 3341
flash_area (*C struct*), 1193
flash_area_align (*C function*), 1192
flash_area_cb_t (*C type*), 1190
flash_area_close (*C function*), 1191
FLASH_AREA_DEVICE (*C macro*), 1190
flash_area_erase (*C function*), 1192
flash_area_erased_val (*C function*), 1193
flash_area_flatten (*C function*), 1192
flash_area_foreach (*C function*), 1193
flash_area_get_device (*C function*), 1193
flash_area_get_sectors (*C function*), 1192
flash_area_has_driver (*C function*), 1193
flash_area_open (*C function*), 1191
flash_area_read (*C function*), 1191
flash_area_write (*C function*), 1191
flash_area.fa_dev (*C var*), 1194
flash_area.fa_id (*C var*), 1193
flash_area.fa_off (*C var*), 1194
flash_area.fa_size (*C var*), 1194
flash_driver_api (*C struct*), 3342
flash_erase (*C function*), 3336
FLASH_ERASE_C_EXPLICIT (*C macro*), 3335
FLASH_ERASE_C_SUPPORTED (*C macro*), 3335
FLASH_ERASE_C_VAL_BIT (*C macro*), 3335
FLASH_ERASE_CAPS_UNSET (*C macro*), 3335
FLASH_ERASE_UNIFORM_PAGE (*C macro*), 3335
flash_ex_op (*C function*), 3340
FLASH_EX_OP_IS_VENDOR (*C macro*), 3335
flash_ex_op_types (*C enum*), 3335
flash_ex_op_types.FLASH_EX_OP_RESET (*C enumerator*), 3336
FLASH_EX_OP_VENDOR_BASE (*C macro*), 3335
flash_fill (*C function*), 3337
flash_flatten (*C function*), 3337
flash_get_page_count (*C function*), 3338
flash_get_page_info_by_idx (*C function*), 3338
flash_get_page_info_by_offs (*C function*), 3338
flash_get_parameters (*C function*), 3339
flash_get_write_block_size (*C function*), 3339
flash_img_buffered_write (*C function*), 820
flash_img_bytes_written (*C function*), 820
flash_img_check (*C function*), 820
flash_img_check (*C struct*), 821
flash_img_check.clen (*C var*), 821
flash_img_context (*C struct*), 821

flash_img_init (*C function*), 820
flash_img_init_id (*C function*), 820
flash_page_cb (*C type*), 3335
flash_page_foreach (*C function*), 3338
flash_pages_info (*C struct*), 3340
flash_pages_layout (*C struct*), 3342
flash_parameters (*C struct*), 3340
flash_parameters.erase_value (*C var*), 3340
flash_parameters.write_block_size (*C var*), 3340
flash_params_get_erase_cap (*C function*), 3336
flash_read (*C function*), 3336
flash_read_jedec_id (*C function*), 3339
flash_sector (*C struct*), 1194
flash_sector.fs_off (*C var*), 1194
flash_sector.fs_size (*C var*), 1194
flash_sfdp_read (*C function*), 3339
flash_write (*C function*), 3336
float16_t (*C type*), 841
float32_t (*C type*), 841
float64_t (*C type*), 841
FONT_ENTRY_DEFINE (*C macro*), 3302
FOR_EACH (*C macro*), 692
FOR_EACH_FIXED_ARG (*C macro*), 693
FOR_EACH_IDX (*C macro*), 693
FOR_EACH_IDX_FIXED_ARG (*C macro*), 694
FOR_EACH_NONEMPTY_TERM (*C macro*), 692
fprintfcb (*C function*), 870
freeaddrinfo (*C function*), 2489
fs_close (*C function*), 847
fs_closedir (*C function*), 851
fs_dir_entry_type (*C enum*), 845
fs_dir_entry_type.FS_DIR_ENTRY_DIR (*C enumerator*), 845
fs_dir_entry_type.FS_DIR_ENTRY_FILE (*C enumerator*), 845
fs_dir_t (*C struct*), 855
fs_dir_t_init (*C function*), 846
fs_dir_t.dirp (*C var*), 856
fs_dir_t.mp (*C var*), 856
fs_dirent (*C struct*), 854
fs_dirent.name (*C var*), 855
fs_dirent.size (*C var*), 855
fs_dirent.type (*C var*), 855
fs_file_system_t (*C struct*), 856
fs_file_system_t.close (*C var*), 857
fs_file_system_t.closedir (*C var*), 858
fs_file_system_t.lseek (*C var*), 856
fs_file_system_t.mkdir (*C var*), 858
fs_file_system_t.mkfs (*C var*), 859
fs_file_system_t.mount (*C var*), 858
fs_file_system_t.open (*C var*), 856
fs_file_system_t.opendir (*C var*), 857
fs_file_system_t.read (*C var*), 856
fs_file_system_t.readdir (*C var*), 857
fs_file_system_t.rename (*C var*), 858
fs_file_system_t.stat (*C var*), 858
fs_file_system_t.statvfs (*C var*), 859
fs_file_system_t.sync (*C var*), 857
fs_file_system_t.tell (*C var*), 857
fs_file_system_t.truncate (*C var*), 857

fs_file_system_t.unlink (*C var*), 858
fs_file_system_t.unmount (*C var*), 858
fs_file_system_t.write (*C var*), 856
fs_file_t (*C struct*), 855
fs_file_t_init (*C function*), 845
fs_file_t.filep (*C var*), 855
fs_file_t.flags (*C var*), 855
fs_file_t.mp (*C var*), 855
FS_FSTAB_DECLARE_ENTRY (*C macro*), 845
FS_FSTAB_ENTRY (*C macro*), 844
fs_mgmt_group_events (*C enum*), 770
fs_mgmt_group_events.MGMT_EVT_OP_FS_MGMT_ALL (*C enumerator*), 770
fs_mgmt_group_events.MGMT_EVT_OP_FS_MGMT_FILE_ACCESS (*C enumerator*), 770
fs_mkdir (*C function*), 850
fs_mkfs (*C function*), 853
fs_mount (*C function*), 851
FS_MOUNT_FLAG_AUTOMOUNT (*C macro*), 844
FS_MOUNT_FLAG_NO_FORMAT (*C macro*), 844
FS_MOUNT_FLAG_READ_ONLY (*C macro*), 844
FS_MOUNT_FLAG_USE_DISK_ACCESS (*C macro*), 844
fs_mount_t (*C struct*), 854
fs_mount_t.flags (*C var*), 854
fs_mount_t.fs (*C var*), 854
fs_mount_t.fs_data (*C var*), 854
fs_mount_t.mnt_point (*C var*), 854
fs_mount_t.mountp_len (*C var*), 854
fs_mount_t.node (*C var*), 854
fs_mount_t.storage_dev (*C var*), 854
fs_mount_t.type (*C var*), 854
FS_O_APPEND (*C macro*), 843
FS_O_CREATE (*C macro*), 843
FS_O_FLAGS_MASK (*C macro*), 843
FS_O_MASK (*C macro*), 844
FS_O_MODE_MASK (*C macro*), 843
FS_O_RDWR (*C macro*), 843
FS_O_READ (*C macro*), 843
FS_O_TRUNC (*C macro*), 843
FS_O_WRITE (*C macro*), 843
fs_open (*C function*), 846
fs_opendir (*C function*), 850
fs_read (*C function*), 848
fs_readdir (*C function*), 851
fs_readmount (*C function*), 852
fs_register (*C function*), 853
fs_rename (*C function*), 847
fs_seek (*C function*), 848
FS_SEEK_CUR (*C macro*), 844
FS_SEEK_END (*C macro*), 844
FS_SEEK_SET (*C macro*), 844
fs_stat (*C function*), 852
fs_statvfs (*C function*), 853
fs_statvfs (*C struct*), 855
fs_statvfs.f_bfree (*C var*), 855
fs_statvfs.f_blocks (*C var*), 855
fs_statvfs.f_bsize (*C var*), 855
fs_statvfs.f_frsize (*C var*), 855
fs_sync (*C function*), 849
fs_tell (*C function*), 849

fs_truncate (*C function*), 849
fs_unlink (*C function*), 847
fs_unmount (*C function*), 852
fs_unregister (*C function*), 854
fs_write (*C function*), 848
FSTAB_ENTRY_DT_MOUNT_FLAGS (*C macro*), 844
fuel_gauge_battery_cutoff (*C function*), 3347
fuel_gauge_battery_cutoff_t (*C type*), 3343
fuel_gauge_driver_api (*C struct*), 3349
fuel_gauge_driver_api.get_property (*C var*), 3349
fuel_gauge_get_buffer_prop (*C function*), 3347
fuel_gauge_get_buffer_property_t (*C type*), 3343
fuel_gauge_get_prop (*C function*), 3346
fuel_gauge_get_property_t (*C type*), 3343
fuel_gauge_get_props (*C function*), 3346
fuel_gauge_prop_t (*C type*), 3343
fuel_gauge_prop_type (*C enum*), 3344
fuel_gauge_prop_type.FUEL_GAUGE_ABSOLUTE_STATE_OF_CHARGE (*C enumerator*), 3344
fuel_gauge_prop_type.FUEL_GAUGE_AVG_CURRENT (*C enumerator*), 3344
fuel_gauge_prop_type.FUEL_GAUGE_BATTERY_CUTOFF (*C enumerator*), 3344
fuel_gauge_prop_type.FUEL_GAUGE_CHARGE_CURRENT (*C enumerator*), 3345
fuel_gauge_prop_type.FUEL_GAUGE_CHARGE_CUTOFF (*C enumerator*), 3344
fuel_gauge_prop_type.FUEL_GAUGE_CHARGE_VOLTAGE (*C enumerator*), 3345
fuel_gauge_prop_type.FUEL_GAUGE_COMMON_COUNT (*C enumerator*), 3346
fuel_gauge_prop_type.FUEL_GAUGE_CONNECT_STATE (*C enumerator*), 3344
fuel_gauge_prop_type.FUEL_GAUGE_CURRENT (*C enumerator*), 3344
fuel_gauge_prop_type.FUEL_GAUGE_CUSTOM_BEGIN (*C enumerator*), 3346
fuel_gauge_prop_type.FUEL_GAUGE_CYCLE_COUNT (*C enumerator*), 3344
fuel_gauge_prop_type.FUEL_GAUGE_DESIGN_CAPACITY (*C enumerator*), 3345
fuel_gauge_prop_type.FUEL_GAUGE_DESIGN_VOLTAGE (*C enumerator*), 3345
fuel_gauge_prop_type.FUEL_GAUGE_DEVICE_CHEMISTRY (*C enumerator*), 3346
fuel_gauge_prop_type.FUEL_GAUGE_DEVICE_NAME (*C enumerator*), 3346
fuel_gauge_prop_type.FUEL_GAUGE_FLAGS (*C enumerator*), 3344
fuel_gauge_prop_type.FUEL_GAUGE_FULL_CHARGE_CAPACITY (*C enumerator*), 3344
fuel_gauge_prop_type.FUEL_GAUGE_MANUFACTURER_NAME (*C enumerator*), 3345
fuel_gauge_prop_type.FUEL_GAUGE_PRESENT_STATE (*C enumerator*), 3344
fuel_gauge_prop_type.FUEL_GAUGE_PROP_MAX (*C enumerator*), 3346
fuel_gauge_prop_type.FUEL_GAUGE_RELATIVE_STATE_OF_CHARGE (*C enumerator*), 3345
fuel_gauge_prop_type.FUEL_GAUGE_REMAINING_CAPACITY (*C enumerator*), 3344
fuel_gauge_prop_type.FUEL_GAUGE_RUNTIME_TO_EMPTY (*C enumerator*), 3344
fuel_gauge_prop_type.FUEL_GAUGE_RUNTIME_TO_FULL (*C enumerator*), 3344
fuel_gauge_prop_type.FUEL_GAUGE_SBS_ATRATE (*C enumerator*), 3345
fuel_gauge_prop_type.FUEL_GAUGE_SBS_ATRATE_OK (*C enumerator*), 3345
fuel_gauge_prop_type.FUEL_GAUGE_SBS_ATRATE_TIME_TO_EMPTY (*C enumerator*), 3345
fuel_gauge_prop_type.FUEL_GAUGE_SBS_ATRATE_TIME_TO_FULL (*C enumerator*), 3345
fuel_gauge_prop_type.FUEL_GAUGE_SBS_MFR_ACCESS (*C enumerator*), 3344
fuel_gauge_prop_type.FUEL_GAUGE_SBS_MODE (*C enumerator*), 3345
fuel_gauge_prop_type.FUEL_GAUGE_SBS_REMAINING_CAPACITY_ALARM (*C enumerator*), 3345
fuel_gauge_prop_type.FUEL_GAUGE_SBS_REMAINING_TIME_ALARM (*C enumerator*), 3345
fuel_gauge_prop_type.FUEL_GAUGE_STATUS (*C enumerator*), 3345
fuel_gauge_prop_type.FUEL_GAUGE_TEMPERATURE (*C enumerator*), 3345
fuel_gauge_prop_type.FUEL_GAUGE_VOLTAGE (*C enumerator*), 3345
fuel_gauge_prop_val (*C union*), 3347
fuel_gauge_prop_val.absolute_state_of_charge (*C var*), 3348
fuel_gauge_prop_val.avg_current (*C var*), 3348
fuel_gauge_prop_val.chg_current (*C var*), 3348
fuel_gauge_prop_val.chg_voltage (*C var*), 3348
fuel_gauge_prop_val.current (*C var*), 3348

fuel_gauge_prop_val.cutoff (*C var*), 3348
fuel_gauge_prop_val.cycle_count (*C var*), 3348
fuel_gauge_prop_val.design_cap (*C var*), 3349
fuel_gauge_prop_val.design_volt (*C var*), 3349
fuel_gauge_prop_val.fg_status (*C var*), 3349
fuel_gauge_prop_val.flags (*C var*), 3348
fuel_gauge_prop_val.full_charge_capacity (*C var*), 3348
fuel_gauge_prop_val.relative_state_of_charge (*C var*), 3348
fuel_gauge_prop_val.remaining_capacity (*C var*), 3348
fuel_gauge_prop_val.runtime_to_empty (*C var*), 3348
fuel_gauge_prop_val.runtime_to_full (*C var*), 3348
fuel_gauge_prop_val.sbs_at_rate (*C var*), 3349
fuel_gauge_prop_val.sbs_at_rate_ok (*C var*), 3349
fuel_gauge_prop_val.sbs_at_rate_time_to_empty (*C var*), 3349
fuel_gauge_prop_val.sbs_at_rate_time_to_full (*C var*), 3349
fuel_gauge_prop_val.sbs_mfr_access_word (*C var*), 3348
fuel_gauge_prop_val.sbs_mode (*C var*), 3348
fuel_gauge_prop_val.sbs_remaining_capacity_alarm (*C var*), 3349
fuel_gauge_prop_val.sbs_remaining_time_alarm (*C var*), 3349
fuel_gauge_prop_val.temperature (*C var*), 3348
fuel_gauge_prop_val.voltage (*C var*), 3348
fuel_gauge_set_prop (*C function*), 3346
fuel_gauge_set_property_t (*C type*), 3343
fuel_gauge_set_props (*C function*), 3347

G

gai_strerror (*C function*), 2489
GB (*C macro*), 685
gcm_op_t (*C type*), 721
gcm_params (*C struct*), 724
gdb (*runners.core.RunnerConfig attribute*), 197
GENMASK (*C macro*), 679
GENMASK64 (*C macro*), 679
GET_ARG_N (*C macro*), 690
GET_ARGS_LESS_N (*C macro*), 690
get_flash_address() (*runners.core.ZephyrBinaryRunner static method*), 200
get_runners() (*runners.core.ZephyrBinaryRunner static method*), 200
get_unused_ports() (*runners.core.NetworkPortHelper method*), 196
getaddrinfo (*C function*), 2489
getboolean() (*runners.core.BuildConfiguration method*), 196
gethostname (*C function*), 2489
getnameinfo (*C function*), 2489
getpeername (*C function*), 2489
getsockname (*C function*), 2489
getsockopt (*C function*), 2489
glcd_clear (*C function*), 3298
glcd_color_select (*C function*), 3299
glcd_color_set (*C function*), 3299
glcd_cursor_pos_set (*C function*), 3298
glcd_display_state_get (*C function*), 3298
glcd_display_state_set (*C function*), 3298
GLCD_DS_BLINK_OFF (*C macro*), 3297
GLCD_DS_BLINK_ON (*C macro*), 3297
GLCD_DS_CURSOR_OFF (*C macro*), 3297
GLCD_DS_CURSOR_ON (*C macro*), 3297
GLCD_DS_DISPLAY_OFF (*C macro*), 3297
GLCD_DS_DISPLAY_ON (*C macro*), 3297
GLCD_FS_8BIT_MODE (*C macro*), 3297

GLCD_FS_DOT_SIZE_BIG (*C macro*), 3297
GLCD_FS_DOT_SIZE_LITTLE (*C macro*), 3297
GLCD_FS_ROWS_1 (*C macro*), 3297
GLCD_FS_ROWS_2 (*C macro*), 3297
glcd_function_get (*C function*), 3299
glcd_function_set (*C function*), 3299
glcd_input_state_get (*C function*), 3298
glcd_input_state_set (*C function*), 3298
GLCD_IS_ENTRY_LEFT (*C macro*), 3297
GLCD_IS_ENTRY_RIGHT (*C macro*), 3297
GLCD_IS_SHIFT_DECREMENT (*C macro*), 3297
GLCD_IS_SHIFT_INCREMENT (*C macro*), 3297
glcd_print (*C function*), 3298
gnss_data (*C struct*), 3359
gnss_data_callback (*C struct*), 3360
GNSS_DATA_CALLBACK_DEFINE (*C macro*), 3353
gnss_data_callback_t (*C type*), 3354
gnss_data_callback.callback (*C var*), 3360
gnss_data_callback.dev (*C var*), 3360
gnss_data.info (*C var*), 3359
gnss_data.nav_data (*C var*), 3359
gnss_data.utc (*C var*), 3359
gnss_driver_api (*C struct*), 3359
gnss_fix_quality (*C enum*), 3355
gnss_fix_quality.GNSS_FIX_QUALITY_DGNSS (*C enumerator*), 3356
gnss_fix_quality.GNSS_FIX_QUALITY_ESTIMATED (*C enumerator*), 3356
gnss_fix_quality.GNSS_FIX_QUALITY_FLOAT_RTK (*C enumerator*), 3356
gnss_fix_quality.GNSS_FIX_QUALITY_GNSS_PPS (*C enumerator*), 3356
gnss_fix_quality.GNSS_FIX_QUALITY_GNSS_SPS (*C enumerator*), 3356
gnss_fix_quality.GNSS_FIX_QUALITY_INVALID (*C enumerator*), 3355
gnss_fix_quality.GNSS_FIX_QUALITY_RTK (*C enumerator*), 3356
gnss_fix_status (*C enum*), 3355
gnss_fix_status.GNSS_FIX_STATUS_DGNSS_FIX (*C enumerator*), 3355
gnss_fix_status.GNSS_FIX_STATUS_ESTIMATED_FIX (*C enumerator*), 3355
gnss_fix_status.GNSS_FIX_STATUS_GNSS_FIX (*C enumerator*), 3355
gnss_fix_status.GNSS_FIX_STATUS_NO_FIX (*C enumerator*), 3355
gnss_get_enabled_systems (*C function*), 3358
gnss_get_enabled_systems_t (*C type*), 3353
gnss_get_fix_rate (*C function*), 3356
gnss_get_fix_rate_t (*C type*), 3353
gnss_get_navigation_mode (*C function*), 3357
gnss_get_navigation_mode_t (*C type*), 3353
gnss_get_periodic_config (*C function*), 3357
gnss_get_periodic_config_t (*C type*), 3353
gnss_get_supported_systems (*C function*), 3358
gnss_get_supported_systems_t (*C type*), 3353
gnss_info (*C struct*), 3358
gnss_info.fix_quality (*C var*), 3359
gnss_info.fix_status (*C var*), 3359
gnss_info.hdop (*C var*), 3358
gnss_info.satellites_cnt (*C var*), 3358
gnss_navigation_mode (*C enum*), 3354
gnss_navigation_mode.GNSS_NAVIGATION_MODE_BALANCED_DYNAMICS (*C enumerator*), 3354
gnss_navigation_mode.GNSS_NAVIGATION_MODE_HIGH_DYNAMICS (*C enumerator*), 3354
gnss_navigation_mode.GNSS_NAVIGATION_MODE_LOW_DYNAMICS (*C enumerator*), 3354
gnss_navigation_mode.GNSS_NAVIGATION_MODE_ZERO_DYNAMICS (*C enumerator*), 3354
gnss_periodic_config (*C struct*), 3358
gnss_periodic_config.active_time_ms (*C var*), 3358

gnss_periodic_config.inactive_time_ms (*C var*), 3358
gnss_pps_mode (*C enum*), 3354
gnss_pps_mode.GNSS_PPS_MODE_DISABLED (*C enumerator*), 3354
gnss_pps_mode.GNSS_PPS_MODE_ENABLED (*C enumerator*), 3354
gnss_pps_mode.GNSS_PPS_MODE_ENABLED_AFTER_LOCK (*C enumerator*), 3354
gnss_pps_mode.GNSS_PPS_MODE_ENABLED_WHILE_LOCKED (*C enumerator*), 3354
gnss_satellite (*C struct*), 3360
gnss_satellite.azimuth (*C var*), 3360
gnss_satellite.elevation (*C var*), 3360
gnss_satellite.is_tracked (*C var*), 3360
gnss_satellite.prn (*C var*), 3360
gnss_satellites_callback (*C struct*), 3360
GNSS_SATELLITES_CALLBACK_DEFINE (*C macro*), 3353
gnss_satellites_callback_t (*C type*), 3354
gnss_satellites_callback.callback (*C var*), 3360
gnss_satellites_callback.dev (*C var*), 3360
gnss_satellite.snr (*C var*), 3360
gnss_satellite.system (*C var*), 3360
gnss_set_enabled_systems (*C function*), 3357
gnss_set_enabled_systems_t (*C type*), 3353
gnss_set_fix_rate (*C function*), 3356
gnss_set_fix_rate_t (*C type*), 3353
gnss_set_navigation_mode (*C function*), 3357
gnss_set_navigation_mode_t (*C type*), 3353
gnss_set_periodic_config (*C function*), 3356
gnss_set_periodic_config_t (*C type*), 3353
gnss_system (*C enum*), 3354
gnss_system.GNSS_SYSTEM_BEIDOU (*C enumerator*), 3355
gnss_system.GNSS_SYSTEM_GALILEO (*C enumerator*), 3355
gnss_system.GNSS_SYSTEM_GLONASS (*C enumerator*), 3355
gnss_system.GNSS_SYSTEM_GPS (*C enumerator*), 3355
gnss_system.GNSS_SYSTEM_IMES (*C enumerator*), 3355
gnss_system.GNSS_SYSTEM_IRNSS (*C enumerator*), 3355
gnss_system.GNSS_SYSTEM_QZSS (*C enumerator*), 3355
gnss_system.GNSS_SYSTEM_SBAS (*C enumerator*), 3355
gnss_systems_t (*C type*), 3353
gnss_time (*C struct*), 3359
gnss_time.century_year (*C var*), 3359
gnss_time.hour (*C var*), 3359
gnss_time.millisecond (*C var*), 3359
gnss_time.minute (*C var*), 3359
gnss_time.month (*C var*), 3359
gnss_time.month_day (*C var*), 3359
GNUARMEMB_TOOLCHAIN_PATH, 290, 291
GPIO_ACTIVE_HIGH (*C macro*), 3363
GPIO_ACTIVE_LOW (*C macro*), 3363
gpio_add_callback (*C function*), 3380
gpio_add_callback_dt (*C function*), 3381
gpio_callback (*C struct*), 3383
gpio_callback_handler_t (*C type*), 3371
gpio_callback.handler (*C var*), 3383
gpio_callback.node (*C var*), 3383
gpio_callback.pin_mask (*C var*), 3383
GPIO_DISCONNECTED (*C macro*), 3361
gpio_driver_config (*C struct*), 3383
gpio_driver_config.port_pin_mask (*C var*), 3383
gpio_driver_data (*C struct*), 3383
gpio_driver_data.invert (*C var*), 3383

GPIO_DT_FLAGS_MASK (*C macro*), 3370
gpio_dt_flags_t (*C type*), 3370
GPIO_DT_INST_PORT_PIN_MASK_NGPIOS_EXC (*C macro*), 3369
GPIO_DT_INST_RESERVED_RANGES (*C macro*), 3368
GPIO_DT_INST_RESERVED_RANGES_NGPIOS (*C macro*), 3368
GPIO_DT_PORT_PIN_MASK_NGPIOS_EXC (*C macro*), 3369
GPIO_DT_RESERVED_RANGES (*C macro*), 3368
GPIO_DT_RESERVED_RANGES_NGPIOS (*C macro*), 3367
gpio_dt_spec (*C struct*), 3382
GPIO_DT_SPEC_GET (*C macro*), 3365
GPIO_DT_SPEC_GET_BY_IDX (*C macro*), 3364
GPIO_DT_SPEC_GET_BY_IDX_OR (*C macro*), 3364
GPIO_DT_SPEC_GET_OR (*C macro*), 3365
GPIO_DT_SPEC_INST_GET (*C macro*), 3366
GPIO_DT_SPEC_INST_GET_BY_IDX (*C macro*), 3365
GPIO_DT_SPEC_INST_GET_BY_IDX_OR (*C macro*), 3366
GPIO_DT_SPEC_INST_GET_OR (*C macro*), 3366
gpio_dt_spec.dt_flags (*C var*), 3382
gpio_dt_spec.pin (*C var*), 3382
gpio_dt_spec.port (*C var*), 3382
gpio_flags_t (*C type*), 3370
gpio_get_pending_int (*C function*), 3382
gpio_init_callback (*C function*), 3380
GPIO_INPUT (*C macro*), 3361
GPIO_INT_DISABLE (*C macro*), 3362
GPIO_INT_EDGE_BOTH (*C macro*), 3362
GPIO_INT_EDGE_FALLING (*C macro*), 3362
GPIO_INT_EDGE_RISING (*C macro*), 3362
GPIO_INT_EDGE_TO_ACTIVE (*C macro*), 3362
GPIO_INT_EDGE_TO_INACTIVE (*C macro*), 3362
GPIO_INT_LEVEL_ACTIVE (*C macro*), 3363
GPIO_INT_LEVEL_HIGH (*C macro*), 3362
GPIO_INT_LEVEL_INACTIVE (*C macro*), 3363
GPIO_INT_LEVEL_LOW (*C macro*), 3362
GPIO_INT_WAKEUP (*C macro*), 3370
gpio_is_ready_dt (*C function*), 3371
GPIO_MAX_PINS_PER_PORT (*C macro*), 3370
GPIO_OPEN_DRAIN (*C macro*), 3363
GPIO_OPEN_SOURCE (*C macro*), 3363
GPIO_OUTPUT (*C macro*), 3361
GPIO_OUTPUT_ACTIVE (*C macro*), 3362
GPIO_OUTPUT_HIGH (*C macro*), 3362
GPIO_OUTPUT_INACTIVE (*C macro*), 3362
GPIO_OUTPUT_LOW (*C macro*), 3362
gpio_pin_configure (*C function*), 3372
gpio_pin_configure_dt (*C function*), 3372
gpio_pin_get (*C function*), 3378
gpio_pin_get_config (*C function*), 3374
gpio_pin_get_config_dt (*C function*), 3374
gpio_pin_get_dt (*C function*), 3378
gpio_pin_get_raw (*C function*), 3378
gpio_pin_interrupt_configure (*C function*), 3371
gpio_pin_interrupt_configure_dt (*C function*), 3372
gpio_pin_is_input (*C function*), 3373
gpio_pin_is_input_dt (*C function*), 3373
gpio_pin_is_output (*C function*), 3373
gpio_pin_is_output_dt (*C function*), 3374
gpio_pin_set (*C function*), 3379

gpio_pin_set_dt (*C function*), 3379
gpio_pin_set_raw (*C function*), 3379
gpio_pin_t (*C type*), 3370
gpio_pin_toggle (*C function*), 3379
gpio_pin_toggle_dt (*C function*), 3380
gpio_port_clear_bits (*C function*), 3377
gpio_port_clear_bits_raw (*C function*), 3376
gpio_port_get (*C function*), 3375
gpio_port_get_direction (*C function*), 3373
gpio_port_get_raw (*C function*), 3375
gpio_port_pins_t (*C type*), 3370
gpio_port_set_bits (*C function*), 3376
gpio_port_set_bits_raw (*C function*), 3376
gpio_port_set_clr_bits (*C function*), 3377
gpio_port_set_clr_bits_raw (*C function*), 3377
gpio_port_set_masked (*C function*), 3376
gpio_port_set_masked_raw (*C function*), 3375
gpio_port_toggle_bits (*C function*), 3377
gpio_port_value_t (*C type*), 3370
GPIO_PULL_DOWN (*C macro*), 3363
GPIO_PULL_UP (*C macro*), 3363
gpio_remove_callback (*C function*), 3381
gpio_remove_callback_dt (*C function*), 3381
gptp_call_phase_dis_cb (*C function*), 2959
gptp_clk_src_time_invoke (*C function*), 2960
gptp_clk_src_time_invoke_params (*C struct*), 2963
gptp_clk_src_time_invoke_params.last_gm_freq_change (*C var*), 2963
gptp_clk_src_time_invoke_params.last_gm_phase_change (*C var*), 2963
gptp_clk_src_time_invoke_params.src_time (*C var*), 2963
gptp_clk_src_time_invoke_params.time_base_indicator (*C var*), 2963
gptp_event_capture (*C function*), 2959
gptp_flags (*C struct*), 2961
gptp_flags.all (*C var*), 2961
gptp_flags.octets (*C var*), 2961
gptp_foreach_port (*C function*), 2960
gptp_get_domain (*C function*), 2960
gptp_get_hdr (*C function*), 2960
gptp_hdr (*C struct*), 2961
gptp_hdr.control (*C var*), 2962
gptp_hdr.correction_field (*C var*), 2962
gptp_hdr.domain_number (*C var*), 2962
gptp_hdr.flags (*C var*), 2962
gptp_hdr.log_msg_interval (*C var*), 2962
gptp_hdr.message_length (*C var*), 2962
gptp_hdr.message_type (*C var*), 2961
gptp_hdr.port_id (*C var*), 2962
gptp_hdr.ptp_version (*C var*), 2962
gptp_hdr.reserved0 (*C var*), 2962
gptp_hdr.reserved1 (*C var*), 2962
gptp_hdr.reserved2 (*C var*), 2962
gptp_hdr.sequence_id (*C var*), 2962
gptp_hdr.transport_specific (*C var*), 2961
gptp_phase_dis_callback_t (*C type*), 2959
gptp_phase_dis_cb (*C struct*), 2962
gptp_phase_dis_cb.cb (*C var*), 2963
gptp_phase_dis_cb.node (*C var*), 2963
gptp_port_cb_t (*C type*), 2959
gptp_port_identity (*C struct*), 2961

gptp_port_identity.clk_id (*C var*), 2961
gptp_port_identity.port_number (*C var*), 2961
gptp_register_phase_dis_cb (*C function*), 2959
gptp_scaled_ns (*C struct*), 2960
gptp_scaled_ns.high (*C var*), 2960
gptp_scaled_ns.low (*C var*), 2960
gptp_sprint_clock_id (*C function*), 2960
gptp_unregister_phase_dis_cb (*C function*), 2959
gptp_uscaled_ns (*C struct*), 2961
gptp_uscaled_ns.high (*C var*), 2961
gptp_uscaled_ns.low (*C var*), 2961
GROVE_RGB_BLUE (*C macro*), 3297
GROVE_RGB_GREEN (*C macro*), 3297
GROVE_RGB_RED (*C macro*), 3297
GROVE_RGB_WHITE (*C macro*), 3297

H

heap_event_types (*C enum*), 574
heap_event_types.HEAP_ALLOC (*C enumerator*), 574
heap_event_types.HEAP_EVT_UNKNOWN (*C enumerator*), 574
heap_event_types.HEAP_FREE (*C enumerator*), 575
heap_event_types.HEAP_MAX_EVENTS (*C enumerator*), 575
heap_event_types.HEAP_REALLOC (*C enumerator*), 575
heap_event_types.HEAP_RESIZE (*C enumerator*), 574
HEAP_ID_FROM_POINTER (*C macro*), 572
HEAP_ID_LIBC (*C macro*), 572
heap_listener (*C struct*), 576
heap_listener_alloc_cb_t (*C type*), 573
HEAP_LISTENER_ALLOC_DEFINE (*C macro*), 572
heap_listener_free_cb_t (*C type*), 574
HEAP_LISTENER_FREE_DEFINE (*C macro*), 572
heap_listener_notify_alloc (*C function*), 575
heap_listener_notify_free (*C function*), 575
heap_listener_notify_resize (*C function*), 575
heap_listener_register (*C function*), 575
heap_listener_resize_cb_t (*C type*), 573
HEAP_LISTENER_RESIZE_DEFINE (*C macro*), 573
heap_listener_unregister (*C function*), 575
heap_listener.event (*C var*), 576
heap_listener.heap_id (*C var*), 576
heap_listener.node (*C var*), 576
hex2bin (*C function*), 697
hex2char (*C function*), 697
hex_file (*runners.core.RunnerConfig attribute*), 197
HFP_HF_CMD_CME_ERROR (*C macro*), 1705
HFP_HF_CMD_ERROR (*C macro*), 1705
HFP_HF_CMD_OK (*C macro*), 1705
HFP_HF_CMD_UNKNOWN_ERROR (*C macro*), 1705
HID_BOOT_IFACE_CODE_KEYBOARD (*C macro*), 3097
HID_BOOT_IFACE_CODE_MOUSE (*C macro*), 3097
HID_BOOT_IFACE_CODE_NONE (*C macro*), 3097
hid_cb_t (*C type*), 3033
HID_COLLECTION (*C macro*), 3101
HID_COLLECTION_APPLICATION (*C macro*), 3099
HID_COLLECTION_LOGICAL (*C macro*), 3099
HID_COLLECTION_MODIFIER (*C macro*), 3099
HID_COLLECTION_NAMED_ARRAY (*C macro*), 3099
HID_COLLECTION_PHYSICAL (*C macro*), 3099

HID_COLLECTION_REPORT (*C macro*), 3099
HID_COLLECTION_USAGE_SWITCH (*C macro*), 3099
hid_device_ops (*C struct*), 3064
hid_device_ops.get_idle (*C var*), 3064
hid_device_ops.get_report (*C var*), 3064
hid_device_ops.iface_ready (*C var*), 3064
hid_device_ops.input_report_done (*C var*), 3065
hid_device_ops.output_report (*C var*), 3065
hid_device_ops.set_idle (*C var*), 3064
hid_device_ops.set_protocol (*C var*), 3065
hid_device_ops.set_report (*C var*), 3064
hid_device_ops.sof (*C var*), 3065
hid_device_register (*C function*), 3063
hid_device_submit_report (*C function*), 3063
HID_END_COLLECTION (*C macro*), 3101
HID_FEATURE (*C macro*), 3101
hid_idle_cb_t (*C type*), 3034
HID_INPUT (*C macro*), 3101
hid_int_ep_read (*C function*), 3034
hid_int_ep_write (*C function*), 3034
hid_int_ready_callback (*C type*), 3033
HID_ITEM (*C macro*), 3100
HID_ITEM_TAG_COLLECTION (*C macro*), 3098
HID_ITEM_TAG_COLLECTION_END (*C macro*), 3098
HID_ITEM_TAG_FEATURE (*C macro*), 3098
HID_ITEM_TAG_INPUT (*C macro*), 3098
HID_ITEM_TAG_LOGICAL_MAX (*C macro*), 3098
HID_ITEM_TAG_LOGICAL_MIN (*C macro*), 3098
HID_ITEM_TAG_OUTPUT (*C macro*), 3098
HID_ITEM_TAG_PHYSICAL_MAX (*C macro*), 3098
HID_ITEM_TAG_PHYSICAL_MIN (*C macro*), 3098
HID_ITEM_TAG_REPORT_COUNT (*C macro*), 3099
HID_ITEM_TAG_REPORT_ID (*C macro*), 3099
HID_ITEM_TAG_REPORT_SIZE (*C macro*), 3098
HID_ITEM_TAG_UNIT (*C macro*), 3098
HID_ITEM_TAG_UNIT_EXPONENT (*C macro*), 3098
HID_ITEM_TAG_USAGE (*C macro*), 3099
HID_ITEM_TAG_USAGE_MAX (*C macro*), 3099
HID_ITEM_TAG_USAGE_MIN (*C macro*), 3099
HID_ITEM_TAG_USAGE_PAGE (*C macro*), 3098
HID_ITEM_TYPE_GLOBAL (*C macro*), 3098
HID_ITEM_TYPE_LOCAL (*C macro*), 3098
HID_ITEM_TYPE_MAIN (*C macro*), 3098
hid_kbd_code (*C enum*), 3105
hid_kbd_code.HID_KEY_0 (*C enumerator*), 3106
hid_kbd_code.HID_KEY_1 (*C enumerator*), 3106
hid_kbd_code.HID_KEY_2 (*C enumerator*), 3106
hid_kbd_code.HID_KEY_3 (*C enumerator*), 3106
hid_kbd_code.HID_KEY_4 (*C enumerator*), 3106
hid_kbd_code.HID_KEY_5 (*C enumerator*), 3106
hid_kbd_code.HID_KEY_6 (*C enumerator*), 3106
hid_kbd_code.HID_KEY_7 (*C enumerator*), 3106
hid_kbd_code.HID_KEY_8 (*C enumerator*), 3106
hid_kbd_code.HID_KEY_9 (*C enumerator*), 3106
hid_kbd_code.HID_KEY_A (*C enumerator*), 3105
hid_kbd_code.HID_KEY_APOSTROPHE (*C enumerator*), 3107
hid_kbd_code.HID_KEY_B (*C enumerator*), 3105
hid_kbd_code.HID_KEY_BACKSLASH (*C enumerator*), 3107

hid_kbd_code.HID_KEY_BACKSPACE (*C enumerator*), 3106
hid_kbd_code.HID_KEY_C (*C enumerator*), 3105
hid_kbd_code.HID_KEY_CAPSLOCK (*C enumerator*), 3107
hid_kbd_code.HID_KEY_COMMA (*C enumerator*), 3107
hid_kbd_code.HID_KEY_D (*C enumerator*), 3105
hid_kbd_code.HID_KEY_DELETE (*C enumerator*), 3108
hid_kbd_code.HID_KEY_DOT (*C enumerator*), 3107
hid_kbd_code.HID_KEY_DOWN (*C enumerator*), 3108
hid_kbd_code.HID_KEY_E (*C enumerator*), 3105
hid_kbd_code.HID_KEY_END (*C enumerator*), 3108
hid_kbd_code.HID_KEY_ENTER (*C enumerator*), 3106
hid_kbd_code.HID_KEY_EQUAL (*C enumerator*), 3107
hid_kbd_code.HID_KEY_ESC (*C enumerator*), 3106
hid_kbd_code.HID_KEY_F (*C enumerator*), 3105
hid_kbd_code.HID_KEY_F1 (*C enumerator*), 3107
hid_kbd_code.HID_KEY_F2 (*C enumerator*), 3107
hid_kbd_code.HID_KEY_F3 (*C enumerator*), 3107
hid_kbd_code.HID_KEY_F4 (*C enumerator*), 3107
hid_kbd_code.HID_KEY_F5 (*C enumerator*), 3107
hid_kbd_code.HID_KEY_F6 (*C enumerator*), 3107
hid_kbd_code.HID_KEY_F7 (*C enumerator*), 3107
hid_kbd_code.HID_KEY_F8 (*C enumerator*), 3107
hid_kbd_code.HID_KEY_F9 (*C enumerator*), 3107
hid_kbd_code.HID_KEY_F10 (*C enumerator*), 3107
hid_kbd_code.HID_KEY_F11 (*C enumerator*), 3107
hid_kbd_code.HID_KEY_F12 (*C enumerator*), 3108
hid_kbd_code.HID_KEY_G (*C enumerator*), 3105
hid_kbd_code.HID_KEY_GRAVE (*C enumerator*), 3107
hid_kbd_code.HID_KEY_H (*C enumerator*), 3105
hid_kbd_code.HID_KEY_HASH (*C enumerator*), 3107
hid_kbd_code.HID_KEY_HOME (*C enumerator*), 3108
hid_kbd_code.HID_KEY_I (*C enumerator*), 3105
hid_kbd_code.HID_KEY_INSERT (*C enumerator*), 3108
hid_kbd_code.HID_KEY_J (*C enumerator*), 3105
hid_kbd_code.HID_KEY_K (*C enumerator*), 3105
hid_kbd_code.HID_KEY_KP_0 (*C enumerator*), 3109
hid_kbd_code.HID_KEY_KP_1 (*C enumerator*), 3108
hid_kbd_code.HID_KEY_KP_2 (*C enumerator*), 3108
hid_kbd_code.HID_KEY_KP_3 (*C enumerator*), 3108
hid_kbd_code.HID_KEY_KP_4 (*C enumerator*), 3109
hid_kbd_code.HID_KEY_KP_5 (*C enumerator*), 3109
hid_kbd_code.HID_KEY_KP_6 (*C enumerator*), 3109
hid_kbd_code.HID_KEY_KP_7 (*C enumerator*), 3109
hid_kbd_code.HID_KEY_KP_8 (*C enumerator*), 3109
hid_kbd_code.HID_KEY_KP_9 (*C enumerator*), 3109
hid_kbd_code.HID_KEY_KPASTERISK (*C enumerator*), 3108
hid_kbd_code.HID_KEY_KPENTER (*C enumerator*), 3108
hid_kbd_code.HID_KEY_KPMINUS (*C enumerator*), 3108
hid_kbd_code.HID_KEY_KPPLUS (*C enumerator*), 3108
hid_kbd_code.HID_KEY_KPSLASH (*C enumerator*), 3108
hid_kbd_code.HID_KEY_L (*C enumerator*), 3105
hid_kbd_code.HID_KEY_LEFT (*C enumerator*), 3108
hid_kbd_code.HID_KEY_LEFTBRACE (*C enumerator*), 3107
hid_kbd_code.HID_KEY_M (*C enumerator*), 3105
hid_kbd_code.HID_KEY_MINUS (*C enumerator*), 3106
hid_kbd_code.HID_KEY_N (*C enumerator*), 3105
hid_kbd_code.HID_KEY_NUMLOCK (*C enumerator*), 3108
hid_kbd_code.HID_KEY_O (*C enumerator*), 3105

hid_kbd_code.HID_KEY_P (*C enumerator*), 3105
hid_kbd_code.HID_KEY_PAGEDOWN (*C enumerator*), 3108
hid_kbd_code.HID_KEY_PAGEUP (*C enumerator*), 3108
hid_kbd_code.HID_KEY_PAUSE (*C enumerator*), 3108
hid_kbd_code.HID_KEY_Q (*C enumerator*), 3105
hid_kbd_code.HID_KEY_R (*C enumerator*), 3105
hid_kbd_code.HID_KEY_RIGHT (*C enumerator*), 3108
hid_kbd_code.HID_KEY_RIGHTBRACE (*C enumerator*), 3107
hid_kbd_code.HID_KEY_S (*C enumerator*), 3105
hid_kbd_code.HID_KEY_SCROLLLOCK (*C enumerator*), 3108
hid_kbd_code.HID_KEY_SEMICOLON (*C enumerator*), 3107
hid_kbd_code.HID_KEY_SLASH (*C enumerator*), 3107
hid_kbd_code.HID_KEY_SPACE (*C enumerator*), 3106
hid_kbd_code.HID_KEY_SYSRQ (*C enumerator*), 3108
hid_kbd_code.HID_KEY_T (*C enumerator*), 3106
hid_kbd_code.HID_KEY_TAB (*C enumerator*), 3106
hid_kbd_code.HID_KEY_U (*C enumerator*), 3106
hid_kbd_code.HID_KEY_UP (*C enumerator*), 3108
hid_kbd_code.HID_KEY_V (*C enumerator*), 3106
hid_kbd_code.HID_KEY_W (*C enumerator*), 3106
hid_kbd_code.HID_KEY_X (*C enumerator*), 3106
hid_kbd_code.HID_KEY_Y (*C enumerator*), 3106
hid_kbd_code.HID_KEY_Z (*C enumerator*), 3106
hid_kbd_led (*C enum*), 3109
hid_kbd_led.HID_KBD_LED_CAPS_LOCK (*C enumerator*), 3109
hid_kbd_led.HID_KBD_LED_COMPOSE (*C enumerator*), 3110
hid_kbd_led.HID_KBD_LED_KANA (*C enumerator*), 3110
hid_kbd_led.HID_KBD_LED_NUM_LOCK (*C enumerator*), 3109
hid_kbd_led.HID_KBD_LED_SCROLL_LOCK (*C enumerator*), 3109
hid_kbd_modifier (*C enum*), 3109
hid_kbd_modifier.HID_KBD_MODIFIER_LEFT_ALT (*C enumerator*), 3109
hid_kbd_modifier.HID_KBD_MODIFIER_LEFT_CTRL (*C enumerator*), 3109
hid_kbd_modifier.HID_KBD_MODIFIER_LEFT_SHIFT (*C enumerator*), 3109
hid_kbd_modifier.HID_KBD_MODIFIER_LEFT_UI (*C enumerator*), 3109
hid_kbd_modifier.HID_KBD_MODIFIER_NONE (*C enumerator*), 3109
hid_kbd_modifier.HID_KBD_MODIFIER_RIGHT_ALT (*C enumerator*), 3109
hid_kbd_modifier.HID_KBD_MODIFIER_RIGHT_CTRL (*C enumerator*), 3109
hid_kbd_modifier.HID_KBD_MODIFIER_RIGHT_SHIFT (*C enumerator*), 3109
hid_kbd_modifier.HID_KBD_MODIFIER_RIGHT_UI (*C enumerator*), 3109
HID_KEYBOARD_REPORT_DESC (*C macro*), 3104
HID_LOGICAL_MAX8 (*C macro*), 3102
HID_LOGICAL_MAX16 (*C macro*), 3102
HID_LOGICAL_MAX32 (*C macro*), 3102
HID_LOGICAL_MIN8 (*C macro*), 3102
HID_LOGICAL_MIN16 (*C macro*), 3102
HID_LOGICAL_MIN32 (*C macro*), 3102
HID_MOUSE_REPORT_DESC (*C macro*), 3104
hid_ops (*C struct*), 3035
HID_OUTPUT (*C macro*), 3101
HID_PROTOCOL_BOOT (*C macro*), 3097
hid_protocol_cb_t (*C type*), 3034
HID_PROTOCOL_REPORT (*C macro*), 3097
HID_REPORT_COUNT (*C macro*), 3103
HID_REPORT_ID (*C macro*), 3103
HID_REPORT_SIZE (*C macro*), 3103
HID_USAGE (*C macro*), 3103
HID_USAGE_GEN_BUTTON (*C macro*), 3099
HID_USAGE_GEN_DESKTOP (*C macro*), 3099

HID_USAGE_GEN_DESKTOP_GAMEPAD (*C macro*), 3100
HID_USAGE_GEN_DESKTOP_JOYSTICK (*C macro*), 3100
HID_USAGE_GEN_DESKTOP_KEYBOARD (*C macro*), 3100
HID_USAGE_GEN_DESKTOP_KEYPAD (*C macro*), 3100
HID_USAGE_GEN_DESKTOP_MOUSE (*C macro*), 3100
HID_USAGE_GEN_DESKTOP_POINTER (*C macro*), 3100
HID_USAGE_GEN_DESKTOP_UNDEFINED (*C macro*), 3100
HID_USAGE_GEN_DESKTOP_WHEEL (*C macro*), 3100
HID_USAGE_GEN_DESKTOP_X (*C macro*), 3100
HID_USAGE_GEN_DESKTOP_Y (*C macro*), 3100
HID_USAGE_GEN_KEYBOARD (*C macro*), 3099
HID_USAGE_GEN_LEDS (*C macro*), 3099
HID_USAGE_MAX8 (*C macro*), 3104
HID_USAGE_MAX16 (*C macro*), 3104
HID_USAGE_MIN8 (*C macro*), 3103
HID_USAGE_MIN16 (*C macro*), 3104
HID_USAGE_PAGE (*C macro*), 3101
HOST_KBC_EVT_IBF (*C macro*), 3309
HOST_KBC_EVT_OBE (*C macro*), 3309
HOUR_PER_DAY (*C macro*), 476
htonl (*C macro*), 2509
htonll (*C macro*), 2509
htons (*C macro*), 2508
http2_frame (*C struct*), 2807
http2_frame.flags (*C var*), 2807
http2_frame.length (*C var*), 2807
http2_frame.padding_len (*C var*), 2807
http2_frame.stream_identifier (*C var*), 2807
http2_frame.type (*C var*), 2807
http2_stream_ctx (*C struct*), 2806
http2_stream_ctx.end_stream_sent (*C var*), 2807
http2_stream_ctx.headers_sent (*C var*), 2807
http2_stream_ctx.stream_id (*C var*), 2807
http2_stream_ctx.stream_state (*C var*), 2807
http2_stream_ctx.window_size (*C var*), 2807
http_client_ctx (*C struct*), 2807
http_client_ctx.buffer (*C var*), 2807
http_client_ctx.content_len (*C var*), 2808
http_client_ctx.content_type (*C var*), 2808
http_client_ctx.current_detail (*C var*), 2808
http_client_ctx.current_frame (*C var*), 2808
http_client_ctx.current_stream (*C var*), 2808
http_client_ctx.cursor (*C var*), 2807
http_client_ctx.data_len (*C var*), 2808
http_client_ctx.expect_continuation (*C var*), 2809
http_client_ctx.fd (*C var*), 2807
http_client_ctx.has_upgrade_header (*C var*), 2809
http_client_ctx.header_buffer (*C var*), 2808
http_client_ctx.header_field (*C var*), 2808
http_client_ctx.http1_frag_data_len (*C var*), 2809
http_client_ctx.http1_headers_sent (*C var*), 2809
http_client_ctx.http2_upgrade (*C var*), 2809
http_client_ctx.inactivity_timer (*C var*), 2809
http_client_ctx.method (*C var*), 2808
http_client_ctx.parser (*C var*), 2808
http_client_ctx.parser_settings (*C var*), 2808
http_client_ctx.parser_state (*C var*), 2808
http_client_ctx.preface_sent (*C var*), 2809

http_client_ctx.server_state (*C var*), 2808
http_client_ctx.streams (*C var*), 2808
http_client_ctx.url_buffer (*C var*), 2808
http_client_ctx.websocket_sec_key_next (*C var*), 2809
http_client_ctx.websocket_upgrade (*C var*), 2809
http_client_ctx.window_size (*C var*), 2808
http_client_internal_data (*C struct*), 2792
http_client_internal_data.parser (*C var*), 2792
http_client_internal_data.parser_settings (*C var*), 2792
http_client_internal_data.response (*C var*), 2792
http_client_internal_data.sock (*C var*), 2793
http_client_internal_data.user_data (*C var*), 2793
http_client_req (*C function*), 2790
http_data_status (*C enum*), 2804
http_data_status.HTTP_SERVER_DATA_ABORTED (*C enumerator*), 2804
http_data_status.HTTP_SERVER_DATA_FINAL (*C enumerator*), 2804
http_data_status.HTTP_SERVER_DATA_MORE (*C enumerator*), 2804
http_final_call (*C enum*), 2790
http_final_call.HTTP_DATA_FINAL (*C enumerator*), 2790
http_final_call.HTTP_DATA_MORE (*C enumerator*), 2790
http_header_cb_t (*C type*), 2789
http_payload_cb_t (*C type*), 2789
http_request (*C struct*), 2793
http_request.content_type_value (*C var*), 2793
http_request.header_fields (*C var*), 2793
http_request.host (*C var*), 2794
http_request.http_cb (*C var*), 2793
http_request.internal (*C var*), 2793
http_request.method (*C var*), 2793
http_request.optional_headers (*C var*), 2794
http_request.optional_headers_cb (*C var*), 2794
http_request.payload (*C var*), 2794
http_request.payload_cb (*C var*), 2794
http_request.payload_len (*C var*), 2794
http_request.port (*C var*), 2794
http_request.protocol (*C var*), 2793
http_request.recv_buf (*C var*), 2793
http_request.recv_buf_len (*C var*), 2793
http_request.response (*C var*), 2793
http_request.url (*C var*), 2793
HTTP_RESOURCE_DEFINE (*C macro*), 2799
http_resource_desc (*C struct*), 2803
http_resource_desc.detail (*C var*), 2803
http_resource_desc.resource (*C var*), 2803
http_resource_detail (*C struct*), 2805
http_resource_detail_dynamic (*C struct*), 2805
http_resource_detail_dynamic.cb (*C var*), 2806
http_resource_detail_dynamic.common (*C var*), 2806
http_resource_detail_dynamic.data_buffer (*C var*), 2806
http_resource_detail_dynamic.data_buffer_len (*C var*), 2806
http_resource_detail_dynamic.holder (*C var*), 2806
http_resource_detail_dynamic.user_data (*C var*), 2806
http_resource_detail_static (*C struct*), 2805
http_resource_detail_static.common (*C var*), 2805
http_resource_detail_static.static_data (*C var*), 2805
http_resource_detail_static.static_data_len (*C var*), 2805
http_resource_detail_websocket (*C struct*), 2806
http_resource_detail_websocket.cb (*C var*), 2806

http_resource_detail_websocket.common (*C var*), 2806
http_resource_detail_websocket.data_buffer (*C var*), 2806
http_resource_detail_websocket.data_buffer_len (*C var*), 2806
http_resource_detail_websocket.user_data (*C var*), 2806
http_resource_detail_websocket.ws_sock (*C var*), 2806
http_resource_detail.bitmask_of_supported_http_methods (*C var*), 2805
http_resource_detail.content_encoding (*C var*), 2805
http_resource_detail.content_type (*C var*), 2805
http_resource_detail.path_len (*C var*), 2805
http_resource_detail.type (*C var*), 2805
http_resource_dynamic_cb_t (*C type*), 2803
HTTP_RESOURCE_FOREACH (*C macro*), 2802
http_resource_type (*C enum*), 2804
http_resource_type.HTTP_RESOURCE_TYPE_DYNAMIC (*C enumerator*), 2804
http_resource_type.HTTP_RESOURCE_TYPE_STATIC (*C enumerator*), 2804
http_resource_type.HTTP_RESOURCE_TYPE_WEBSOCKET (*C enumerator*), 2804
http_resource_websocket_cb_t (*C type*), 2803
http_response (*C struct*), 2791
http_response_cb_t (*C type*), 2790
http_response.body_found (*C var*), 2792
http_response.body_frag_len (*C var*), 2791
http_response.body_frag_start (*C var*), 2791
http_response.cb (*C var*), 2791
http_response.cl_present (*C var*), 2792
http_response.content_length (*C var*), 2792
http_response.data_len (*C var*), 2791
http_response.http_cb (*C var*), 2791
http_response.http_status (*C var*), 2792
http_response.http_status_code (*C var*), 2792
http_response.message_complete (*C var*), 2792
http_response.processed (*C var*), 2792
http_response.recv_buf (*C var*), 2791
http_response.recv_buf_len (*C var*), 2791
http_server_start (*C function*), 2804
http_server_stop (*C function*), 2805
HTTP_SERVICE_COUNT (*C macro*), 2802
HTTP_SERVICE_DEFINE (*C macro*), 2801
HTTP_SERVICE_DEFINE_EMPTY (*C macro*), 2800
HTTP_SERVICE_FOREACH (*C macro*), 2802
HTTP_SERVICE_FOREACH_RESOURCE (*C macro*), 2802
HTTP_SERVICE_RESOURCE_COUNT (*C macro*), 2802
HTTPS_SERVICE_DEFINE (*C macro*), 2801
HTTPS_SERVICE_DEFINE_EMPTY (*C macro*), 2800
hwinfo_clear_reset_cause (*C function*), 3386
hwinfo_get_device_eui64 (*C function*), 3385
hwinfo_get_device_id (*C function*), 3385
hwinfo_get_reset_cause (*C function*), 3386
hwinfo_get_supported_reset_cause (*C function*), 3386

I

I2C_ADDR_10_BITS (*C macro*), 3449
i2c_burst_read (*C function*), 3463
i2c_burst_read_dt (*C function*), 3463
i2c_burst_write (*C function*), 3464
i2c_burst_write_dt (*C function*), 3464
i2c_callback_t (*C type*), 3452
i2c_configure (*C function*), 3455
I2C_DEVICE_DT_DEFINE (*C macro*), 3451

I2C_DEVICE_DT_INST_DEFINE (*C macro*), 3451
i2c_device_state (*C struct*), 3468
I2C_DT_IODEV_DEFINE (*C macro*), 3451
i2c_dt_spec (*C struct*), 3467
I2C_DT_SPEC_GET (*C macro*), 3450
I2C_DT_SPEC_GET_ON_I2C (*C macro*), 3450
I2C_DT_SPEC_GET_ON_I3C (*C macro*), 3449
I2C_DT_SPEC_INST_GET (*C macro*), 3450
i2c_dump_msgs (*C function*), 3454
i2c_dump_msgs_rw (*C function*), 3454
i2c_get_config (*C function*), 3455
i2c_iodev_api (*C var*), 3467
I2C_IODEV_DEFINE (*C macro*), 3451
i2c_iodev_submit (*C function*), 3459
i2c_is_read_op (*C function*), 3453
i2c_is_ready_dt (*C function*), 3453
I2C_MODE_CONTROLLER (*C macro*), 3449
i2c_msg (*C struct*), 3467
I2C_MSG_ADDR_10_BITS (*C macro*), 3450
I2C_MSG_READ (*C macro*), 3450
I2C_MSG_RESTART (*C macro*), 3450
I2C_MSG_STOP (*C macro*), 3450
I2C_MSG_WRITE (*C macro*), 3450
i2c_msg.buf (*C var*), 3467
i2c_msg.flags (*C var*), 3467
i2c_msg.len (*C var*), 3467
i2c_read (*C function*), 3462
i2c_read_dt (*C function*), 3462
i2c_recover_bus (*C function*), 3459
i2c_reg_read_byte (*C function*), 3464
i2c_reg_read_byte_dt (*C function*), 3465
i2c_reg_update_byte (*C function*), 3466
i2c_reg_update_byte_dt (*C function*), 3466
i2c_reg_write_byte (*C function*), 3465
i2c_reg_write_byte_dt (*C function*), 3466
i2c_rtio_copy (*C function*), 3459
I2C_SPEED_DT (*C macro*), 3449
I2C_SPEED_FAST (*C macro*), 3449
I2C_SPEED_FAST_PLUS (*C macro*), 3449
I2C_SPEED_GET (*C macro*), 3449
I2C_SPEED_HIGH (*C macro*), 3449
I2C_SPEED_MASK (*C macro*), 3449
I2C_SPEED_SET (*C macro*), 3449
I2C_SPEED_SHIFT (*C macro*), 3449
I2C_SPEED_STANDARD (*C macro*), 3449
I2C_SPEED_ULTRA (*C macro*), 3449
i2c_target_callbacks (*C struct*), 3467
i2c_target_config (*C struct*), 3468
i2c_target_config.address (*C var*), 3468
i2c_target_config.callbacks (*C var*), 3468
i2c_target_config.flags (*C var*), 3468
i2c_target_config.node (*C var*), 3468
i2c_target_driver_register (*C function*), 3460
i2c_target_driver_unregister (*C function*), 3461
I2C_TARGET_FLAGS_ADDR_10_BITS (*C macro*), 3451
i2c_target_read_processed_cb_t (*C type*), 3453
i2c_target_read_requested_cb_t (*C type*), 3452
i2c_target_register (*C function*), 3460

i2c_target_stop_cb_t (*C type*), 3453
i2c_target_unregister (*C function*), 3460
i2c_target_write_received_cb_t (*C type*), 3452
i2c_target_write_requested_cb_t (*C type*), 3452
i2c_transfer (*C function*), 3455
i2c_transfer_cb (*C function*), 3456
i2c_transfer_cb_dt (*C function*), 3457
i2c_transfer_dt (*C function*), 3459
i2c_transfer_signal (*C function*), 3458
i2c_write (*C function*), 3461
i2c_write_dt (*C function*), 3461
i2c_write_read (*C function*), 3462
i2c_write_read_cb (*C function*), 3457
i2c_write_read_cb_dt (*C function*), 3458
i2c_write_read_dt (*C function*), 3463
i2c_xfer_stats (*C function*), 3454
i2s_buf_read (*C function*), 3198
i2s_buf_write (*C function*), 3199
i2s_config (*C struct*), 3200
i2s_config_get (*C function*), 3197
i2s_config.block_size (*C var*), 3201
i2s_config.channels (*C var*), 3200
i2s_config.format (*C var*), 3200
i2s_config.frame_clk_freq (*C var*), 3201
i2s_config.mem_slab (*C var*), 3201
i2s_config.options (*C var*), 3200
i2s_config.timeout (*C var*), 3201
i2s_configure (*C function*), 3197
i2s_config.word_size (*C var*), 3200
i2s_dir (*C enum*), 3195
i2s_dir.I2S_DIR_BOTH (*C enumerator*), 3196
i2s_dir.I2S_DIR_RX (*C enumerator*), 3195
i2s_dir.I2S_DIR_TX (*C enumerator*), 3196
I2S_FMT_BIT_CLK_INV (*C macro*), 3194
I2S_FMT_CLK_FORMAT_MASK (*C macro*), 3194
I2S_FMT_CLK_FORMAT_SHIFT (*C macro*), 3194
I2S_FMT_CLK_IF_IB (*C macro*), 3194
I2S_FMT_CLK_IF_NB (*C macro*), 3194
I2S_FMT_CLK_NF_IB (*C macro*), 3194
I2S_FMT_CLK_NF_NB (*C macro*), 3194
I2S_FMT_DATA_FORMAT_I2S (*C macro*), 3192
I2S_FMT_DATA_FORMAT_LEFT_JUSTIFIED (*C macro*), 3193
I2S_FMT_DATA_FORMAT_MASK (*C macro*), 3192
I2S_FMT_DATA_FORMAT_PCM_LONG (*C macro*), 3193
I2S_FMT_DATA_FORMAT_PCM_SHORT (*C macro*), 3193
I2S_FMT_DATA_FORMAT_RIGHT_JUSTIFIED (*C macro*), 3194
I2S_FMT_DATA_FORMAT_SHIFT (*C macro*), 3192
I2S_FMT_DATA_ORDER_INV (*C macro*), 3194
I2S_FMT_DATA_ORDER_LSB (*C macro*), 3194
I2S_FMT_DATA_ORDER_MSB (*C macro*), 3194
I2S_FMT_FRAME_CLK_INV (*C macro*), 3194
i2s_fmt_t (*C type*), 3195
I2S_OPT_BIT_CLK_CONT (*C macro*), 3195
I2S_OPT_BIT_CLK_GATED (*C macro*), 3195
I2S_OPT_BIT_CLK_MASTER (*C macro*), 3195
I2S_OPT_BIT_CLK_SLAVE (*C macro*), 3195
I2S_OPT_FRAME_CLK_MASTER (*C macro*), 3195
I2S_OPT_FRAME_CLK_SLAVE (*C macro*), 3195

I2S_OPT_LOOPBACK (*C macro*), 3195
I2S_OPT_PINGPONG (*C macro*), 3195
i2s_opt_t (*C type*), 3195
i2s_read (*C function*), 3198
i2s_state (*C enum*), 3196
i2s_state.I2S_STATE_ERROR (*C enumerator*), 3196
i2s_state.I2S_STATE_NOT_READY (*C enumerator*), 3196
i2s_state.I2S_STATE_READY (*C enumerator*), 3196
i2s_state.I2S_STATE_RUNNING (*C enumerator*), 3196
i2s_state.I2S_STATE_STOPPING (*C enumerator*), 3196
i2s_trigger (*C function*), 3200
i2s_trigger_cmd (*C enum*), 3196
i2s_trigger_cmd.I2S_TRIGGER_DRAIN (*C enumerator*), 3197
i2s_trigger_cmd.I2S_TRIGGER_DROP (*C enumerator*), 3197
i2s_trigger_cmd.I2S_TRIGGER_PREPARE (*C enumerator*), 3197
i2s_trigger_cmd.I2S_TRIGGER_START (*C enumerator*), 3196
i2s_trigger_cmd.I2S_TRIGGER_STOP (*C enumerator*), 3196
i2s_write (*C function*), 3199
i3c_addr_slot_status (*C enum*), 3441
i3c_addr_slot_status.I3C_ADDR_SLOT_STATUS_FREE (*C enumerator*), 3441
i3c_addr_slot_status.I3C_ADDR_SLOT_STATUS_I2C_DEV (*C enumerator*), 3442
i3c_addr_slot_status.I3C_ADDR_SLOT_STATUS_I3C_DEV (*C enumerator*), 3442
i3c_addr_slot_status.I3C_ADDR_SLOT_STATUS_MASK (*C enumerator*), 3442
i3c_addr_slot_status.I3C_ADDR_SLOT_STATUS_RSVD (*C enumerator*), 3441
i3c_addr_slots (*C struct*), 3443
i3c_addr_slots_init (*C function*), 3442
i3c_addr_slots_is_free (*C function*), 3442
i3c_addr_slots_mark_free (*C function*), 3443
i3c_addr_slots_mark_i2c (*C function*), 3443
i3c_addr_slots_mark_i3c (*C function*), 3443
i3c_addr_slots_mark_rsvd (*C function*), 3443
i3c_addr_slots_next_free_find (*C function*), 3443
i3c_addr_slots_set (*C function*), 3442
i3c_addr_slots_status (*C function*), 3442
i3c_attach_i2c_device (*C function*), 3403
i3c_attach_i3c_device (*C function*), 3402
I3C_BCR_ADV_CAPABILITIES (*C macro*), 3394
I3C_BCR_DEVICE_ROLE (*C macro*), 3394
I3C_BCR_DEVICE_ROLE_I3C_CONTROLLER_CAPABLE (*C macro*), 3394
I3C_BCR_DEVICE_ROLE_I3C_TARGET (*C macro*), 3394
I3C_BCR_DEVICE_ROLE_MASK (*C macro*), 3394
I3C_BCR_DEVICE_ROLE_SHIFT (*C macro*), 3394
I3C_BCR_IBI_PAYLOAD_HAS_DATA_BYTE (*C macro*), 3393
I3C_BCR_IBI_REQUEST_CAPABLE (*C macro*), 3393
I3C_BCR_MAX_DATA_SPEED_LIMIT (*C macro*), 3393
I3C_BCR_OFFLINE_CAPABLE (*C macro*), 3394
I3C_BCR_VIRTUAL_TARGET (*C macro*), 3394
I3C_BROADCAST_ADDR (*C macro*), 3441
i3c_bus_init (*C function*), 3405
i3c_bus_mode (*C enum*), 3396
i3c_bus_mode.I3C_BUS_MODE_INVALID (*C enumerator*), 3396
i3c_bus_mode.I3C_BUS_MODE_MAX (*C enumerator*), 3396
i3c_bus_mode.I3C_BUS_MODE_MIXED_FAST (*C enumerator*), 3396
i3c_bus_mode.I3C_BUS_MODE_MIXED_LIMITED (*C enumerator*), 3396
i3c_bus_mode.I3C_BUS_MODE_MIXED_SLOW (*C enumerator*), 3396
i3c_bus_mode.I3C_BUS_MODE_PURE (*C enumerator*), 3396
i3c_ccc_address (*C struct*), 3435
i3c_ccc_address.addr (*C var*), 3436

I3C_CCC_BROADCAST_MAX_ID (*C macro*), 3412
I3C_CCC_D2DXFER (*C macro*), 3415
I3C_CCC_DEFGRPA (*C macro*), 3414
I3C_CCC_DEFTGTS (*C macro*), 3413
i3c_ccc_deftgts (*C struct*), 3435
i3c_ccc_deftgts_active_controller (*C struct*), 3434
i3c_ccc_deftgts_active_controller.addr (*C var*), 3434
i3c_ccc_deftgts_active_controller.bcr (*C var*), 3434
i3c_ccc_deftgts_active_controller.dcr (*C var*), 3434
i3c_ccc_deftgts_active_controller.static_addr (*C var*), 3435
i3c_ccc_deftgts_target (*C struct*), 3435
i3c_ccc_deftgts_target.addr (*C var*), 3435
i3c_ccc_deftgts_target.bcr (*C var*), 3435
i3c_ccc_deftgts_target.dcr (*C var*), 3435
i3c_ccc_deftgts_target.lvr (*C var*), 3435
i3c_ccc_deftgts_target.static_addr (*C var*), 3435
i3c_ccc_deftgts.active_controller (*C var*), 3435
i3c_ccc_deftgts.targets (*C var*), 3435
I3C_CCC_DISEC (*C macro*), 3412
I3C_CCC_DISEC_EVT_ALL (*C macro*), 3416
I3C_CCC_DISEC_EVT_DISCR (*C macro*), 3416
I3C_CCC_DISEC_EVT_DISHJ (*C macro*), 3416
I3C_CCC_DISEC_EVT_DISINTR (*C macro*), 3416
i3c_ccc_do_events_all_set (*C function*), 3427
i3c_ccc_do_events_set (*C function*), 3427
i3c_ccc_do_getbcr (*C function*), 3425
i3c_ccc_do_getcaps (*C function*), 3431
i3c_ccc_do_getcaps_fmt1 (*C function*), 3431
i3c_ccc_do_getcaps_fmt2 (*C function*), 3432
i3c_ccc_do_getdcr (*C function*), 3425
i3c_ccc_do_getmrl (*C function*), 3429
i3c_ccc_do_getmwl (*C function*), 3428
i3c_ccc_do_getpid (*C function*), 3425
i3c_ccc_do_getstatus (*C function*), 3430
i3c_ccc_do_getstatus_fmt1 (*C function*), 3430
i3c_ccc_do_getstatus_fmt2 (*C function*), 3431
i3c_ccc_do_rstact_all (*C function*), 3426
i3c_ccc_do_rstdaa_all (*C function*), 3426
i3c_ccc_do_setdasa (*C function*), 3426
i3c_ccc_do_setmrl (*C function*), 3429
i3c_ccc_do_setmrl_all (*C function*), 3429
i3c_ccc_do_setmwl (*C function*), 3428
i3c_ccc_do_setmwl_all (*C function*), 3428
i3c_ccc_do_setnewda (*C function*), 3427
I3C_CCC_ENDXFER (*C macro*), 3413
I3C_CCC_ENEC (*C macro*), 3412
I3C_CCC_ENEC_EVT_ALL (*C macro*), 3416
I3C_CCC_ENEC_EVT_ENCR (*C macro*), 3416
I3C_CCC_ENEC_EVT_ENHJ (*C macro*), 3416
I3C_CCC_ENEC_EVT_ENINTR (*C macro*), 3415
I3C_CCC_ENTAS (*C macro*), 3412
I3C_CCC_ENTAS0 (*C macro*), 3412
I3C_CCC_ENTAS1 (*C macro*), 3412
I3C_CCC_ENTAS2 (*C macro*), 3413
I3C_CCC_ENTAS3 (*C macro*), 3413
I3C_CCC_ENTDAA (*C macro*), 3413
I3C_CCC_ENTHDR (*C macro*), 3413
I3C_CCC_ENTHDR0 (*C macro*), 3413

I3C_CCC_ENTHDR1 (*C macro*), 3413
I3C_CCC_ENTHDR2 (*C macro*), 3414
I3C_CCC_ENTHDR3 (*C macro*), 3414
I3C_CCC_ENTHDR4 (*C macro*), 3414
I3C_CCC_ENTHDR5 (*C macro*), 3414
I3C_CCC_ENTHDR6 (*C macro*), 3414
I3C_CCC_ENTHDR7 (*C macro*), 3414
I3C_CCC_ENTTM (*C macro*), 3413
i3c_ccc_events (*C struct*), 3433
i3c_ccc_events.events (*C var*), 3433
I3C_CCC_EVT_ALL (*C macro*), 3416
I3C_CCC_EVT_CR (*C macro*), 3416
I3C_CCC_EVT_HJ (*C macro*), 3416
I3C_CCC_EVT_INTR (*C macro*), 3416
I3C_CCC_GETACCCR (*C macro*), 3415
I3C_CCC_GETBCR (*C macro*), 3415
i3c_ccc_getbcr (*C struct*), 3436
i3c_ccc_getbcr.bcr (*C var*), 3437
I3C_CCC_GETCAPS (*C macro*), 3415
i3c_ccc_getcaps (*C union*), 3439
I3C_CCC_GETCAPS1_HDR_BT (*C macro*), 3419
I3C_CCC_GETCAPS1_HDR_DDR (*C macro*), 3419
I3C_CCC_GETCAPS1_HDR_MODE (*C macro*), 3419
I3C_CCC_GETCAPS1_HDR_MODE0 (*C macro*), 3419
I3C_CCC_GETCAPS1_HDR_MODE1 (*C macro*), 3419
I3C_CCC_GETCAPS1_HDR_MODE2 (*C macro*), 3419
I3C_CCC_GETCAPS1_HDR_MODE3 (*C macro*), 3419
I3C_CCC_GETCAPS1_HDR_MODE4 (*C macro*), 3419
I3C_CCC_GETCAPS1_HDR_MODE5 (*C macro*), 3419
I3C_CCC_GETCAPS1_HDR_MODE6 (*C macro*), 3420
I3C_CCC_GETCAPS1_HDR_MODE7 (*C macro*), 3420
I3C_CCC_GETCAPS1_HDR_TSL (*C macro*), 3419
I3C_CCC_GETCAPS1_HDR_TSP (*C macro*), 3419
I3C_CCC_GETCAPS2_GRPADDR_CAP (*C macro*), 3420
I3C_CCC_GETCAPS2_GRPADDR_CAP_MASK (*C macro*), 3420
I3C_CCC_GETCAPS2_GRPADDR_CAP_SHIFT (*C macro*), 3420
I3C_CCC_GETCAPS2_HDRDDR_ABORT_CRC (*C macro*), 3420
I3C_CCC_GETCAPS2_HDRDDR_WRITE_ABORT (*C macro*), 3420
I3C_CCC_GETCAPS2_SPEC_VER (*C macro*), 3420
I3C_CCC_GETCAPS2_SPEC_VER_MASK (*C macro*), 3420
I3C_CCC_GETCAPS2_SPEC_VER_SHIFT (*C macro*), 3420
I3C_CCC_GETCAPS3_D2DXFER_IBI_CAPABLE (*C macro*), 3421
I3C_CCC_GETCAPS3_D2DXFER_SUPPORT (*C macro*), 3421
I3C_CCC_GETCAPS3_GETCAPS_DEFINING_BYTE_SUPPORT (*C macro*), 3421
I3C_CCC_GETCAPS3_GETSTATUS_DEFINING_BYTE_SUPPORT (*C macro*), 3421
I3C_CCC_GETCAPS3_HDRBT_CRC32_SUPPORT (*C macro*), 3421
I3C_CCC_GETCAPS3_IBI_MDR_PENDING_READ_NOTIFICATION (*C macro*), 3421
I3C_CCC_GETCAPS3_MLANE_SUPPORT (*C macro*), 3420
I3C_CCC_GETCAPS_CRCAPS1_GRP_MANAGEMENT_SUPPORT (*C macro*), 3421
I3C_CCC_GETCAPS_CRCAPS1_HJ_SUPPORT (*C macro*), 3421
I3C_CCC_GETCAPS_CRCAPS1_ML_SUPPORT (*C macro*), 3421
I3C_CCC_GETCAPS_CRCAPS2_CONTROLLER_PASSBACK (*C macro*), 3422
I3C_CCC_GETCAPS_CRCAPS2_DEEP_SLEEP_CAPABLE (*C macro*), 3422
I3C_CCC_GETCAPS_CRCAPS2_DELAYED_CONTROLLER_HANDOFF (*C macro*), 3422
I3C_CCC_GETCAPS_CRCAPS2_IBI_TIR_SUPPORT (*C macro*), 3422
i3c_ccc_getcaps_defbyte (*C enum*), 3424
i3c_ccc_getcaps_defbyte.GETCAPS_FORMAT_2_CRCAPS (*C enumerator*), 3424
i3c_ccc_getcaps_defbyte.GETCAPS_FORMAT_2_DBGCAPS (*C enumerator*), 3424

i3c_ccc_getcaps_defbyte.GETCAPS_FORMAT_2_INVALID (*C enumerator*), 3424
i3c_ccc_getcaps_defbyte.GETCAPS_FORMAT_2_TESTPAT (*C enumerator*), 3424
i3c_ccc_getcaps_defbyte.GETCAPS_FORMAT_2_TGTCAPS (*C enumerator*), 3424
i3c_ccc_getcaps_defbyte.GETCAPS_FORMAT_2_VTCAPS (*C enumerator*), 3424
i3c_ccc_getcaps_fmt (*C enum*), 3423
i3c_ccc_getcaps_fmt.GETCAPS_FORMAT_1 (*C enumerator*), 3423
i3c_ccc_getcaps_fmt.GETCAPS_FORMAT_2 (*C enumerator*), 3423
I3C_CCC_GETCAPS_TESTPAT (*C macro*), 3421
I3C_CCC_GETCAPS_TESTPAT1 (*C macro*), 3421
I3C_CCC_GETCAPS_TESTPAT2 (*C macro*), 3421
I3C_CCC_GETCAPS_TESTPAT3 (*C macro*), 3421
I3C_CCC_GETCAPS_TESTPAT4 (*C macro*), 3421
I3C_CCC_GETCAPS_VTCAP1_SHARED_PERIPH_DETECT (*C macro*), 3422
I3C_CCC_GETCAPS_VTCAP1_SIDE_EFFECTS (*C macro*), 3422
I3C_CCC_GETCAPS_VTCAP1_VITRUAL_TARGET_TYPE (*C macro*), 3422
I3C_CCC_GETCAPS_VTCAP1_VITRUAL_TARGET_TYPE_MASK (*C macro*), 3422
I3C_CCC_GETCAPS_VTCAP1_VITRUAL_TARGET_TYPE_SHIFT (*C macro*), 3422
I3C_CCC_GETCAPS_VTCAP2_ADDRESS_REMAPPING (*C macro*), 3422
I3C_CCC_GETCAPS_VTCAP2_BUS_CONTEXT_AND_COND (*C macro*), 3423
I3C_CCC_GETCAPS_VTCAP2_BUS_CONTEXT_AND_COND_MASK (*C macro*), 3423
I3C_CCC_GETCAPS_VTCAP2_BUS_CONTEXT_AND_COND_SHIFT (*C macro*), 3423
I3C_CCC_GETCAPS_VTCAP2_INTERRUPT_REQUESTS (*C macro*), 3422
I3C_CCC_GETCAPS_VTCAP2_INTERRUPT_REQUESTS_MASK (*C macro*), 3422
I3C_CCC_GETCAPS_VTCAP2_INTERRUPT_REQUESTS_SHIFT (*C macro*), 3422
i3c_ccc_getcaps.crcaps (*C var*), 3441
i3c_ccc_getcaps.fmt1 (*C var*), 3440
i3c_ccc_getcaps.fmt2 (*C var*), 3441
i3c_ccc_getcaps.getcaps (*C var*), 3440
i3c_ccc_getcaps.gethdrcap (*C var*), 3440
i3c_ccc_getcaps.testpat (*C var*), 3440
i3c_ccc_getcaps.tgtcaps (*C var*), 3440
i3c_ccc_getcaps.vtcaps (*C var*), 3441
I3C_CCC_GETDCR (*C macro*), 3415
i3c_ccc_getdcr (*C struct*), 3437
i3c_ccc_getdcr.dcr (*C var*), 3437
I3C_CCC_GETMRL (*C macro*), 3415
I3C_CCC_GETMWL (*C macro*), 3415
I3C_CCC_GETMXDS (*C macro*), 3415
i3c_ccc_getmxds (*C union*), 3439
I3C_CCC_GETMXDS_CRDHLY1_CTRL_HANDOFF_ACT_STATE (*C macro*), 3419
I3C_CCC_GETMXDS_CRDHLY1_CTRL_HANDOFF_ACT_STATE_MASK (*C macro*), 3419
I3C_CCC_GETMXDS_CRDHLY1_CTRL_HANDOFF_ACT_STATE_SHIFT (*C macro*), 3419
I3C_CCC_GETMXDS_CRDHLY1_SET_BUS_ACT_STATE (*C macro*), 3418
I3C_CCC_GETMXDS_MAX_SDR_FSCL_2MHZ (*C macro*), 3417
I3C_CCC_GETMXDS_MAX_SDR_FSCL_4MHZ (*C macro*), 3417
I3C_CCC_GETMXDS_MAX_SDR_FSCL_6MHZ (*C macro*), 3417
I3C_CCC_GETMXDS_MAX_SDR_FSCL_8MHZ (*C macro*), 3417
I3C_CCC_GETMXDS_MAX_SDR_FSCL_MAX (*C macro*), 3417
I3C_CCC_GETMXDS_MAXRD_MAX_SDR_FSCL (*C macro*), 3418
I3C_CCC_GETMXDS_MAXRD_MAX_SDR_FSCL_MASK (*C macro*), 3418
I3C_CCC_GETMXDS_MAXRD_MAX_SDR_FSCL_SHIFT (*C macro*), 3418
I3C_CCC_GETMXDS_MAXRD_TSCO (*C macro*), 3418
I3C_CCC_GETMXDS_MAXRD_TSCO_MASK (*C macro*), 3418
I3C_CCC_GETMXDS_MAXRD_TSCO_SHIFT (*C macro*), 3418
I3C_CCC_GETMXDS_MAXRD_W2R_PERMITS_STOP_BETWEEN (*C macro*), 3418
I3C_CCC_GETMXDS_MAXWR_DEFINING_BYTE_SUPPORT (*C macro*), 3418
I3C_CCC_GETMXDS_MAXWR_MAX_SDR_FSCL (*C macro*), 3418
I3C_CCC_GETMXDS_MAXWR_MAX_SDR_FSCL_MASK (*C macro*), 3418

I3C_CCC_GETMXDS_MAXWR_MAX_SDR_FSCL_SHIFT (*C macro*), 3418
I3C_CCC_GETMXDS_TSCO_8NS (*C macro*), 3417
I3C_CCC_GETMXDS_TSCO_9NS (*C macro*), 3417
I3C_CCC_GETMXDS_TSCO_10NS (*C macro*), 3417
I3C_CCC_GETMXDS_TSCO_11NS (*C macro*), 3417
I3C_CCC_GETMXDS_TSCO_12NS (*C macro*), 3417
I3C_CCC_GETMXDS_TSCO_GT_12NS (*C macro*), 3417
i3c_ccc_getmxds.crhdly1 (*C var*), 3439
i3c_ccc_getmxds.fmt1 (*C var*), 3439
i3c_ccc_getmxds.fmt2 (*C var*), 3439
i3c_ccc_getmxds.fmt3 (*C var*), 3439
i3c_ccc_getmxds.maxrd (*C var*), 3439
i3c_ccc_getmxds.maxrdturn (*C var*), 3439
i3c_ccc_getmxds.maxwr (*C var*), 3439
i3c_ccc_getmxds.wrrdturn (*C var*), 3439
I3C_CCC_GETPID (*C macro*), 3415
i3c_ccc_getpid (*C struct*), 3436
i3c_ccc_getpid.pid (*C var*), 3436
I3C_CCC_GETSTATUS (*C macro*), 3415
i3c_ccc_getstatus (*C union*), 3437
I3C_CCC_GETSTATUS_ACTIVITY_MODE (*C macro*), 3416
I3C_CCC_GETSTATUS_ACTIVITY_MODE_MASK (*C macro*), 3416
I3C_CCC_GETSTATUS_ACTIVITY_MODE_SHIFT (*C macro*), 3416
i3c_ccc_getstatus_defbyte (*C enum*), 3423
i3c_ccc_getstatus_defbyte.GETSTATUS_FORMAT_2_INVALID (*C enumerator*), 3423
i3c_ccc_getstatus_defbyte.GETSTATUS_FORMAT_2_PRECR (*C enumerator*), 3423
i3c_ccc_getstatus_defbyte.GETSTATUS_FORMAT_2_TGTSTAT (*C enumerator*), 3423
i3c_ccc_getstatus_fmt (*C enum*), 3423
i3c_ccc_getstatus_fmt.GETSTATUS_FORMAT_1 (*C enumerator*), 3423
i3c_ccc_getstatus_fmt.GETSTATUS_FORMAT_2 (*C enumerator*), 3423
I3C_CCC_GETSTATUS_NUM_INT (*C macro*), 3417
I3C_CCC_GETSTATUS_NUM_INT_MASK (*C macro*), 3417
I3C_CCC_GETSTATUS_NUM_INT_SHIFT (*C macro*), 3416
I3C_CCC_GETSTATUS_PRECR_DEEP_SLEEP_DETECTED (*C macro*), 3417
I3C_CCC_GETSTATUS_PRECR_HANDOFF_DELAY_NACK (*C macro*), 3417
I3C_CCC_GETSTATUS_PROTOCOL_ERR (*C macro*), 3416
i3c_ccc_getstatus.fmt1 (*C var*), 3437
i3c_ccc_getstatus.fmt2 (*C var*), 3438
i3c_ccc_getstatus.precr (*C var*), 3437
i3c_ccc_getstatus.raw_u16 (*C var*), 3438
i3c_ccc_getstatus.status (*C var*), 3437
i3c_ccc_getstatus.tgtstat (*C var*), 3437
I3C_CCC_GETXTIME (*C macro*), 3415
i3c_ccc_is_payload_broadcast (*C function*), 3424
I3C_CCC_MLANE (*C macro*), 3414
i3c_ccc_mrl (*C struct*), 3434
i3c_ccc_mrl.ibi_len (*C var*), 3434
i3c_ccc_mrl.len (*C var*), 3434
i3c_ccc_mwl (*C struct*), 3434
i3c_ccc_mwl.len (*C var*), 3434
i3c_ccc_payload (*C struct*), 3433
i3c_ccc_payload.data (*C var*), 3433
i3c_ccc_payload.data_len (*C var*), 3433
i3c_ccc_payload.id (*C var*), 3433
i3c_ccc_payload.num_targets (*C var*), 3433
i3c_ccc_payload.num_xfer (*C var*), 3433
i3c_ccc_payload.payloads (*C var*), 3433
I3C_CCC_RSTACT (*C macro*), 3414

i3c_ccc_rstact_defining_byte (*C enum*), 3424
i3c_ccc_rstact_defining_byte.I3C_CCC_RSTACT_DEBUG_NETWORK_ADAPTER (*C enumerator*), 3424
i3c_ccc_rstact_defining_byte.I3C_CCC_RSTACT_NO_RESET (*C enumerator*), 3424
i3c_ccc_rstact_defining_byte.I3C_CCC_RSTACT_PERIPHERAL_ONLY (*C enumerator*), 3424
i3c_ccc_rstact_defining_byte.I3C_CCC_RSTACT_RESET_WHOLE_TARGET (*C enumerator*), 3424
i3c_ccc_rstact_defining_byte.I3C_CCC_RSTACT_VIRTUAL_TARGET_DETECT (*C enumerator*), 3424
I3C_CCC_RSTDAA (*C macro*), 3413
I3C_CCC_RSTGRPA (*C macro*), 3414
I3C_CCC_SETAASA (*C macro*), 3414
I3C_CCC_SETBRGTGT (*C macro*), 3415
i3c_ccc_setbrgtgt (*C struct*), 3438
i3c_ccc_setbrgtgt_tgt (*C struct*), 3438
i3c_ccc_setbrgtgt_tgt.addr (*C var*), 3438
i3c_ccc_setbrgtgt_tgt.id (*C var*), 3438
i3c_ccc_setbrgtgt.count (*C var*), 3438
i3c_ccc_setbrgtgt.targets (*C var*), 3438
I3C_CCC_SETBUSCON (*C macro*), 3413
I3C_CCC_SETDASA (*C macro*), 3414
I3C_CCC_SETGRPA (*C macro*), 3415
I3C_CCC_SETMRL (*C macro*), 3413
I3C_CCC_SETMWL (*C macro*), 3413
I3C_CCC_SETNEWDA (*C macro*), 3415
I3C_CCC_SETROUTE (*C macro*), 3415
I3C_CCC_SETXTIME (*C macro*), 3414
i3c_ccc_target_payload (*C struct*), 3432
i3c_ccc_target_payload.addr (*C var*), 3432
i3c_ccc_target_payload.data (*C var*), 3432
i3c_ccc_target_payload.data_len (*C var*), 3432
i3c_ccc_target_payload.num_xfer (*C var*), 3432
i3c_ccc_target_payload.rnw (*C var*), 3432
I3C_CCC_VENDOR (*C macro*), 3414
i3c_config_controller (*C struct*), 3405
i3c_config_controller.i2c (*C var*), 3406
i3c_config_controller.i3c (*C var*), 3406
i3c_config_controller.is_secondary (*C var*), 3406
i3c_config_controller.supported_hdr (*C var*), 3406
i3c_config_custom (*C struct*), 3406
i3c_config_custom.id (*C var*), 3406
i3c_config_custom.ptr (*C var*), 3406
i3c_config_custom.val (*C var*), 3406
i3c_config_get (*C function*), 3401
i3c_config_target (*C struct*), 3445
i3c_config_target.bcr (*C var*), 3445
i3c_config_target.dcr (*C var*), 3445
i3c_config_target.enable (*C var*), 3445
i3c_config_target.max_read_len (*C var*), 3445
i3c_config_target.max_write_len (*C var*), 3445
i3c_config_target.pid (*C var*), 3445
i3c_config_target.pid_random (*C var*), 3445
i3c_config_target.static_addr (*C var*), 3445
i3c_config_target.supported_hdr (*C var*), 3445
i3c_config_type (*C enum*), 3399
i3c_config_type.I3C_CONFIG_CONTROLLER (*C enumerator*), 3399
i3c_config_type.I3C_CONFIG_CUSTOM (*C enumerator*), 3399
i3c_config_type.I3C_CONFIG_TARGET (*C enumerator*), 3399
i3c_configure (*C function*), 3401
i3c_data_rate (*C enum*), 3397
i3c_data_rate.I3C_DATA_RATE_HDR_BT (*C enumerator*), 3397

i3c_data_rate.I3C_DATA_RATE_HDR_DDR (*C enumerator*), 3397
i3c_data_rate.I3C_DATA_RATE_HDR_TSL (*C enumerator*), 3397
i3c_data_rate.I3C_DATA_RATE_HDR_TSP (*C enumerator*), 3397
i3c_data_rate.I3C_DATA_RATE_INVALID (*C enumerator*), 3397
i3c_data_rate.I3C_DATA_RATE_MAX (*C enumerator*), 3397
i3c_data_rate.I3C_DATA_RATE_SDR (*C enumerator*), 3397
i3c_detach_i2c_device (*C function*), 3403
i3c_detach_i3c_device (*C function*), 3403
i3c_determine_default_addr (*C function*), 3400
i3c_dev_attached_list (*C struct*), 3410
i3c_dev_attached_list.addr_slots (*C var*), 3411
i3c_dev_attached_list.i2c (*C var*), 3411
i3c_dev_attached_list.i3c (*C var*), 3411
i3c_dev_list (*C struct*), 3411
i3c_dev_list_daa_addr_helper (*C function*), 3400
i3c_dev_list_find (*C function*), 3399
i3c_dev_list_i2c_addr_find (*C function*), 3399
i3c_dev_list_i3c_addr_find (*C function*), 3399
i3c_dev_list.i2c (*C var*), 3411
i3c_dev_list.i3c (*C var*), 3411
i3c_dev_list.num_i2c (*C var*), 3411
i3c_dev_list.num_i3c (*C var*), 3411
i3c_device_basic_info_get (*C function*), 3405
i3c_device_desc (*C struct*), 3407
i3c_device_desc.bcr (*C var*), 3408
i3c_device_desc.bus (*C var*), 3407
i3c_device_desc.dcr (*C var*), 3408
i3c_device_desc.dev (*C var*), 3407
i3c_device_desc.dynamic_addr (*C var*), 3407
i3c_device_desc.getcap1 (*C var*), 3409
i3c_device_desc.getcap2 (*C var*), 3409
i3c_device_desc.getcap3 (*C var*), 3409
i3c_device_desc.getcap4 (*C var*), 3410
i3c_device_desc.getcaps (*C var*), 3410
i3c_device_desc.gethdrcap (*C var*), 3409
i3c_device_desc.group_addr (*C var*), 3408
i3c_device_desc.ibi_cb (*C var*), 3410
i3c_device_desc.init_dynamic_addr (*C var*), 3407
i3c_device_desc.max_ibi (*C var*), 3409
i3c_device_desc.max_read_turnaround (*C var*), 3409
i3c_device_desc.maxrd (*C var*), 3408
i3c_device_desc.maxwr (*C var*), 3408
i3c_device_desc.mrl (*C var*), 3409
i3c_device_desc.mwl (*C var*), 3409
i3c_device_desc.pid (*C var*), 3407
i3c_device_desc.static_addr (*C var*), 3407
i3c_device_find (*C function*), 3405
I3C_DEVICE_ID (*C macro*), 3396
i3c_device_id (*C struct*), 3406
i3c_device_id.pid (*C var*), 3407
i3c_do_ccc (*C function*), 3404
i3c_do_daa (*C function*), 3404
i3c_driver_config (*C struct*), 3411
i3c_driver_config.dev_list (*C var*), 3412
i3c_driver_data (*C struct*), 3412
i3c_driver_data.attached_dev (*C var*), 3412
i3c_driver_data.ctrl_config (*C var*), 3412
i3c_i2c_device_desc (*C struct*), 3410

i3c_i2c_device_desc.addr (*C var*), 3410
i3c_i2c_device_desc.bus (*C var*), 3410
i3c_i2c_device_desc.lvr (*C var*), 3410
i3c_i2c_speed_type (*C enum*), 3396
i3c_i2c_speed_type.I3C_I2C_SPEED_FM (*C enumerator*), 3396
i3c_i2c_speed_type.I3C_I2C_SPEED_FMPLUS (*C enumerator*), 3396
i3c_i2c_speed_type.I3C_I2C_SPEED_INVALID (*C enumerator*), 3397
i3c_i2c_speed_type.I3C_I2C_SPEED_MAX (*C enumerator*), 3396
I3C_LVR_I2C_DEV_IDX (*C macro*), 3395
I3C_LVR_I2C_DEV_IDX_0 (*C macro*), 3395
I3C_LVR_I2C_DEV_IDX_1 (*C macro*), 3395
I3C_LVR_I2C_DEV_IDX_2 (*C macro*), 3395
I3C_LVR_I2C_DEV_IDX_MASK (*C macro*), 3395
I3C_LVR_I2C_DEV_IDX_SHIFT (*C macro*), 3395
I3C_LVR_I2C_FM_MODE (*C macro*), 3395
I3C_LVR_I2C_FM_PLUS_MODE (*C macro*), 3395
I3C_LVR_I2C_MODE (*C macro*), 3395
I3C_LVR_I2C_MODE_MASK (*C macro*), 3395
I3C_LVR_I2C_MODE_SHIFT (*C macro*), 3395
I3C_MAX_ADDR (*C macro*), 3441
i3c_reattach_i3c_device (*C function*), 3402
i3c_recover_bus (*C function*), 3402
i3c_sdr_controller_error_codes (*C enum*), 3397
i3c_sdr_controller_error_codes.I3C_ERROR_CE0 (*C enumerator*), 3397
i3c_sdr_controller_error_codes.I3C_ERROR_CE1 (*C enumerator*), 3397
i3c_sdr_controller_error_codes.I3C_ERROR_CE2 (*C enumerator*), 3397
i3c_sdr_controller_error_codes.I3C_ERROR_CE3 (*C enumerator*), 3397
i3c_sdr_controller_error_codes.I3C_ERROR_CE_INVALID (*C enumerator*), 3398
i3c_sdr_controller_error_codes.I3C_ERROR_CE_MAX (*C enumerator*), 3398
i3c_sdr_controller_error_codes.I3C_ERROR_CE_NONE (*C enumerator*), 3398
i3c_sdr_controller_error_codes.I3C_ERROR_CE_UNKNOWN (*C enumerator*), 3397
i3c_sdr_target_error_codes (*C enum*), 3398
i3c_sdr_target_error_codes.I3C_ERROR_DBR (*C enumerator*), 3398
i3c_sdr_target_error_codes.I3C_ERROR_TE0 (*C enumerator*), 3398
i3c_sdr_target_error_codes.I3C_ERROR_TE1 (*C enumerator*), 3398
i3c_sdr_target_error_codes.I3C_ERROR_TE2 (*C enumerator*), 3398
i3c_sdr_target_error_codes.I3C_ERROR_TE3 (*C enumerator*), 3398
i3c_sdr_target_error_codes.I3C_ERROR_TE4 (*C enumerator*), 3398
i3c_sdr_target_error_codes.I3C_ERROR_TE5 (*C enumerator*), 3398
i3c_sdr_target_error_codes.I3C_ERROR_TE6 (*C enumerator*), 3398
i3c_sdr_target_error_codes.I3C_ERROR_TE_INVALID (*C enumerator*), 3399
i3c_sdr_target_error_codes.I3C_ERROR_TE_MAX (*C enumerator*), 3398
i3c_sdr_target_error_codes.I3C_ERROR_TE_NONE (*C enumerator*), 3398
i3c_sdr_target_error_codes.I3C_ERROR_TE_UNKNOWN (*C enumerator*), 3398
i3c_target_callbacks (*C struct*), 3446
i3c_target_callbacks.read_processed_cb (*C var*), 3447
i3c_target_callbacks.read_requested_cb (*C var*), 3447
i3c_target_callbacks.stop_cb (*C var*), 3447
i3c_target_callbacks.write_received_cb (*C var*), 3446
i3c_target_callbacks.write_requested_cb (*C var*), 3446
i3c_target_config (*C struct*), 3446
i3c_target_config.address (*C var*), 3446
i3c_target_config.callbacks (*C var*), 3446
i3c_target_config.flags (*C var*), 3446
i3c_target_config.node (*C var*), 3446
i3c_target_driver_api (*C struct*), 3448
i3c_target_register (*C function*), 3444
i3c_target_tx_write (*C function*), 3444

i3c_target_unregister (*C function*), 3444
IDENTITY (*C macro*), 690
idle thread, **3936**
IDT, **3936**
ieee802154_attr (*C enum*), 2682
ieee802154_attr_get_channel_page_and_range (*C function*), 2682
ieee802154_attr_value (*C struct*), 2692
ieee802154_attr_value.phy_hrp_uwb_supported_nominal_prfs (*C var*), 2693
ieee802154_attr_value.phy_supported_channel_pages (*C var*), 2692
ieee802154_attr_value.phy_supported_channels (*C var*), 2692
ieee802154_attr.IEEE802154_ATTR_COMMON_COUNT (*C enumerator*), 2682
ieee802154_attr.IEEE802154_ATTR_PHY_HRP_UWB_SUPPORTED_PRFS (*C enumerator*), 2682
ieee802154_attr.IEEE802154_ATTR_PHY_SUPPORTED_CHANNEL_PAGES (*C enumerator*), 2682
ieee802154_attr.IEEE802154_ATTR_PHY_SUPPORTED_CHANNEL_RANGES (*C enumerator*), 2682
ieee802154_attr.IEEE802154_ATTR_PRIV_START (*C enumerator*), 2682
IEEE802154_BROADCAST_ADDRESS (*C macro*), 2704
IEEE802154_BROADCAST_PAN_ID (*C macro*), 2704
ieee802154_config (*C struct*), 2689
IEEE802154_CONFIG_RX_SLOT_NONE (*C macro*), 2683
IEEE802154_CONFIG_RX_SLOT_OFF (*C macro*), 2683
ieee802154_config_type (*C enum*), 2674
ieee802154_config_type.IEEE802154_CONFIG_ACK_FPB (*C enumerator*), 2674
ieee802154_config_type.IEEE802154_CONFIG_AUTO_ACK_FPB (*C enumerator*), 2674
ieee802154_config_type.IEEE802154_CONFIG_COMMON_COUNT (*C enumerator*), 2682
ieee802154_config_type.IEEE802154_CONFIG_CSL_PERIOD (*C enumerator*), 2677
ieee802154_config_type.IEEE802154_CONFIG_ENH_ACK_HEADER_IE (*C enumerator*), 2680
ieee802154_config_type.IEEE802154_CONFIG_EVENT_HANDLER (*C enumerator*), 2675
ieee802154_config_type.IEEE802154_CONFIG_EXPECTED_RX_TIME (*C enumerator*), 2679
ieee802154_config_type.IEEE802154_CONFIG_FRAME_COUNTER (*C enumerator*), 2675
ieee802154_config_type.IEEE802154_CONFIG_FRAME_COUNTER_IF_LARGER (*C enumerator*), 2676
ieee802154_config_type.IEEE802154_CONFIG_MAC_KEYS (*C enumerator*), 2675
ieee802154_config_type.IEEE802154_CONFIG_PAN_COORDINATOR (*C enumerator*), 2674
ieee802154_config_type.IEEE802154_CONFIG_PRIV_START (*C enumerator*), 2682
ieee802154_config_type.IEEE802154_CONFIG_PROMISCUOUS (*C enumerator*), 2675
ieee802154_config_type.IEEE802154_CONFIG_RX_ON_WHEN_IDLE (*C enumerator*), 2681
ieee802154_config_type.IEEE802154_CONFIG_RX_SLOT (*C enumerator*), 2676
ieee802154_config.ack_fpb (*C var*), 2690
ieee802154_config.ack_ie (*C var*), 2692
ieee802154_config.addr (*C var*), 2690
ieee802154_config.auto_ack_fpb (*C var*), 2690
ieee802154_config.channel (*C var*), 2691
ieee802154_config.csl_period (*C var*), 2691
ieee802154_config.duration (*C var*), 2691
ieee802154_config.enabled (*C var*), 2690
ieee802154_config.event_handler (*C var*), 2690
ieee802154_config.expected_rx_time (*C var*), 2691
ieee802154_config.ext_addr (*C var*), 2691
ieee802154_config.extended (*C var*), 2690
ieee802154_config.frame_counter (*C var*), 2690
ieee802154_config.header_ie (*C var*), 2691
ieee802154_config.mac_keys (*C var*), 2690
ieee802154_config.mode (*C var*), 2690
ieee802154_config.pan_coordinator (*C var*), 2690
ieee802154_config.promiscuous (*C var*), 2690
ieee802154_config.purge_ie (*C var*), 2691
ieee802154_config.rx_on_when_idle (*C var*), 2690
ieee802154_config.rx_slot (*C var*), 2691
ieee802154_config.short_addr (*C var*), 2691

ieee802154_config.start (*C var*), 2690
ieee802154_context (*C struct*), 2706
ieee802154_context.ack_lock (*C var*), 2708
ieee802154_context.ack_requested (*C var*), 2707
ieee802154_context.ack_seq (*C var*), 2707
ieee802154_context.channel (*C var*), 2706
ieee802154_context.coord_ext_addr (*C var*), 2707
ieee802154_context.coord_short_addr (*C var*), 2707
ieee802154_context.ctx_lock (*C var*), 2708
ieee802154_context.device_role (*C var*), 2707
ieee802154_context.ext_addr (*C var*), 2706
ieee802154_context.flags (*C var*), 2707
ieee802154_context.linkaddr (*C var*), 2706
ieee802154_context.pan_id (*C var*), 2706
ieee802154_context.scan_ctx (*C var*), 2707
ieee802154_context.scan_ctx_lock (*C var*), 2707
ieee802154_context.sec_ctx (*C var*), 2706
ieee802154_context.sequence (*C var*), 2707
ieee802154_context.short_addr (*C var*), 2706
ieee802154_context.tx_power (*C var*), 2707
IEEE802154_DEFINE_HEADER_IE_CSL_FULL (*C macro*), 2666
IEEE802154_DEFINE_HEADER_IE_CSL_REDUCED (*C macro*), 2666
IEEE802154_DEFINE_HEADER_IE_TIME_CORRECTION (*C macro*), 2667
IEEE802154_DEFINE_HEADER_IE_VENDOR_SPECIFIC (*C macro*), 2666
IEEE802154_DEFINE_PHY_SUPPORTED_CHANNELS (*C macro*), 2669
ieee802154_device_role (*C enum*), 2704
ieee802154_device_role.IEEE802154_DEVICE_ROLE_COORDINATOR (*C enumerator*), 2705
ieee802154_device_role.IEEE802154_DEVICE_ROLE_ENDDEVICE (*C enumerator*), 2705
ieee802154_device_role.IEEE802154_DEVICE_ROLE_PAN_COORDINATOR (*C enumerator*), 2705
ieee802154_event (*C enum*), 2672
ieee802154_event_cb_t (*C type*), 2682
ieee802154_event.IEEE802154_EVENT_RX_FAILED (*C enumerator*), 2672
ieee802154_event.IEEE802154_EVENT_RX_OFF (*C enumerator*), 2672
ieee802154_event.IEEE802154_EVENT_TX_STARTED (*C enumerator*), 2672
IEEE802154_EXT_ADDR_LENGTH (*C macro*), 2704
IEEE802154_FCS_LENGTH (*C macro*), 2703
ieee802154_filter (*C struct*), 2689
ieee802154_filter_type (*C enum*), 2672
ieee802154_filter_type.IEEE802154_FILTER_TYPE_IEEE_ADDR (*C enumerator*), 2672
ieee802154_filter_type.IEEE802154_FILTER_TYPE_PAN_ID (*C enumerator*), 2672
ieee802154_filter_type.IEEE802154_FILTER_TYPE_SHORT_ADDR (*C enumerator*), 2672
ieee802154_filter_type.IEEE802154_FILTER_TYPE_SRC_IEEE_ADDR (*C enumerator*), 2672
ieee802154_filter_type.IEEE802154_FILTER_TYPE_SRC_SHORT_ADDR (*C enumerator*), 2672
ieee802154_filter.ieee_addr (*C var*), 2689
ieee802154_filter.pan_id (*C var*), 2689
ieee802154_filter.short_addr (*C var*), 2689
ieee802154_fpb_mode (*C enum*), 2674
ieee802154_fpb_mode.IEEE802154_FPB_ADDR_MATCH_THREAD (*C enumerator*), 2674
ieee802154_fpb_mode.IEEE802154_FPB_ADDR_MATCH_ZIGBEE (*C enumerator*), 2674
ieee802154_handle_ack (*C function*), 2684
ieee802154_header_ie_csl (*C struct*), 2687
ieee802154_header_ie_csl_full (*C struct*), 2686
ieee802154_header_ie_csl_full.csl_period (*C var*), 2686
ieee802154_header_ie_csl_full.csl_phase (*C var*), 2686
ieee802154_header_ie_csl_full.csl_rendezvous_time (*C var*), 2686
ieee802154_header_ie_csl_reduced (*C struct*), 2686
ieee802154_header_ie_csl_reduced.csl_period (*C var*), 2687
ieee802154_header_ie_csl_reduced.csl_phase (*C var*), 2687

ieee802154_header_ie_csl.full (*C var*), 2687
ieee802154_header_ie_csl.reduced (*C var*), 2687
ieee802154_header_ie_element_id (*C enum*), 2665
ieee802154_header_ie_element_id.IEEE802154_HEADER_IE_ELEMENT_ID_CSL_IE (*C enumerator*), 2665
ieee802154_header_ie_element_id.IEEE802154_HEADER_IE_ELEMENT_ID_HEADER_TERMINATION_1 (*C enumerator*), 2665
ieee802154_header_ie_element_id.IEEE802154_HEADER_IE_ELEMENT_ID_HEADER_TERMINATION_2 (*C enumerator*), 2665
ieee802154_header_ie_element_id.IEEE802154_HEADER_IE_ELEMENT_ID_RENDEZVOUS_TIME_IE (*C enumerator*), 2665
ieee802154_header_ie_element_id.IEEE802154_HEADER_IE_ELEMENT_ID_RIT_IE (*C enumerator*), 2665
ieee802154_header_ie_element_id.IEEE802154_HEADER_IE_ELEMENT_ID_TIME_CORRECTION_IE (*C enumerator*), 2665
ieee802154_header_ie_element_id.IEEE802154_HEADER_IE_ELEMENT_ID_VENDOR_SPECIFIC_IE (*C enumerator*), 2665
ieee802154_header_ie_get_element_id (*C function*), 2665
ieee802154_header_ie_get_time_correction_us (*C function*), 2665
IEEE802154_HEADER_IE_HEADER_LENGTH (*C macro*), 2666
ieee802154_header_ie_rendezvous_time (*C struct*), 2688
ieee802154_header_ie_rendezvous_time_full (*C struct*), 2687
ieee802154_header_ie_rendezvous_time_full.rendezvous_time (*C var*), 2687
ieee802154_header_ie_rendezvous_time_full.wakeup_interval (*C var*), 2687
ieee802154_header_ie_rendezvous_time_reduced (*C struct*), 2687
ieee802154_header_ie_rendezvous_time_reduced.rendezvous_time (*C var*), 2688
ieee802154_header_ie_rendezvous_time.full (*C var*), 2688
ieee802154_header_ie_rendezvous_time.reduced (*C var*), 2688
ieee802154_header_ie_rit (*C struct*), 2687
ieee802154_header_ie_rit.number_of_repeat_listen (*C var*), 2687
ieee802154_header_ie_rit.repeat_listen_interval (*C var*), 2687
ieee802154_header_ie_rit.time_to_first_listen (*C var*), 2687
ieee802154_header_ie_set_element_id (*C function*), 2665
ieee802154_header_ie_time_correction (*C struct*), 2688
ieee802154_header_ie_time_correction.time_sync_info (*C var*), 2688
ieee802154_header_ie_vendor_specific (*C struct*), 2686
ieee802154_header_ie_vendor_specific.vendor_oui (*C var*), 2686
ieee802154_header_ie_vendor_specific.vendor_specific_info (*C var*), 2686
IEEE802154_HEADER_TERMINATION_1_HEADER_IE_LEN (*C macro*), 2667
ieee802154_hw_caps (*C enum*), 2670
IEEE802154_HW_CAPS_BITS_COMMON_COUNT (*C macro*), 2683
IEEE802154_HW_CAPS_BITS_PRIV_START (*C macro*), 2683
ieee802154_hw_caps.IEEE802154_HW_CSMA (*C enumerator*), 2671
ieee802154_hw_caps.IEEE802154_HW_ENERGY_SCAN (*C enumerator*), 2670
ieee802154_hw_caps.IEEE802154_HW_FCS (*C enumerator*), 2670
ieee802154_hw_caps.IEEE802154_HW_FILTER (*C enumerator*), 2670
ieee802154_hw_caps.IEEE802154_HW_PROMISC (*C enumerator*), 2671
ieee802154_hw_caps.IEEE802154_HW_RETRANSMISSION (*C enumerator*), 2671
ieee802154_hw_caps.IEEE802154_HW_RX_TX_ACK (*C enumerator*), 2671
ieee802154_hw_caps.IEEE802154_HW_RXTIME (*C enumerator*), 2671
ieee802154_hw_caps.IEEE802154_HW_SLEEP_TO_TX (*C enumerator*), 2671
ieee802154_hw_caps.IEEE802154_HW_TX_RX_ACK (*C enumerator*), 2671
ieee802154_hw_caps.IEEE802154_HW_TX_SEC (*C enumerator*), 2671
ieee802154_hw_caps.IEEE802154_HW_TXTIME (*C enumerator*), 2671
ieee802154_hw_caps.IEEE802154_RX_ON_WHEN_IDLE (*C enumerator*), 2671
ieee802154_ie_type (*C enum*), 2664
ieee802154_ie_type.IEEE802154_IE_TYPE_HEADER (*C enumerator*), 2664
ieee802154_ie_type.IEEE802154_IE_TYPE_PAYLOAD (*C enumerator*), 2665

ieee802154_init (*C function*), 2684
ieee802154_is_ar_flag_set (*C function*), 2683
ieee802154_key (*C struct*), 2689
ieee802154_key.frame_counter_per_key (*C var*), 2689
ieee802154_key.key_frame_counter (*C var*), 2689
ieee802154_key.key_id (*C var*), 2689
ieee802154_key.key_id_mode (*C var*), 2689
ieee802154_key.key_value (*C var*), 2689
IEEE802154_MAC_A_BASE_SLOT_DURATION (*C macro*), 2685
IEEE802154_MAC_A_BASE_SUPERFRAME_DURATION (*C macro*), 2685
IEEE802154_MAC_A_NUM_SUPERFRAME_SLOTS (*C macro*), 2685
IEEE802154_MAC_A_UNIT_BACKOFF_PERIOD (*C macro*), 2685
IEEE802154_MAC_RESPONSE_WAIT_TIME_DEFAULT (*C macro*), 2685
IEEE802154_MAX_ADDR_LENGTH (*C macro*), 2704
IEEE802154_MAX_PHY_PACKET_SIZE (*C macro*), 2703
IEEE802154_MTU (*C macro*), 2703
IEEE802154_NO_CHANNEL (*C macro*), 2704
IEEE802154_NO_SHORT_ADDRESS_ASSIGNED (*C macro*), 2704
IEEE802154_PAN_ID_NOT_ASSOCIATED (*C macro*), 2704
IEEE802154_PHY_A_CCA_TIME (*C macro*), 2685
IEEE802154_PHY_A_TURNAROUND_TIME_1MS (*C macro*), 2685
IEEE802154_PHY_A_TURNAROUND_TIME_DEFAULT (*C macro*), 2685
IEEE802154_PHY_BPSK_868MHZ_SYMBOL_PERIOD_NS (*C macro*), 2685
IEEE802154_PHY_BPSK_915MHZ_SYMBOL_PERIOD_NS (*C macro*), 2686
ieee802154_phy_channel_page (*C enum*), 2667
ieee802154_phy_channel_page.IEEE802154_ATTR_PHY_CHANNEL_PAGE_EIGHT_LRP_UWB (*C enumerator*), 2668
ieee802154_phy_channel_page.IEEE802154_ATTR_PHY_CHANNEL_PAGE_ELEVEN_OQPSK_2380 (*C enumerator*), 2669
ieee802154_phy_channel_page.IEEE802154_ATTR_PHY_CHANNEL_PAGE_FIVE_OQPSK_780 (*C enumerator*), 2668
ieee802154_phy_channel_page.IEEE802154_ATTR_PHY_CHANNEL_PAGE_FOUR_HRP_UWB (*C enumerator*), 2668
ieee802154_phy_channel_page.IEEE802154_ATTR_PHY_CHANNEL_PAGE_NINE_SUN_PREDEFINED (*C enumerator*), 2668
ieee802154_phy_channel_page.IEEE802154_ATTR_PHY_CHANNEL_PAGE_ONE_DEPRECATED (*C enumerator*), 2668
ieee802154_phy_channel_page.IEEE802154_ATTR_PHY_CHANNEL_PAGE_SEVEN_MSK (*C enumerator*), 2668
ieee802154_phy_channel_page.IEEE802154_ATTR_PHY_CHANNEL_PAGE_SIX_RESERVED (*C enumerator*), 2668
ieee802154_phy_channel_page.IEEE802154_ATTR_PHY_CHANNEL_PAGE_TEN_SUN_FSK_GENERIC (*C enumerator*), 2668
ieee802154_phy_channel_page.IEEE802154_ATTR_PHY_CHANNEL_PAGE_THIRTEEN_RCC (*C enumerator*), 2669
ieee802154_phy_channel_page.IEEE802154_ATTR_PHY_CHANNEL_PAGE_THREE_CSS (*C enumerator*), 2668
ieee802154_phy_channel_page.IEEE802154_ATTR_PHY_CHANNEL_PAGE_TWELVE_LECIM (*C enumerator*), 2669
ieee802154_phy_channel_page.IEEE802154_ATTR_PHY_CHANNEL_PAGE_TWO_OQPSK_868_915 (*C enumerator*), 2668
ieee802154_phy_channel_page.IEEE802154_ATTR_PHY_CHANNEL_PAGE_ZERO_OQPSK_2450_BPSK_868_915 (*C enumerator*), 2668
ieee802154_phy_channel_range (*C struct*), 2688
ieee802154_phy_channel_range.from_channel (*C var*), 2688
ieee802154_phy_channel_range.to_channel (*C var*), 2688
IEEE802154_PHY_HRP_UWB_ERDEV (*C macro*), 2670
IEEE802154_PHY_HRP_UWB_ERDEV_TPSYM_SYMBOL_PERIOD_NS (*C macro*), 2670

ieee802154_phy_hrp_uwb_nominal_prf (*C enum*), 2669
ieee802154_phy_hrp_uwb_nominal_prf.IEEE802154_PHY_HRP_UWB_NOMINAL_4_M (*C enumerator*), 2669
ieee802154_phy_hrp_uwb_nominal_prf.IEEE802154_PHY_HRP_UWB_NOMINAL_16_M (*C enumerator*), 2669
ieee802154_phy_hrp_uwb_nominal_prf.IEEE802154_PHY_HRP_UWB_NOMINAL_64_M (*C enumerator*), 2670
ieee802154_phy_hrp_uwb_nominal_prf.IEEE802154_PHY_HRP_UWB_NOMINAL_64_M_BPRF (*C enumerator*), 2670
ieee802154_phy_hrp_uwb_nominal_prf.IEEE802154_PHY_HRP_UWB_NOMINAL_128_M_HPRF (*C enumerator*), 2670
ieee802154_phy_hrp_uwb_nominal_prf.IEEE802154_PHY_HRP_UWB_NOMINAL_256_M_HPRF (*C enumerator*), 2670
ieee802154_phy_hrp_uwb_nominal_prf.IEEE802154_PHY_HRP_UWB_PRF_OFF (*C enumerator*), 2669
IEEE802154_PHY_HRP_UWB_PRF4_TPSYM_SYMBOL_PERIOD_NS (*C macro*), 2670
IEEE802154_PHY_HRP_UWB_PRF16_TPSYM_SYMBOL_PERIOD_NS (*C macro*), 2670
IEEE802154_PHY_HRP_UWB_PRF64_TPSYM_SYMBOL_PERIOD_NS (*C macro*), 2670
IEEE802154_PHY_HRP_UWB_RDEV (*C macro*), 2670
IEEE802154_PHY_OQPSK_780_TO_2450MHZ_SYMBOL_PERIOD_NS (*C macro*), 2685
IEEE802154_PHY_OQPSK_868MHZ_SYMBOL_PERIOD_NS (*C macro*), 2685
IEEE802154_PHY_SUN_FSK_863MHZ_915MHZ_SYMBOL_PERIOD_NS (*C macro*), 2686
IEEE802154_PHY_SUN_FSK_PHR_LEN (*C macro*), 2686
ieee802154_phy_supported_channels (*C struct*), 2688
ieee802154_phy_supported_channels.num_ranges (*C var*), 2689
ieee802154_phy_supported_channels.ranges (*C var*), 2689
IEEE802154_PHY_SYMBOLS_PER_SECOND (*C macro*), 2684
ieee802154_radio_api (*C struct*), 2693
ieee802154_radio_api.attr_get (*C var*), 2702
ieee802154_radio_api.cca (*C var*), 2695
ieee802154_radio_api.configure (*C var*), 2700
ieee802154_radio_api.continuous_carrier (*C var*), 2699
ieee802154_radio_api.ed_scan (*C var*), 2701
ieee802154_radio_api.filter (*C var*), 2696
ieee802154_radio_api.get_capabilities (*C var*), 2695
ieee802154_radio_api.get_sch_acc (*C var*), 2702
ieee802154_radio_api.get_time (*C var*), 2701
ieee802154_radio_api.iface_api (*C var*), 2695
ieee802154_radio_api.set_channel (*C var*), 2695
ieee802154_radio_api.set_txpower (*C var*), 2697
ieee802154_radio_api.start (*C var*), 2698
ieee802154_radio_api.stop (*C var*), 2699
ieee802154_radio_api.tx (*C var*), 2697
ieee802154_req_params (*C struct*), 2663
ieee802154_req_params.addr (*C var*), 2663
ieee802154_req_params.channel (*C var*), 2663
ieee802154_req_params.channel_set (*C var*), 2663
ieee802154_req_params.duration (*C var*), 2663
ieee802154_req_params.len (*C var*), 2663
ieee802154_req_params.lqi (*C var*), 2663
ieee802154_req_params.pan_id (*C var*), 2663
ieee802154_req_params.short_addr (*C var*), 2663
ieee802154_rx_fail_reason (*C enum*), 2672
ieee802154_rx_fail_reason.IEEE802154_RX_FAIL_ADDR_FILTERED (*C enumerator*), 2672
ieee802154_rx_fail_reason.IEEE802154_RX_FAIL_INVALID_FCS (*C enumerator*), 2672
ieee802154_rx_fail_reason.IEEE802154_RX_FAIL_NOT_RECEIVED (*C enumerator*), 2672
ieee802154_rx_fail_reason.IEEE802154_RX_FAIL_OTHER (*C enumerator*), 2673
ieee802154_security_ctx (*C struct*), 2705
ieee802154_security_ctx.frame_counter (*C var*), 2705

ieee802154_security_ctx.key (*C var*), 2705
ieee802154_security_ctx.key_len (*C var*), 2705
ieee802154_security_ctx.key_mode (*C var*), 2705
ieee802154_security_ctx.level (*C var*), 2705
ieee802154_security_params (*C struct*), 2663
ieee802154_security_params.key (*C var*), 2663
ieee802154_security_params.key_len (*C var*), 2663
ieee802154_security_params.key_mode (*C var*), 2664
ieee802154_security_params.level (*C var*), 2664
IEEE802154_SHORT_ADDR_LENGTH (*C macro*), 2704
IEEE802154_SHORT_ADDRESS_NOT_ASSOCIATED (*C macro*), 2704
IEEE802154_TIME_CORRECTION_HEADER_IE_LEN (*C macro*), 2667
ieee802154_tx_mode (*C enum*), 2673
ieee802154_tx_mode.IEEE802154_TX_MODE_CCA (*C enumerator*), 2673
ieee802154_tx_mode.IEEE802154_TX_MODE_COMMON_COUNT (*C enumerator*), 2673
ieee802154_tx_mode.IEEE802154_TX_MODE_CSMA_CA (*C enumerator*), 2673
ieee802154_tx_mode.IEEE802154_TX_MODE_DIRECT (*C enumerator*), 2673
ieee802154_tx_mode.IEEE802154_TX_MODE_PRIV_START (*C enumerator*), 2673
ieee802154_tx_mode.IEEE802154_TX_MODE_TXTIME (*C enumerator*), 2673
ieee802154_tx_mode.IEEE802154_TX_MODE_TXTIME_CCA (*C enumerator*), 2673
IF_DISABLED (*C macro*), 689
IF_ENABLED (*C macro*), 688
IFNAMSIZ (*C macro*), 2493
ifreq (*C struct*), 2502
ifreq.ifr_name (*C var*), 2502
img_mgmt_group_events (*C enum*), 770
img_mgmt_group_events.MGMT_EVT_OP_IMG_MGMT_ALL (*C enumerator*), 771
img_mgmt_group_events.MGMT_EVT_OP_IMG_MGMT_DFU_CHUNK (*C enumerator*), 770
img_mgmt_group_events.MGMT_EVT_OP_IMG_MGMT_DFU_CHUNK_WRITE_COMPLETE (*C enumerator*), 771
img_mgmt_group_events.MGMT_EVT_OP_IMG_MGMT_DFU_CONFIRMED (*C enumerator*), 770
img_mgmt_group_events.MGMT_EVT_OP_IMG_MGMT_DFU_PENDING (*C enumerator*), 770
img_mgmt_group_events.MGMT_EVT_OP_IMG_MGMT_DFU_STARTED (*C enumerator*), 770
img_mgmt_group_events.MGMT_EVT_OP_IMG_MGMT_DFU_STOPPED (*C enumerator*), 770
in6_addr (*C struct*), 2525
in6_addr.s6_addr (*C var*), 2525
in6_addr.s6_addr16 (*C var*), 2525
in6_addr.s6_addr32 (*C var*), 2525
in6_pktinfo (*C struct*), 2503
in6_pktinfo.ipi6_addr (*C var*), 2503
in6_pktinfo.ipi6_ifindex (*C var*), 2503
IN6ADDR_ANY_INIT (*C macro*), 2509
IN6ADDR_LOOPBACK_INIT (*C macro*), 2509
in_addr (*C struct*), 2525
in_addr.s4_addr (*C var*), 2525
in_addr.s4_addr16 (*C var*), 2525
in_addr.s4_addr32 (*C var*), 2526
in_addr.s_addr (*C var*), 2526
in_pktinfo (*C struct*), 2502
in_pktinfo.ipi_addr (*C var*), 2502
in_pktinfo.ipi_ifindex (*C var*), 2502
in_pktinfo.ipi_spec_dst (*C var*), 2502
IN_RANGE (*C macro*), 684
INADDR_ANY (*C macro*), 2509
INADDR_ANY_INIT (*C macro*), 2510
INADDR_LOOPBACK_INIT (*C macro*), 2510
INET6_ADDRSTRLEN (*C macro*), 2510
INET_ADDRSTRLEN (*C macro*), 2510

inet_ntop (*C function*), 2489
inet_pton (*C function*), 2489
INPUT_ABS_BRAKE (*C macro*), 897
INPUT_ABS_GAS (*C macro*), 897
INPUT_ABS_MT_SLOT (*C macro*), 897
INPUT_ABS_RUDDER (*C macro*), 897
INPUT_ABS_RX (*C macro*), 897
INPUT_ABS_RY (*C macro*), 897
INPUT_ABS_RZ (*C macro*), 897
INPUT_ABS_THROTTLE (*C macro*), 897
INPUT_ABS_WHEEL (*C macro*), 897
INPUT_ABS_X (*C macro*), 897
INPUT_ABS_Y (*C macro*), 897
INPUT_ABS_Z (*C macro*), 898
INPUT_BTN_0 (*C macro*), 894
INPUT_BTN_1 (*C macro*), 894
INPUT_BTN_2 (*C macro*), 894
INPUT_BTN_3 (*C macro*), 894
INPUT_BTN_4 (*C macro*), 894
INPUT_BTN_5 (*C macro*), 894
INPUT_BTN_6 (*C macro*), 894
INPUT_BTN_7 (*C macro*), 894
INPUT_BTN_8 (*C macro*), 895
INPUT_BTN_9 (*C macro*), 895
INPUT_BTN_A (*C macro*), 895
INPUT_BTN_B (*C macro*), 895
INPUT_BTN_BACK (*C macro*), 895
INPUT_BTN_C (*C macro*), 895
INPUT_BTN_DPAD_DOWN (*C macro*), 895
INPUT_BTN_DPAD_LEFT (*C macro*), 895
INPUT_BTN_DPAD_RIGHT (*C macro*), 895
INPUT_BTN_DPAD_UP (*C macro*), 895
INPUT_BTN_EAST (*C macro*), 895
INPUT_BTN_EXTRA (*C macro*), 895
INPUT_BTN_FORWARD (*C macro*), 895
INPUT_BTN_GEAR_DOWN (*C macro*), 895
INPUT_BTN_GEAR_UP (*C macro*), 895
INPUT_BTN_LEFT (*C macro*), 895
INPUT_BTN_MIDDLE (*C macro*), 896
INPUT_BTN_MODE (*C macro*), 896
INPUT_BTN_NORTH (*C macro*), 896
INPUT_BTN_RIGHT (*C macro*), 896
INPUT_BTN_SELECT (*C macro*), 896
INPUT_BTN_SIDE (*C macro*), 896
INPUT_BTN_SOUTH (*C macro*), 896
INPUT_BTN_START (*C macro*), 896
INPUT_BTN_TASK (*C macro*), 896
INPUT_BTN_THUMBL (*C macro*), 896
INPUT_BTN_THUMBR (*C macro*), 896
INPUT_BTN_TL (*C macro*), 896
INPUT_BTN_TL2 (*C macro*), 896
INPUT_BTN_TOUCH (*C macro*), 896
INPUT_BTN_TR (*C macro*), 896
INPUT_BTN_TR2 (*C macro*), 896
INPUT_BTN_WEST (*C macro*), 897
INPUT_BTN_X (*C macro*), 897
INPUT_BTN_Y (*C macro*), 897
INPUT_BTN_Z (*C macro*), 897

input_callback (*C struct*), 884
INPUT_CALLBACK_DEFINE (*C macro*), 882
input_callback.callback (*C var*), 884
input_callback.dev (*C var*), 884
INPUT_EV_ABS (*C macro*), 885
INPUT_EV_KEY (*C macro*), 885
INPUT_EV_MSC (*C macro*), 885
INPUT_EV_REL (*C macro*), 885
INPUT_EV_VENDOR_START (*C macro*), 885
INPUT_EV_VENDOR_STOP (*C macro*), 885
input_event (*C struct*), 884
input_event.code (*C var*), 884
input_event.dev (*C var*), 884
input_event.sync (*C var*), 884
input_event.type (*C var*), 884
input_event.value (*C var*), 884
INPUT_KBD_ACTUAL_KEY_MASK_CONST (*C macro*), 878
input_kbd_matrix_actual_key_mask_set (*C function*), 880
input_kbd_matrix_api (*C struct*), 881
input_kbd_matrix_api.drive_column (*C var*), 881
input_kbd_matrix_api.read_row (*C var*), 881
input_kbd_matrix_api.set_detect_mode (*C var*), 881
INPUT_KBD_MATRIX_COLUMN_DRIVE_ALL (*C macro*), 878
INPUT_KBD_MATRIX_COLUMN_DRIVE_NONE (*C macro*), 878
input_kbd_matrix_common_config (*C struct*), 881
input_kbd_matrix_common_data (*C struct*), 881
input_kbd_matrix_common_init (*C function*), 880
INPUT_KBD_MATRIX_DATA_NAME (*C macro*), 878
input_kbd_matrix_drive_column_hook (*C function*), 880
INPUT_KBD_MATRIX_DT_COMMON_CONFIG_INIT (*C macro*), 879
INPUT_KBD_MATRIX_DT_COMMON_CONFIG_INIT_ROW_COL (*C macro*), 879
INPUT_KBD_MATRIX_DT_DEFINE (*C macro*), 879
INPUT_KBD_MATRIX_DT_DEFINE_ROW_COL (*C macro*), 878
INPUT_KBD_MATRIX_DT_INST_COMMON_CONFIG_INIT (*C macro*), 879
INPUT_KBD_MATRIX_DT_INST_COMMON_CONFIG_INIT_ROW_COL (*C macro*), 879
INPUT_KBD_MATRIX_DT_INST_DEFINE (*C macro*), 879
INPUT_KBD_MATRIX_DT_INST_DEFINE_ROW_COL (*C macro*), 879
input_kbd_matrix_poll_start (*C function*), 880
INPUT_KBD_MATRIX_ROW_BITS (*C macro*), 878
INPUT_KBD_MATRIX_SCAN_OCURRENCES (*C macro*), 878
INPUT_KBD_STRUCT_CHECK (*C macro*), 879
INPUT_KEY_0 (*C macro*), 885
INPUT_KEY_1 (*C macro*), 885
INPUT_KEY_2 (*C macro*), 885
INPUT_KEY_3 (*C macro*), 885
INPUT_KEY_4 (*C macro*), 886
INPUT_KEY_5 (*C macro*), 886
INPUT_KEY_6 (*C macro*), 886
INPUT_KEY_7 (*C macro*), 886
INPUT_KEY_8 (*C macro*), 886
INPUT_KEY_9 (*C macro*), 886
INPUT_KEY_A (*C macro*), 886
INPUT_KEY_APOSTROPHE (*C macro*), 886
INPUT_KEY_B (*C macro*), 886
INPUT_KEY_BACK (*C macro*), 886
INPUT_KEY_BACKSLASH (*C macro*), 886
INPUT_KEY_BACKSPACE (*C macro*), 886
INPUT_KEY_BLUETOOTH (*C macro*), 886

INPUT_KEY_BRIGHTNESSDOWN (*C macro*), 886
INPUT_KEY_BRIGHTNESSUP (*C macro*), 886
INPUT_KEY_C (*C macro*), 886
INPUT_KEY_CAPSLOCK (*C macro*), 887
INPUT_KEY_COFFEE (*C macro*), 887
INPUT_KEY_COMMA (*C macro*), 887
INPUT_KEY_COMPOSE (*C macro*), 887
INPUT_KEY_CONNECT (*C macro*), 887
INPUT_KEY_D (*C macro*), 887
INPUT_KEY_DELETE (*C macro*), 887
INPUT_KEY_DOT (*C macro*), 887
INPUT_KEY_DOWN (*C macro*), 887
INPUT_KEY_E (*C macro*), 887
INPUT_KEY_END (*C macro*), 887
INPUT_KEY_ENTER (*C macro*), 887
INPUT_KEY_EQUAL (*C macro*), 887
INPUT_KEY_ESC (*C macro*), 887
INPUT_KEY_F (*C macro*), 887
INPUT_KEY_F1 (*C macro*), 887
INPUT_KEY_F2 (*C macro*), 888
INPUT_KEY_F3 (*C macro*), 889
INPUT_KEY_F4 (*C macro*), 889
INPUT_KEY_F5 (*C macro*), 889
INPUT_KEY_F6 (*C macro*), 889
INPUT_KEY_F7 (*C macro*), 889
INPUT_KEY_F8 (*C macro*), 889
INPUT_KEY_F9 (*C macro*), 889
INPUT_KEY_F10 (*C macro*), 888
INPUT_KEY_F11 (*C macro*), 888
INPUT_KEY_F12 (*C macro*), 888
INPUT_KEY_F13 (*C macro*), 888
INPUT_KEY_F14 (*C macro*), 888
INPUT_KEY_F15 (*C macro*), 888
INPUT_KEY_F16 (*C macro*), 888
INPUT_KEY_F17 (*C macro*), 888
INPUT_KEY_F18 (*C macro*), 888
INPUT_KEY_F19 (*C macro*), 888
INPUT_KEY_F20 (*C macro*), 888
INPUT_KEY_F21 (*C macro*), 888
INPUT_KEY_F22 (*C macro*), 888
INPUT_KEY_F23 (*C macro*), 888
INPUT_KEY_F24 (*C macro*), 888
INPUT_KEY_FASTFORWARD (*C macro*), 889
INPUT_KEY_FORWARD (*C macro*), 889
INPUT_KEY_G (*C macro*), 889
INPUT_KEY_GRAVE (*C macro*), 889
INPUT_KEY_H (*C macro*), 889
INPUT_KEY_HOME (*C macro*), 889
INPUT_KEY_I (*C macro*), 889
INPUT_KEY_INSERT (*C macro*), 889
INPUT_KEY_J (*C macro*), 889
INPUT_KEY_K (*C macro*), 890
INPUT_KEY_KP0 (*C macro*), 890
INPUT_KEY_KP1 (*C macro*), 890
INPUT_KEY_KP2 (*C macro*), 890
INPUT_KEY_KP3 (*C macro*), 890
INPUT_KEY_KP4 (*C macro*), 890
INPUT_KEY_KP5 (*C macro*), 890

INPUT_KEY_KP6 (*C macro*), 890
INPUT_KEY_KP7 (*C macro*), 890
INPUT_KEY_KP8 (*C macro*), 890
INPUT_KEY_KP9 (*C macro*), 890
INPUT_KEY_KPASTERISK (*C macro*), 890
INPUT_KEY_KPCOMMA (*C macro*), 890
INPUT_KEY_KPDOT (*C macro*), 890
INPUT_KEY_KPENTER (*C macro*), 890
INPUT_KEY_KPEQUAL (*C macro*), 890
INPUT_KEY_KPMINUS (*C macro*), 891
INPUT_KEY_KPPLUS (*C macro*), 891
INPUT_KEY_KPPLUSMINUS (*C macro*), 891
INPUT_KEY_KPSLASH (*C macro*), 891
INPUT_KEY_L (*C macro*), 891
INPUT_KEY_LEFT (*C macro*), 891
INPUT_KEY_LEFTALT (*C macro*), 891
INPUT_KEY_LEFTBRACE (*C macro*), 891
INPUT_KEY_LEFTCTRL (*C macro*), 891
INPUT_KEY_LEFTMETA (*C macro*), 891
INPUT_KEY_LEFTSHIFT (*C macro*), 891
INPUT_KEY_M (*C macro*), 891
INPUT_KEY_MENU (*C macro*), 891
INPUT_KEY_MINUS (*C macro*), 891
INPUT_KEY_MUTE (*C macro*), 891
INPUT_KEY_N (*C macro*), 891
INPUT_KEY_NUMLOCK (*C macro*), 892
INPUT_KEY_O (*C macro*), 892
INPUT_KEY_P (*C macro*), 892
INPUT_KEY_PAGEDOWN (*C macro*), 892
INPUT_KEY_PAGEUP (*C macro*), 892
INPUT_KEY_PAUSE (*C macro*), 892
INPUT_KEY_PLAY (*C macro*), 892
INPUT_KEY_POWER (*C macro*), 892
INPUT_KEY_PRINT (*C macro*), 892
INPUT_KEY_Q (*C macro*), 892
INPUT_KEY_R (*C macro*), 892
INPUT_KEY_RESERVED (*C macro*), 885
INPUT_KEY_RIGHT (*C macro*), 892
INPUT_KEY_RIGHTALT (*C macro*), 892
INPUT_KEY_RIGHTBRACE (*C macro*), 892
INPUT_KEY_RIGHTCTRL (*C macro*), 892
INPUT_KEY_RIGHTMETA (*C macro*), 892
INPUT_KEY_RIGHTSHIFT (*C macro*), 893
INPUT_KEY_S (*C macro*), 893
INPUT_KEY_SCALE (*C macro*), 893
INPUT_KEY_SCROLLLOCK (*C macro*), 893
INPUT_KEY_SEMICOLON (*C macro*), 893
INPUT_KEY_SLASH (*C macro*), 893
INPUT_KEY_SLEEP (*C macro*), 893
INPUT_KEY_SPACE (*C macro*), 893
INPUT_KEY_SYSRQ (*C macro*), 893
INPUT_KEY_T (*C macro*), 893
INPUT_KEY_TAB (*C macro*), 893
INPUT_KEY_U (*C macro*), 893
INPUT_KEY_UP (*C macro*), 893
INPUT_KEY_UWB (*C macro*), 893
INPUT_KEY_V (*C macro*), 893
INPUT_KEY_VOLUMEDOWN (*C macro*), 893

INPUT_KEY_VOLUMEUP (*C macro*), 894
INPUT_KEY_W (*C macro*), 894
INPUT_KEY_WAKEUP (*C macro*), 894
INPUT_KEY_WLAN (*C macro*), 894
INPUT_KEY_X (*C macro*), 894
INPUT_KEY_Y (*C macro*), 894
INPUT_KEY_Z (*C macro*), 894
INPUT_MSC_SCAN (*C macro*), 898
input_queue_empty (*C function*), 883
INPUT_REL_DIAL (*C macro*), 898
INPUT_REL_HWHEEL (*C macro*), 898
INPUT_REL_MISC (*C macro*), 898
INPUT_REL_RX (*C macro*), 898
INPUT_REL_RY (*C macro*), 898
INPUT_REL_RZ (*C macro*), 898
INPUT_REL_WHEEL (*C macro*), 898
INPUT_REL_X (*C macro*), 898
INPUT_REL_Y (*C macro*), 898
INPUT_REL_Z (*C macro*), 898
input_report (*C function*), 882
input_report_abs (*C function*), 883
input_report_key (*C function*), 883
input_report_rel (*C function*), 883
input_to_hid_code (*C function*), 883
input_to_hid_modifier (*C function*), 884
INT_TO_POINTER (*C macro*), 679
iovec (*C struct*), 2527
iovec.iov_base (*C var*), 2527
iovec.iov_len (*C var*), 2527
IP_ADD_MEMBERSHIP (*C macro*), 2495
IP_DROP_MEMBERSHIP (*C macro*), 2496
ip_mreqn (*C struct*), 2503
ip_mreqn.imr_address (*C var*), 2503
ip_mreqn.imr_ifindex (*C var*), 2503
ip_mreqn.imr_multiaddr (*C var*), 2503
IP_MULTICAST_TTL (*C macro*), 2495
IP_PKTINFO (*C macro*), 2495
IP_TOS (*C macro*), 2495
IP_TTL (*C macro*), 2495
ipc_ept (*C struct*), 912
ipc_ept_cfg (*C struct*), 912
ipc_ept_cfg.cb (*C var*), 912
ipc_ept_cfg.name (*C var*), 912
ipc_ept_cfg.prio (*C var*), 912
ipc_ept_cfg.priv (*C var*), 913
ipc_ept.instance (*C var*), 912
ipc_ept.token (*C var*), 912
ipc_service_backend (*C struct*), 913
ipc_service_backend.close_instance (*C var*), 913
ipc_service_backend.deregister_endpoint (*C var*), 914
ipc_service_backend.drop_tx_buffer (*C var*), 915
ipc_service_backend.get_tx_buffer (*C var*), 915
ipc_service_backend.get_tx_buffer_size (*C var*), 914
ipc_service_backend.hold_rx_buffer (*C var*), 916
ipc_service_backend.open_instance (*C var*), 913
ipc_service_backend.register_endpoint (*C var*), 914
ipc_service_backend.release_rx_buffer (*C var*), 916
ipc_service_backend.send (*C var*), 913

ipc_service_backend.send_nocopy (*C var*), 916
ipc_service_cb (*C struct*), 911
ipc_service_cb.bound (*C var*), 911
ipc_service_cb.error (*C var*), 912
ipc_service_cb.received (*C var*), 911
ipc_service_close_instance (*C function*), 906
ipc_service_deregister_endpoint (*C function*), 907
ipc_service_drop_tx_buffer (*C function*), 909
ipc_service_get_tx_buffer (*C function*), 908
ipc_service_get_tx_buffer_size (*C function*), 908
ipc_service_hold_rx_buffer (*C function*), 910
ipc_service_open_instance (*C function*), 906
ipc_service_register_endpoint (*C function*), 907
ipc_service_release_rx_buffer (*C function*), 911
ipc_service_send (*C function*), 907
ipc_service_send_nocopy (*C function*), 910
ipm_callback_t (*C type*), 3469
ipm_complete (*C function*), 3471
ipm_complete_t (*C type*), 3470
ipm_driver_api (*C struct*), 3471
ipm_max_data_size_get (*C function*), 3471
ipm_max_data_size_get_t (*C type*), 3469
ipm_max_id_val_get (*C function*), 3471
ipm_max_id_val_get_t (*C type*), 3469
ipm_register_callback (*C function*), 3470
ipm_register_callback_t (*C type*), 3469
ipm_send (*C function*), 3470
ipm_send_t (*C type*), 3469
ipm_set_enabled (*C function*), 3471
ipm_set_enabled_t (*C type*), 3469
IPSO_OBJECT_ACCELEROMETER_ID (*C macro*), 2822
IPSO_OBJECT_BUZZER_ID (*C macro*), 2822
IPSO_OBJECT_CURRENT_SENSOR_ID (*C macro*), 2822
IPSO_OBJECT_FILLING_LEVEL_SENSOR_ID (*C macro*), 2822
IPSO_OBJECT_GENERIC_SENSOR_ID (*C macro*), 2822
IPSO_OBJECT_HUMIDITY_SENSOR_ID (*C macro*), 2822
IPSO_OBJECT_LIGHT_CONTROL_ID (*C macro*), 2822
IPSO_OBJECT_ONOFF_SWITCH_ID (*C macro*), 2822
IPSO_OBJECT_PRESSURE_ID (*C macro*), 2822
IPSO_OBJECT_PUSH_BUTTON_ID (*C macro*), 2822
IPSO_OBJECT_TEMP_SENSOR_ID (*C macro*), 2822
IPSO_OBJECT_TIMER_ID (*C macro*), 2822
IPSO_OBJECT_VOLTAGE_SENSOR_ID (*C macro*), 2822
IPV6_ADD_MEMBERSHIP (*C macro*), 2496
IPV6_ADDR_PREFERENCES (*C macro*), 2496
IPV6_DROP_MEMBERSHIP (*C macro*), 2496
ipv6_mreq (*C struct*), 2503
ipv6_mreq.ipv6mr_ifindex (*C var*), 2503
ipv6_mreq.ipv6mr_multiaddr (*C var*), 2503
IPV6_MULTICAST_HOPS (*C macro*), 2496
IPV6_PREFER_SRC_CGA (*C macro*), 2497
IPV6_PREFER_SRC_COA (*C macro*), 2496
IPV6_PREFER_SRC_HOME (*C macro*), 2496
IPV6_PREFER_SRC_NONCGA (*C macro*), 2497
IPV6_PREFER_SRC_PUBLIC (*C macro*), 2496
IPV6_PREFER_SRC_PUBTMP_DEFAULT (*C macro*), 2496
IPV6_PREFER_SRC_TMP (*C macro*), 2496
IPV6_RECVPKTINFO (*C macro*), 2496

IPV6_TCLASS (*C macro*), 2497
IPV6_UNICAST_HOPS (*C macro*), 2496
IPV6_V6ONLY (*C macro*), 2496
IRQ_CONNECT (*C macro*), 384
irq_connect_dynamic (*C function*), 388
IRQ_DIRECT_CONNECT (*C macro*), 385
irq_disable (*C macro*), 388
irq_disconnect_dynamic (*C function*), 388
irq_enable (*C macro*), 388
irq_is_enabled (*C macro*), 388
irq_lock (*C macro*), 386
irq_unlock (*C macro*), 387
IS_ALIGNED (*C macro*), 682
IS_ARRAY (*C macro*), 680
IS_ARRAY_ELEMENT (*C macro*), 680
IS_BIT_MASK (*C macro*), 686
IS_EMPTY (*C macro*), 689
IS_ENABLED (*C macro*), 686
IS_EQ (*C macro*), 689
is_null_no_warn (*C function*), 696
is_power_of_two (*C function*), 696
IS_POWER_OF_TWO (*C macro*), 686
IS_SHIFTED_BIT_MASK (*C macro*), 686
isotp_bind (*C function*), 2445
isotp_fc_opts (*C struct*), 2448
isotp_fc_opts.bs (*C var*), 2448
isotp_fc_opts.stmin (*C var*), 2448
ISOTP_FIXED_ADDR_PRIO_MASK (*C macro*), 2445
ISOTP_FIXED_ADDR_PRIO_POS (*C macro*), 2445
ISOTP_FIXED_ADDR_RX_MASK (*C macro*), 2445
ISOTP_FIXED_ADDR_SA_MASK (*C macro*), 2445
ISOTP_FIXED_ADDR_SA_POS (*C macro*), 2445
ISOTP_FIXED_ADDR_TA_MASK (*C macro*), 2445
ISOTP_FIXED_ADDR_TA_POS (*C macro*), 2445
ISOTP_MSG_BRS (*C macro*), 2444
ISOTP_MSG_EXT_ADDR (*C macro*), 2443
ISOTP_MSG_FDF (*C macro*), 2444
ISOTP_MSG_FIXED_ADDR (*C macro*), 2443
isotp_msg_id (*C struct*), 2447
isotp_msg_id.dl (*C var*), 2448
ISOTP_MSG_IDE (*C macro*), 2443
isotp_msg_id.ext_addr (*C var*), 2448
isotp_msg_id.flags (*C var*), 2448
ISOTP_N_BUFFER_OVERFLW (*C macro*), 2444
ISOTP_N_ERROR (*C macro*), 2444
ISOTP_N_INVALID_FS (*C macro*), 2444
ISOTP_N_OK (*C macro*), 2444
ISOTP_N_TIMEOUT_A (*C macro*), 2444
ISOTP_N_TIMEOUT_BS (*C macro*), 2444
ISOTP_N_TIMEOUT_CR (*C macro*), 2444
ISOTP_N_UNEXP_PDU (*C macro*), 2444
ISOTP_N_WFT_OVRN (*C macro*), 2444
ISOTP_N_WRONG_SN (*C macro*), 2444
ISOTP_NO_BUF_DATA_LEFT (*C macro*), 2445
ISOTP_NO_CTX_LEFT (*C macro*), 2445
ISOTP_NO_FREE_FILTER (*C macro*), 2444
ISOTP_NO_NET_BUF_LEFT (*C macro*), 2444
isotp_recv (*C function*), 2446

isotp_recv_net (*C function*), 2446
ISOTP_RECV_TIMEOUT (*C macro*), 2445
isotp_send (*C function*), 2447
isotp_tx_callback_t (*C type*), 2445
isotp_unbind (*C function*), 2446
ISR, **3936**
ISR_DIRECT_DECLARE (*C macro*), 386
ISR_DIRECT_FOOTER (*C macro*), 385
ISR_DIRECT_HEADER (*C macro*), 385
ISR_DIRECT_PM (*C macro*), 386
ITERABLE_SECTION_RAM (*C macro*), 701
ITERABLE_SECTION_RAM_GC_ALLOWED (*C macro*), 701
ITERABLE_SECTION_RAM_NUMERIC (*C macro*), 701
ITERABLE_SECTION_ROM (*C macro*), 700
ITERABLE_SECTION_ROM_GC_ALLOWED (*C macro*), 701
ITERABLE_SECTION_ROM_NUMERIC (*C macro*), 700
ivshmem_driver_api (*C struct*), 1234
ivshmem_get_id (*C function*), 1233
ivshmem_get_id_f (*C type*), 1233
ivshmem_get_mem (*C function*), 1233
ivshmem_get_mem_f (*C type*), 1233
ivshmem_get_vectors (*C function*), 1234
ivshmem_get_vectors_f (*C type*), 1233
ivshmem_int_peer (*C function*), 1234
ivshmem_int_peer_f (*C type*), 1233
ivshmem_register_handler (*C function*), 1234
ivshmem_register_handler_f (*C type*), 1233
IVSHMEM_V2_PROTO_NET (*C macro*), 1233
IVSHMEM_V2_PROTO_UNDEFINED (*C macro*), 1233

J

json_append_bytes_t (*C type*), 1300
json_arr_encode (*C function*), 1304
json_arr_encode_buf (*C function*), 1304
json_arr_parse (*C function*), 1301
json_arr_separate_object_parse_init (*C function*), 1302
json_arr_separate_parse_object (*C function*), 1302
json_calc_encoded_arr_len (*C function*), 1303
json_calc_encoded_len (*C function*), 1303
json_calc_escaped_len (*C function*), 1303
json_escape (*C function*), 1302
json_lexer (*C struct*), 1304
json_obj (*C struct*), 1304
json_obj_descr (*C struct*), 1305
JSON_OBJ_DESCR_ARRAY (*C macro*), 1295
JSON_OBJ_DESCR_ARRAY_ARRAY (*C macro*), 1296
JSON_OBJ_DESCR_ARRAY_ARRAY_NAMED (*C macro*), 1297
JSON_OBJ_DESCR_ARRAY_NAMED (*C macro*), 1298
JSON_OBJ_DESCR_OBJ_ARRAY (*C macro*), 1296
JSON_OBJ_DESCR_OBJ_ARRAY_NAMED (*C macro*), 1299
JSON_OBJ_DESCR_OBJECT (*C macro*), 1295
JSON_OBJ_DESCR_OBJECT_NAMED (*C macro*), 1298
JSON_OBJ_DESCR_PRIM (*C macro*), 1294
JSON_OBJ_DESCR_PRIM_NAMED (*C macro*), 1298
json_obj_encode (*C function*), 1304
json_obj_encode_buf (*C function*), 1303
json_obj_parse (*C function*), 1301
json_obj_token (*C struct*), 1305

json_token (*C struct*), 1304
json_tokens (*C enum*), 1300
json_tokens.JSON_TOK_ARRAY_END (*C enumerator*), 1300
json_tokens.JSON_TOK_ARRAY_START (*C enumerator*), 1300
json_tokens.JSON_TOK_COLON (*C enumerator*), 1300
json_tokens.JSON_TOK_COMMA (*C enumerator*), 1300
json_tokens.JSON_TOK_ENCODED_OBJ (*C enumerator*), 1300
json_tokens.JSON_TOK_EOF (*C enumerator*), 1301
json_tokens.JSON_TOK_ERROR (*C enumerator*), 1301
json_tokens.JSON_TOK_FALSE (*C enumerator*), 1301
json_tokens.JSON_TOK_FLOAT (*C enumerator*), 1300
json_tokens.JSON_TOK_NONE (*C enumerator*), 1300
json_tokens.JSON_TOK_NULL (*C enumerator*), 1301
json_tokens.JSON_TOK_NUMBER (*C enumerator*), 1300
json_tokens.JSON_TOK_OBJ_ARRAY (*C enumerator*), 1300
json_tokens.JSON_TOK_OBJECT_END (*C enumerator*), 1300
json_tokens.JSON_TOK_OBJECT_START (*C enumerator*), 1300
json_tokens.JSON_TOK_OPAQUE (*C enumerator*), 1300
json_tokens.JSON_TOK_STRING (*C enumerator*), 1300
json_tokens.JSON_TOK_TRUE (*C enumerator*), 1301
jwt_add_payload (*C function*), 1305
jwt_builder (*C struct*), 1305
jwt_builder.base (*C var*), 1306
jwt_builder.buf (*C var*), 1306
jwt_builder.len (*C var*), 1306
jwt_builder.overflowed (*C var*), 1306
jwt_init_builder (*C function*), 1305
jwt_payload_len (*C function*), 1305
jwt_sign (*C function*), 1305

K

K_AGU_IDX (*C macro*), 319
K_AGU_REGS (*C macro*), 319
k_aligned_alloc (*C function*), 565
k_busy_wait (*C function*), 326
K_CALLBACK_STATE (*C macro*), 319
k_calloc (*C function*), 566
k_can_yield (*C function*), 326
k_condvar_broadcast (*C function*), 413
K_CONDVAR_DEFINE (*C macro*), 412
k_condvar_init (*C function*), 413
k_condvar_signal (*C function*), 413
k_condvar_wait (*C function*), 413
k_cpu_atomic_idle (*C function*), 346
k_cpu_idle (*C function*), 346
k_current_get (*C function*), 327
K_CYC (*C macro*), 475
k_cycle_get_32 (*C function*), 480
k_cycle_get_64 (*C function*), 480
K_DSP_IDX (*C macro*), 319
K_DSP_REGS (*C macro*), 319
K_ESSENTIAL (*C macro*), 318
k_event (*C struct*), 419
k_event_clear (*C function*), 417
K_EVENT_DEFINE (*C macro*), 416
k_event_init (*C function*), 416
k_event_post (*C function*), 416
k_event_set (*C function*), 417

k_event_set_masked (*C function*), 417
k_event_test (*C function*), 419
k_event_wait (*C function*), 418
k_event_wait_all (*C function*), 418
k_fatal_halt (*C function*), 509
k_fifo_alloc_put (*C macro*), 435
k_fifo_cancel_wait (*C macro*), 435
K_FIFO_DEFINE (*C macro*), 437
k_fifo_get (*C macro*), 436
k_fifo_init (*C macro*), 435
k_fifo_is_empty (*C macro*), 437
k_fifo_peek_head (*C macro*), 437
k_fifo_peek_tail (*C macro*), 437
k_fifo_put (*C macro*), 435
k_fifo_put_list (*C macro*), 436
k_fifo_put_slist (*C macro*), 436
k_float_disable (*C function*), 505
k_float_enable (*C function*), 505
K_FOREVER (*C macro*), 476
K_FP_IDX (*C macro*), 318
K_FP_REGS (*C macro*), 319
k_free (*C function*), 565
k_futex_wait (*C function*), 408
k_futex_wake (*C function*), 409
k_heap (*C struct*), 566
k_heap_aligned_alloc (*C function*), 563
k_heap_alloc (*C function*), 564
K_HEAP_DEFINE (*C macro*), 562
K_HEAP_DEFINE_NOCACHE (*C macro*), 562
k_heap_free (*C function*), 565
k_heap_init (*C function*), 563
k_heap_realloc (*C function*), 564
K_HOURS (*C macro*), 475
K_INHERIT_PERMS (*C macro*), 319
k_is_in_isr (*C function*), 389
k_is_pre_kernel (*C function*), 389
k_is_preempt_thread (*C function*), 389
K_KERNEL_PINNED_STACK_ARRAY_DECLARE (*C macro*), 335
K_KERNEL_PINNED_STACK_ARRAY_DEFINE (*C macro*), 336
K_KERNEL_PINNED_STACK_DEFINE (*C macro*), 336
K_KERNEL_STACK_ARRAY_DECLARE (*C macro*), 335
K_KERNEL_STACK_ARRAY_DEFINE (*C macro*), 336
K_KERNEL_STACK_DECLARE (*C macro*), 335
K_KERNEL_STACK_DEFINE (*C macro*), 335
K_KERNEL_STACK_MEMBER (*C macro*), 336
K_KERNEL_STACK_SIZEOF (*C macro*), 336
K_KERNEL_THREAD_DEFINE (*C macro*), 320
k_lifo_alloc_put (*C macro*), 440
K_LIFO_DEFINE (*C macro*), 441
k_lifo_get (*C macro*), 441
k_lifo_init (*C macro*), 440
k_lifo_put (*C macro*), 440
k_malloc (*C function*), 565
k_mbox (*C struct*), 462
k_mbox_async_put (*C function*), 461
k_mbox_data_get (*C function*), 461
K_MBOX_DEFINE (*C macro*), 460
k_mbox_get (*C function*), 461

k_mbox_init (*C function*), 460
k_mbox_msg (*C struct*), 462
k_mbox_msg.info (*C var*), 462
k_mbox_msg.rx_source_thread (*C var*), 462
k_mbox_msg.size (*C var*), 462
k_mbox_msg.tx_data (*C var*), 462
k_mbox_msg.tx_target_thread (*C var*), 462
k_mbox_put (*C function*), 460
k_mbox.rx_msg_queue (*C var*), 462
k_mbox.tx_msg_queue (*C var*), 462
K_MEM_CACHE_MASK (*C macro*), 603
K_MEM_CACHE_NONE (*C macro*), 603
K_MEM_CACHE_WB (*C macro*), 603
K_MEM_CACHE_WT (*C macro*), 603
K_MEM_DIRECT_MAP (*C macro*), 604
k_mem_domain (*C struct*), 542
k_mem_domain_add_partition (*C function*), 540
k_mem_domain_add_thread (*C function*), 541
k_mem_domain_default (*C var*), 541
k_mem_domain_init (*C function*), 540
k_mem_domain_remove_partition (*C function*), 541
k_mem_domain.mem_domain_q (*C var*), 542
k_mem_domain.num_partitions (*C var*), 542
k_mem_domain.partitions (*C var*), 542
k_mem_free_get (*C function*), 604
k_mem_map (*C function*), 604
K_MEM_MAP_LOCK (*C macro*), 604
K_MEM_MAP_UNINIT (*C macro*), 604
k_mem_page_in (*C function*), 595
k_mem_page_out (*C function*), 595
k_mem_paging_backing_store_init (*C function*), 600
k_mem_paging_backing_store_location_free (*C function*), 599
k_mem_paging_backing_store_location_get (*C function*), 599
k_mem_paging_backing_store_page_finalize (*C function*), 600
k_mem_paging_backing_store_page_in (*C function*), 600
k_mem_paging_backing_store_page_out (*C function*), 600
k_mem_paging_eviction_accessed (*C function*), 598
k_mem_paging_eviction_add (*C function*), 598
k_mem_paging_eviction_init (*C function*), 598
k_mem_paging_eviction_remove (*C function*), 598
k_mem_paging_eviction_select (*C function*), 598
k_mem_paging_histogram_backing_store_page_in_get (*C function*), 597
k_mem_paging_histogram_backing_store_page_out_get (*C function*), 597
k_mem_paging_histogram_eviction_get (*C function*), 596
k_mem_paging_histogram_t (*C struct*), 597
k_mem_paging_stats_get (*C function*), 596
k_mem_paging_stats_t (*C struct*), 597
k_mem_paging_stats_t.clean (*C var*), 597
k_mem_paging_stats_t.cnt (*C var*), 597
k_mem_paging_stats_t.dirty (*C var*), 597
k_mem_paging_stats_t.in_isr (*C var*), 597
k_mem_paging_stats_t.irq_locked (*C var*), 597
k_mem_paging_stats_t.irq_unlocked (*C var*), 597
k_mem_paging_thread_stats_get (*C function*), 596
k_mem_partition (*C struct*), 541
K_MEM_PARTITION_DEFINE (*C macro*), 540
k_mem_partition.attr (*C var*), 542
k_mem_partition.size (*C var*), 542

k_mem_partition.start (*C var*), 542
K_MEM_PERM_EXEC (*C macro*), 604
K_MEM_PERM_RW (*C macro*), 604
K_MEM_PERM_USER (*C macro*), 604
k_mem_pin (*C function*), 596
k_mem_region_align (*C function*), 605
k_mem_slab_alloc (*C function*), 583
K_MEM_SLAB_DEFINE (*C macro*), 582
K_MEM_SLAB_DEFINE_STATIC (*C macro*), 583
k_mem_slab_free (*C function*), 584
k_mem_slab_init (*C function*), 583
k_mem_slab_max_used_get (*C function*), 584
k_mem_slab_num_free_get (*C function*), 584
k_mem_slab_num_used_get (*C function*), 584
k_mem_slab_runtime_stats_get (*C function*), 585
k_mem_slab_runtime_stats_reset_max (*C function*), 585
k_mem_unmap (*C function*), 605
k_mem_unpin (*C function*), 596
K_MINUTES (*C macro*), 475
K_MSEC (*C macro*), 475
k_msgq (*C struct*), 452
k_msgq_alloc_init (*C function*), 449
k_msgq_attrs (*C struct*), 453
k_msgq_attrs.max_msgs (*C var*), 453
k_msgq_attrs.msg_size (*C var*), 453
k_msgq_attrs.used_msgs (*C var*), 453
k_msgq_cleanup (*C function*), 449
K_MSGQ_DEFINE (*C macro*), 448
K_MSGQ_FLAG_ALLOC (*C macro*), 448
k_msgq_get (*C function*), 450
k_msgq_get_attrs (*C function*), 452
k_msgq_init (*C function*), 449
k_msgq_num_free_get (*C function*), 451
k_msgq_num_used_get (*C function*), 452
k_msgq_peek (*C function*), 450
k_msgq_peek_at (*C function*), 451
k_msgq_purge (*C function*), 451
k_msgq_put (*C function*), 449
k_msgq.buffer_end (*C var*), 452
k_msgq.buffer_start (*C var*), 452
k_msgq.flags (*C var*), 453
k_msgq.lock (*C var*), 452
k_msgq.max_msgs (*C var*), 452
k_msgq.msg_size (*C var*), 452
k_msgq.read_ptr (*C var*), 452
k_msgq.used_msgs (*C var*), 452
k_msgq.wait_q (*C var*), 452
k_msgq.write_ptr (*C var*), 452
k_msleep (*C function*), 325
k_mutex (*C struct*), 408
K_MUTEX_DEFINE (*C macro*), 406
k_mutex_init (*C function*), 407
k_mutex_lock (*C function*), 407
k_mutex_unlock (*C function*), 407
k_mutex.lock_count (*C var*), 408
k_mutex.owner (*C var*), 408
k_mutex.owner_orig_prio (*C var*), 408
k_mutex.wait_q (*C var*), 408

K_NO_WAIT (*C macro*), 474
K_NSEC (*C macro*), 474
K_OBJ_CORE (*C macro*), 666
k_obj_core (*C struct*), 669
k_obj_core_init (*C function*), 668
k_obj_core_init_and_link (*C function*), 668
k_obj_core_link (*C function*), 668
k_obj_core_stats_deregister (*C function*), 670
k_obj_core_stats_desc (*C struct*), 668
k_obj_core_stats_desc.disable (*C var*), 669
k_obj_core_stats_desc.enable (*C var*), 669
k_obj_core_stats_desc.query (*C var*), 668
k_obj_core_stats_desc.query_size (*C var*), 668
k_obj_core_stats_desc.raw (*C var*), 668
k_obj_core_stats_desc.raw_size (*C var*), 668
k_obj_core_stats_desc.reset (*C var*), 669
k_obj_core_stats_disable (*C function*), 671
k_obj_core_stats_enable (*C function*), 671
k_obj_core_stats_query (*C function*), 670
k_obj_core_stats_raw (*C function*), 670
k_obj_core_stats_register (*C function*), 669
k_obj_core_stats_reset (*C function*), 671
k_obj_core_unlink (*C function*), 668
k_obj_core.node (*C var*), 669
k_obj_core.type (*C var*), 669
K_OBJ_FLAG_ALLOC (*C macro*), 547
K_OBJ_FLAG_DRIVER (*C macro*), 547
K_OBJ_FLAG_INITIALIZED (*C macro*), 546
K_OBJ_FLAG_PUBLIC (*C macro*), 547
k_obj_type (*C struct*), 669
K_OBJ_TYPE_CONDVAR_ID (*C macro*), 666
K_OBJ_TYPE_CPU_ID (*C macro*), 666
K_OBJ_TYPE_EVENT_ID (*C macro*), 666
K_OBJ_TYPE_FIFO_ID (*C macro*), 666
k_obj_type_find (*C function*), 667
K_OBJ_TYPE_ID_GEN (*C macro*), 666
K_OBJ_TYPE_KERNEL_ID (*C macro*), 666
K_OBJ_TYPE_LIFO_ID (*C macro*), 666
K_OBJ_TYPE_MBOX_ID (*C macro*), 666
K_OBJ_TYPE_MEM_BLOCK_ID (*C macro*), 666
K_OBJ_TYPE_MEM_SLAB_ID (*C macro*), 666
K_OBJ_TYPE_MSGQ_ID (*C macro*), 666
K_OBJ_TYPE_MUTEX_ID (*C macro*), 666
K_OBJ_TYPE_PIPE_ID (*C macro*), 666
K_OBJ_TYPE_SEM_ID (*C macro*), 666
K_OBJ_TYPE_STACK_ID (*C macro*), 666
K_OBJ_TYPE_THREAD_ID (*C macro*), 666
K_OBJ_TYPE_TIMER_ID (*C macro*), 667
k_obj_type_walk_locked (*C function*), 667
k_obj_type_walk_unlocked (*C function*), 667
k_obj_type.id (*C var*), 669
k_obj_type.list (*C var*), 669
k_obj_type.node (*C var*), 669
k_obj_type.obj_core_offset (*C var*), 669
k_object_access_all_grant (*C function*), 547
k_object_access_grant (*C function*), 547
k_object_access_revoke (*C function*), 547
k_object_alloc (*C function*), 548

k_object_alloc_size (*C function*), 548
k_object_free (*C function*), 549
k_object_is_valid (*C function*), 548
k_object_release (*C function*), 547
k_pipe (*C struct*), 468
k_pipe_alloc_init (*C function*), 467
k_pipe_buffer_flush (*C function*), 468
k_pipe_cleanup (*C function*), 466
K_PIPE_DEFINE (*C macro*), 466
k_pipe_flush (*C function*), 468
k_pipe_get (*C function*), 467
k_pipe_init (*C function*), 466
k_pipe_put (*C function*), 467
k_pipe_read_avail (*C function*), 468
k_pipe_write_avail (*C function*), 468
k_pipe.buffer (*C var*), 469
k_pipe.bytes_used (*C var*), 469
k_pipe.flags (*C var*), 469
k_pipe.lock (*C var*), 469
k_pipe.read_index (*C var*), 469
k_pipe.readers (*C var*), 469
k_pipe.size (*C var*), 469
k_pipe.write_index (*C var*), 469
k_pipe.writers (*C var*), 469
k_poll (*C function*), 396
k_poll_event (*C struct*), 398
k_poll_event_init (*C function*), 396
K_POLL_EVENT_INITIALIZER (*C macro*), 395
K_POLL_EVENT_STATIC_INITIALIZER (*C macro*), 395
k_poll_event.mode (*C var*), 398
k_poll_event.poller (*C var*), 398
k_poll_event.state (*C var*), 398
k_poll_event.tag (*C var*), 398
k_poll_event.type (*C var*), 398
k_poll_event.unused (*C var*), 398
k_poll_modes (*C enum*), 395
k_poll_modes.K_POLL_MODE_NOTIFY_ONLY (*C enumerator*), 395
k_poll_modes.K_POLL_NUM_MODES (*C enumerator*), 396
k_poll_signal (*C struct*), 398
k_poll_signal_check (*C function*), 397
k_poll_signal_init (*C function*), 397
K_POLL_SIGNAL_INITIALIZER (*C macro*), 395
k_poll_signal_raise (*C function*), 397
k_poll_signal_reset (*C function*), 397
k_poll_signal.poll_events (*C var*), 398
k_poll_signal.result (*C var*), 398
k_poll_signal.signaled (*C var*), 398
K_POLL_STATE_CANCELLED (*C macro*), 395
K_POLL_STATE_DATA_AVAILABLE (*C macro*), 395
K_POLL_STATE_FIFO_DATA_AVAILABLE (*C macro*), 395
K_POLL_STATE_MSGQ_DATA_AVAILABLE (*C macro*), 395
K_POLL_STATE_NOT_READY (*C macro*), 395
K_POLL_STATE_PIPE_DATA_AVAILABLE (*C macro*), 395
K_POLL_STATE_SEM_AVAILABLE (*C macro*), 395
K_POLL_STATE_SIGNALED (*C macro*), 395
K_POLL_TYPE_DATA_AVAILABLE (*C macro*), 395
K_POLL_TYPE_FIFO_DATA_AVAILABLE (*C macro*), 395
K_POLL_TYPE_IGNORE (*C macro*), 395

K_POLL_TYPE_MSGQ_DATA_AVAILABLE (*C macro*), 395
K_POLL_TYPE_PIPE_DATA_AVAILABLE (*C macro*), 395
K_POLL_TYPE_SEM_AVAILABLE (*C macro*), 395
K_POLL_TYPE_SIGNAL (*C macro*), 395
k_queue_alloc_append (*C function*), 428
k_queue_alloc_prepend (*C function*), 429
k_queue_append (*C function*), 428
k_queue_append_list (*C function*), 430
k_queue_cancel_wait (*C function*), 428
K_QUEUE_DEFINE (*C macro*), 427
k_queue_get (*C function*), 430
k_queue_init (*C function*), 428
k_queue_insert (*C function*), 429
k_queue_is_empty (*C function*), 431
k_queue_merge_slist (*C function*), 430
k_queue_peek_head (*C function*), 432
k_queue_peek_tail (*C function*), 432
k_queue_prepend (*C function*), 429
k_queue_remove (*C function*), 431
k_queue_unique_append (*C function*), 431
k_realloc (*C function*), 566
k_sched_current_thread_query (*C function*), 326
k_sched_lock (*C function*), 332
k_sched_time_slice_set (*C function*), 330
k_sched_unlock (*C function*), 332
K_SECONDS (*C macro*), 475
k_sem_count_get (*C function*), 402
K_SEM_DEFINE (*C macro*), 401
k_sem_give (*C function*), 402
k_sem_init (*C function*), 401
K_SEM_MAX_LIMIT (*C macro*), 401
k_sem_reset (*C function*), 402
k_sem_take (*C function*), 402
k_sleep (*C function*), 325
k_spin_lock (*C function*), 425
k_spin_trylock (*C function*), 425
k_spin_unlock (*C function*), 426
K_SPINLOCK (*C macro*), 424
k_spinlock (*C struct*), 426
K_SPINLOCK_BREAK (*C macro*), 424
k_spinlock_key_t (*C type*), 425
K_SSE_REGS (*C macro*), 319
k_stack_alloc_init (*C function*), 444
k_stack_cleanup (*C function*), 444
K_STACK_DEFINE (*C macro*), 443
k_stack_init (*C function*), 444
k_stack_pop (*C function*), 445
k_stack_push (*C function*), 444
k_sys_fatal_error_handler (*C function*), 510
k_thread (*C struct*), 333
k_thread_abort (*C function*), 327
K_THREAD_ACCESS_GRANT (*C macro*), 546
k_thread_access_grant (*C macro*), 319
k_thread_cpu_mask_clear (*C function*), 329
k_thread_cpu_mask_disable (*C function*), 330
k_thread_cpu_mask_enable (*C function*), 329
k_thread_cpu_mask_enable_all (*C function*), 329
k_thread_cpu_pin (*C function*), 330

k_thread_create (*C function*), 323
k_thread_custom_data_get (*C function*), 332
k_thread_custom_data_set (*C function*), 332
k_thread_deadline_set (*C function*), 328
K_THREAD_DEFINE (*C macro*), 320
k_thread_foreach (*C function*), 321
k_thread_foreach_unlocked (*C function*), 322
k_thread_heap_assign (*C function*), 324
k_thread_join (*C function*), 324
k_thread_name_copy (*C function*), 333
k_thread_name_get (*C function*), 333
k_thread_name_set (*C function*), 332
K_THREAD_PINNED_STACK_ARRAY_DEFINE (*C macro*), 338
K_THREAD_PINNED_STACK_DEFINE (*C macro*), 337
k_thread_priority_get (*C function*), 327
k_thread_priority_set (*C function*), 328
k_thread_resume (*C function*), 330
k_thread_stack_alloc (*C function*), 322
K_THREAD_STACK_ARRAY_DECLARE (*C macro*), 337
K_THREAD_STACK_ARRAY_DEFINE (*C macro*), 338
K_THREAD_STACK_DECLARE (*C macro*), 336
K_THREAD_STACK_DEFINE (*C macro*), 337
k_thread_stack_free (*C function*), 323
K_THREAD_STACK_LEN (*C macro*), 338
K_THREAD_STACK_MEMBER (*C macro*), 339
K_THREAD_STACK_SIZEOF (*C macro*), 337
k_thread_start (*C function*), 327
k_thread_state_str (*C function*), 333
k_thread_suspend (*C function*), 330
k_thread_time_slice_set (*C function*), 331
k_thread_timeout_expires_ticks (*C function*), 327
k_thread_timeout_remaining_ticks (*C function*), 327
k_thread_user_cb_t (*C type*), 321
k_thread_user_mode_enter (*C function*), 324
k_thread.arch (*C var*), 334
k_thread.callee_saved (*C var*), 334
k_thread.custom_data (*C var*), 334
k_thread.entry (*C var*), 334
k_thread.halt_queue (*C var*), 334
k_thread.init_data (*C var*), 334
k_thread.join_queue (*C var*), 334
k_thread.mem_domain_info (*C var*), 334
k_thread.next_thread (*C var*), 334
k_thread.resource_pool (*C var*), 334
k_thread.stack_info (*C var*), 334
k_thread.stack_obj (*C var*), 334
k_thread.swap_retval (*C var*), 334
k_thread.switch_handle (*C var*), 334
k_thread.syscall_frame (*C var*), 334
K_TICKS (*C macro*), 475
K_TICKS_FOREVER (*C macro*), 476
k_ticks_t (*C type*), 477
K_TIMEOUT_EQ (*C macro*), 476
k_timeout_t (*C struct*), 482
k_timepoint_t (*C struct*), 482
K_TIMER_DEFINE (*C macro*), 487
k_timer_expires_ticks (*C function*), 489
k_timer_expiry_t (*C type*), 487

k_timer_init (*C function*), 487
k_timer_remaining_get (*C function*), 489
k_timer_remaining_ticks (*C function*), 489
k_timer_start (*C function*), 487
k_timer_status_get (*C function*), 488
k_timer_status_sync (*C function*), 488
k_timer_stop (*C function*), 488
k_timer_stop_t (*C type*), 487
k_timer_user_data_get (*C function*), 489
k_timer_user_data_set (*C function*), 489
k_uptime_delta (*C function*), 480
k_uptime_get (*C function*), 479
k_uptime_get_32 (*C function*), 479
k_uptime_seconds (*C function*), 480
k_uptime_ticks (*C function*), 479
K_USEC (*C macro*), 474
K_USER (*C macro*), 319
k_usleep (*C function*), 325
k_wakeup (*C function*), 326
k_work (*C struct*), 371
k_work_busy_get (*C function*), 357
k_work_cancel (*C function*), 359
k_work_cancel_delayable (*C function*), 366
k_work_cancel_delayable_sync (*C function*), 367
k_work_cancel_sync (*C function*), 360
K_WORK_DEFINE (*C macro*), 356
k_work_delayable (*C struct*), 371
k_work_delayable_busy_get (*C function*), 362
K_WORK_DELAYABLE_DEFINE (*C macro*), 355
k_work_delayable_expires_get (*C function*), 363
k_work_delayable_from_work (*C function*), 362
k_work_delayable_is_pending (*C function*), 363
k_work_delayable_remaining_get (*C function*), 363
k_work_flush (*C function*), 359
k_work_flush_delayable (*C function*), 366
k_work_handler_t (*C type*), 356
k_work_init (*C function*), 357
k_work_init_delayable (*C function*), 362
k_work_is_pending (*C function*), 357
k_work_poll_cancel (*C function*), 371
k_work_poll_init (*C function*), 369
k_work_poll_submit (*C function*), 370
k_work_poll_submit_to_queue (*C function*), 369
k_work_q (*C struct*), 372
k_work_queue_config (*C struct*), 371
k_work_queue_config.essential (*C var*), 372
k_work_queue_config.name (*C var*), 372
k_work_queue_config.no_yield (*C var*), 372
k_work_queue_drain (*C function*), 361
k_work_queue_init (*C function*), 360
k_work_queue_start (*C function*), 360
k_work_queue_thread_get (*C function*), 361
k_work_queue_unplug (*C function*), 361
k_work_reschedule (*C function*), 365
k_work_reschedule_for_queue (*C function*), 365
k_work_schedule (*C function*), 364
k_work_schedule_for_queue (*C function*), 364
k_work_submit (*C function*), 358

k_work_submit_to_queue (*C function*), 358
k_work_sync (*C struct*), 371
K_WORK_USER_DEFINE (*C macro*), 355
k_work_user_handler_t (*C type*), 356
k_work_user_init (*C function*), 367
k_work_user_is_pending (*C function*), 368
k_work_user_queue_start (*C function*), 368
k_work_user_queue_thread_get (*C function*), 369
k_work_user_submit_to_queue (*C function*), 368
k_yield (*C function*), 326
KB (*C macro*), 685
kbd_row_t (*C type*), 880
kernel, **3936**
KHZ (*C macro*), 685
kscan_callback_t (*C type*), 3472
kscan_config (*C function*), 3473
kscan_disable_callback (*C function*), 3473
kscan_enable_callback (*C function*), 3473

L

launch() (*twister_harness.DeviceAdapter method*), 268
led_api_blink (*C type*), 3474
led_api_get_info (*C type*), 3474
led_api_length (*C type*), 3479
led_api_off (*C type*), 3475
led_api_on (*C type*), 3475
led_api_set_brightness (*C type*), 3475
led_api_set_color (*C type*), 3475
led_api_update_channels (*C type*), 3479
led_api_update_rgb (*C type*), 3479
led_api_write_channels (*C type*), 3475
led_blink (*C function*), 3476
led_driver_api (*C struct*), 3478
led_get_info (*C function*), 3476
led_info (*C struct*), 3478
led_info.color_mapping (*C var*), 3478
led_info.index (*C var*), 3478
led_info.label (*C var*), 3478
led_info.num_colors (*C var*), 3478
led_off (*C function*), 3478
led_on (*C function*), 3477
led_rgb (*C struct*), 3480
led_rgb.b (*C var*), 3480
led_rgb.g (*C var*), 3480
led_rgb.r (*C var*), 3480
led_set_brightness (*C function*), 3476
led_set_channel (*C function*), 3477
led_set_color (*C function*), 3477
led_strip_driver_api (*C struct*), 3480
led_strip_length (*C function*), 3480
led_strip_update_channels (*C function*), 3480
led_strip_update_rgb (*C function*), 3479
led_write_channels (*C function*), 3476
LIST_DROP_EMPTY (*C macro*), 689
listen (*C function*), 2488
LISTIFY (*C macro*), 691
LL_EXTENSION_SYMBOL (*C macro*), 928
llext (*C struct*), 927

llext_add_domain (*C function*), 926
LLEXT_BUF_LOADER (*C macro*), 930
llext_buf_loader (*C struct*), 930
llext_buf_loader.loader (*C var*), 930
llext_by_name (*C function*), 924
llext_call_fn (*C function*), 925
llext_const_symbol (*C struct*), 928
llext_const_symbol.addr (*C var*), 929
llext_const_symbol.name (*C var*), 929
llext_const_symbol.slid (*C var*), 929
llext_find_section (*C function*), 926
llext_find_sym (*C function*), 925
llext_iterate (*C function*), 924
llext_load (*C function*), 925
llext_load_param (*C struct*), 928
LLEXT_LOAD_PARAM_DEFAULT (*C macro*), 923
llext_load_param.pre_located (*C var*), 928
llext_load_param.relocate_local (*C var*), 928
llext_loader (*C struct*), 930
llext_loader.peek (*C var*), 930
llext_loader.read (*C var*), 930
llext_loader.seek (*C var*), 930
llext_mem (*C enum*), 924
llext_mem.LLEXT_MEM_BSS (*C enumerator*), 924
llext_mem.LLEXT_MEM_COUNT (*C enumerator*), 924
llext_mem.LLEXT_MEM_DATA (*C enumerator*), 924
llext_mem.LLEXT_MEM_EXPORT (*C enumerator*), 924
llext_mem.LLEXT_MEM_RODATA (*C enumerator*), 924
llext_mem.LLEXT_MEM_SHSTRTAB (*C enumerator*), 924
llext_mem.LLEXT_MEM_STRTAB (*C enumerator*), 924
llext_mem.LLEXT_MEM_SYMTAB (*C enumerator*), 924
llext_mem.LLEXT_MEM_TEXT (*C enumerator*), 924
llext_symbol (*C struct*), 929
llext_symbol.addr (*C var*), 929
llext_symbol.name (*C var*), 929
llext_symtable (*C struct*), 929
llext_symtable.sym_cnt (*C var*), 929
llext_symtable.syms (*C var*), 929
llext_unload (*C function*), 925
llext.alloc_size (*C var*), 927
llext.exp_tab (*C var*), 927
llext.mem (*C var*), 927
llext.mem_on_heap (*C var*), 927
llext.mem_size (*C var*), 927
llext.name (*C var*), 927
llext.sym_tab (*C var*), 927
llext.use_count (*C var*), 927
LOG2 (*C macro*), 684
LOG2CEIL (*C macro*), 685
log_backend (*C struct*), 961
log_backend_activate (*C function*), 960
log_backend_api (*C struct*), 961
log_backend_control_block (*C struct*), 961
log_backend_count_get (*C function*), 960
log_backend_deactivate (*C function*), 960
LOG_BACKEND_DEFINE (*C macro*), 958
log_backend_disable (*C function*), 953
log_backend_dropped (*C function*), 959

log_backend_enable (*C function*), 953
log_backend_evt (*C enum*), 958
log_backend_evt_arg (*C union*), 961
log_backend_evt_arg.raw (*C var*), 961
log_backend_evt.LOG_BACKEND_EVT_MAX (*C enumerator*), 959
log_backend_evt.LOG_BACKEND_EVT_PROCESS_THREAD_DONE (*C enumerator*), 958
log_backend_format_set (*C function*), 961
log_backend_get (*C function*), 960
log_backend_get_by_name (*C function*), 953
log_backend_id_get (*C function*), 960
log_backend_id_set (*C function*), 960
log_backend_init (*C function*), 959
log_backend_is_active (*C function*), 961
log_backend_is_ready (*C function*), 959
log_backend_msg_process (*C function*), 959
log_backend_notify (*C function*), 961
log_backend_panic (*C function*), 959
log_backend_shell_api (*C var*), 1141
log_buffered_cnt (*C function*), 951
log_core_init (*C function*), 950
LOG_CORE_INIT (*C macro*), 950
log_custom_output_msg_process (*C function*), 962
log_data_pending (*C function*), 954
LOG_DBG (*C macro*), 945
log_domain_name_get (*C function*), 952
log_domains_count (*C function*), 952
LOG_ERR (*C macro*), 945
log_filter_get (*C function*), 952
log_filter_set (*C function*), 952
log_format_func_t (*C type*), 963
log_format_func_t_get (*C function*), 963
log_format_set_all_active_backends (*C function*), 953
log_frontend_filter_get (*C function*), 953
log_frontend_filter_set (*C function*), 953
LOG_HEXDUMP_DBG (*C macro*), 947
LOG_HEXDUMP_ERR (*C macro*), 947
LOG_HEXDUMP_INF (*C macro*), 947
LOG_HEXDUMP_WRN (*C macro*), 947
LOG_INF (*C macro*), 945
log_init (*C function*), 950
LOG_INIT (*C macro*), 950
LOG_INST_DBG (*C macro*), 947
LOG_INST_ERR (*C macro*), 946
LOG_INST_HEXDUMP_DBG (*C macro*), 948
LOG_INST_HEXDUMP_ERR (*C macro*), 948
LOG_INST_HEXDUMP_INF (*C macro*), 948
LOG_INST_HEXDUMP_WRN (*C macro*), 948
LOG_INST_INF (*C macro*), 946
LOG_INST_WRN (*C macro*), 946
LOG_LEVEL_SET (*C macro*), 950
log_mem_get_max_usage (*C function*), 954
log_mem_get_usage (*C function*), 954
LOG_MODULE_DECLARE (*C macro*), 949
LOG_MODULE_REGISTER (*C macro*), 948
log_msg (*C struct*), 957
log_msg_desc (*C struct*), 957
log_msg_generic (*C union*), 957
log_msg_generic_get_wlen (*C function*), 955

LOG_MSG_GENERIC_HDR (*C macro*), 955
log_msg_generic_hdr (*C struct*), 957
log_msg_generic.buf (*C var*), 957
log_msg_generic.generic (*C var*), 957
log_msg_generic.log (*C var*), 958
log_msg_get_data (*C function*), 956
log_msg_get_domain (*C function*), 956
log_msg_get_level (*C function*), 956
log_msg_get_package (*C function*), 957
log_msg_get_source (*C function*), 956
log_msg_get_source_id (*C function*), 956
log_msg_get_tid (*C function*), 956
log_msg_get_timestamp (*C function*), 956
log_msg_get_total_wlen (*C function*), 955
log_msg_hdr (*C struct*), 957
LOG_MSG_SIMPLE_ARG_CNT_CHECK (*C macro*), 955
LOG_MSG_SIMPLE_ARG_TYPE_CHECK (*C macro*), 955
LOG_MSG_SIMPLE_ARG_TYPE_CHECK_0 (*C macro*), 955
LOG_MSG_SIMPLE_ARG_TYPE_CHECK_1 (*C macro*), 955
LOG_MSG_SIMPLE_ARG_TYPE_CHECK_2 (*C macro*), 955
LOG_MSG_SIMPLE_CHECK (*C macro*), 955
LOG_MSG_SIMPLE_FUNC (*C macro*), 955
log_msg_source (*C union*), 957
log_msg_source.dynamic (*C var*), 957
log_msg_source.fixed (*C var*), 957
log_msg_source.raw (*C var*), 957
log_output (*C struct*), 965
log_output_control_block (*C struct*), 965
log_output_ctx_set (*C function*), 964
LOG_OUTPUT_CUSTOM (*C macro*), 962
LOG_OUTPUT_DEFINE (*C macro*), 962
LOG_OUTPUT_DICT (*C macro*), 962
log_output_dropped_process (*C function*), 964
log_output_flush (*C function*), 964
log_output_func_t (*C type*), 963
log_output_hostname_set (*C function*), 964
log_output_msg_process (*C function*), 963
log_output_msg_syst_process (*C function*), 964
log_output_process (*C function*), 963
LOG_OUTPUT_SYST (*C macro*), 962
LOG_OUTPUT_TEXT (*C macro*), 962
log_output_timestamp_freq_set (*C function*), 965
log_output_timestamp_to_us (*C function*), 965
log_panic (*C function*), 951
LOG_PANIC (*C macro*), 950
LOG_PRINTK (*C macro*), 946
log_process (*C function*), 951
LOG_PROCESS (*C macro*), 950
LOG_RAW (*C macro*), 946
log_set_tag (*C function*), 954
log_set_timestamp_func (*C function*), 951
log_source_id_get (*C function*), 952
log_source_name_get (*C function*), 952
log_src_cnt_get (*C function*), 951
log_thread_set (*C function*), 951
log_thread_trigger (*C function*), 951
log_timestamp_get_t (*C type*), 950
LOG_WRN (*C macro*), 945

LOG_WRN_ONCE (*C macro*), 945
logger (*runners.core.ZephyrBinaryRunner attribute*), 200
lora_coding_rate (*C enum*), 2997
lora_coding_rate.CR_4_5 (*C enumerator*), 2997
lora_coding_rate.CR_4_6 (*C enumerator*), 2997
lora_coding_rate.CR_4_7 (*C enumerator*), 2997
lora_coding_rate.CR_4_8 (*C enumerator*), 2997
lora_config (*C function*), 2997
lora_datarate (*C enum*), 2996
lora_datarate.SF_6 (*C enumerator*), 2996
lora_datarate.SF_7 (*C enumerator*), 2996
lora_datarate.SF_8 (*C enumerator*), 2996
lora_datarate.SF_9 (*C enumerator*), 2996
lora_datarate.SF_10 (*C enumerator*), 2996
lora_datarate.SF_11 (*C enumerator*), 2996
lora_datarate.SF_12 (*C enumerator*), 2996
lora_modem_config (*C struct*), 2999
lora_modem_config.bandwidth (*C var*), 2999
lora_modem_config.coding_rate (*C var*), 2999
lora_modem_config.datarate (*C var*), 2999
lora_modem_config.frequency (*C var*), 2999
lora_modem_config.iq_inverted (*C var*), 2999
lora_modem_config.preamble_len (*C var*), 2999
lora_modem_config.public_network (*C var*), 2999
lora_modem_config.tx (*C var*), 2999
lora_modem_config.tx_power (*C var*), 2999
lora_recv (*C function*), 2998
lora_recv_async (*C function*), 2998
lora_send (*C function*), 2997
lora_send_async (*C function*), 2997
lora_signal_bandwidth (*C enum*), 2996
lora_signal_bandwidth.BW_125_KHZ (*C enumerator*), 2996
lora_signal_bandwidth.BW_250_KHZ (*C enumerator*), 2996
lora_signal_bandwidth.BW_500_KHZ (*C enumerator*), 2996
lora_test_cw (*C function*), 2998
lorawan_act_type (*C enum*), 3001
lorawan_act_type.LORAWAN_ACT_ABP (*C enumerator*), 3001
lorawan_act_type.LORAWAN_ACT_OTAA (*C enumerator*), 3001
lorawan_battery_level_cb_t (*C type*), 3000
lorawan_channels_mask_size (*C enum*), 3001
lorawan_channels_mask_size.LORAWAN_CHANNELS_MASK_SIZE_AS923 (*C enumerator*), 3001
lorawan_channels_mask_size.LORAWAN_CHANNELS_MASK_SIZE_AU915 (*C enumerator*), 3001
lorawan_channels_mask_size.LORAWAN_CHANNELS_MASK_SIZE_CN470 (*C enumerator*), 3001
lorawan_channels_mask_size.LORAWAN_CHANNELS_MASK_SIZE_CN779 (*C enumerator*), 3001
lorawan_channels_mask_size.LORAWAN_CHANNELS_MASK_SIZE_EU433 (*C enumerator*), 3001
lorawan_channels_mask_size.LORAWAN_CHANNELS_MASK_SIZE_EU868 (*C enumerator*), 3001
lorawan_channels_mask_size.LORAWAN_CHANNELS_MASK_SIZE_IN865 (*C enumerator*), 3002
lorawan_channels_mask_size.LORAWAN_CHANNELS_MASK_SIZE_KR920 (*C enumerator*), 3001
lorawan_channels_mask_size.LORAWAN_CHANNELS_MASK_SIZE_RU864 (*C enumerator*), 3002
lorawan_channels_mask_size.LORAWAN_CHANNELS_MASK_SIZE_US915 (*C enumerator*), 3002
lorawan_class (*C enum*), 3001
lorawan_class.LORAWAN_CLASS_A (*C enumerator*), 3001
lorawan_class.LORAWAN_CLASS_B (*C enumerator*), 3001
lorawan_class.LORAWAN_CLASS_C (*C enumerator*), 3001
lorawan_datarate (*C enum*), 3002
lorawan_datarate.LORAWAN_DR_0 (*C enumerator*), 3002
lorawan_datarate.LORAWAN_DR_1 (*C enumerator*), 3002
lorawan_datarate.LORAWAN_DR_2 (*C enumerator*), 3002

lorawan_datarate.LORAWAN_DR_3 (*C enumerator*), 3002
lorawan_datarate.LORAWAN_DR_4 (*C enumerator*), 3002
lorawan_datarate.LORAWAN_DR_5 (*C enumerator*), 3002
lorawan_datarate.LORAWAN_DR_6 (*C enumerator*), 3002
lorawan_datarate.LORAWAN_DR_7 (*C enumerator*), 3002
lorawan_datarate.LORAWAN_DR_8 (*C enumerator*), 3002
lorawan_datarate.LORAWAN_DR_9 (*C enumerator*), 3002
lorawan_datarate.LORAWAN_DR_10 (*C enumerator*), 3002
lorawan_datarate.LORAWAN_DR_11 (*C enumerator*), 3002
lorawan_datarate.LORAWAN_DR_12 (*C enumerator*), 3003
lorawan_datarate.LORAWAN_DR_13 (*C enumerator*), 3003
lorawan_datarate.LORAWAN_DR_14 (*C enumerator*), 3003
lorawan_datarate.LORAWAN_DR_15 (*C enumerator*), 3003
lorawan_downlink_cb (*C struct*), 3008
lorawan_downlink_cb.cb (*C var*), 3008
lorawan_downlink_cb.node (*C var*), 3008
lorawan_downlink_cb.port (*C var*), 3008
lorawan_dr_changed_cb_t (*C type*), 3000
lorawan_enable_adr (*C function*), 3005
lorawan_get_min_datarate (*C function*), 3006
lorawan_get_payload_sizes (*C function*), 3006
lorawan_join (*C function*), 3004
lorawan_join_abp (*C struct*), 3007
lorawan_join_abp.app_eui (*C var*), 3007
lorawan_join_abp.app_skey (*C var*), 3007
lorawan_join_abp.dev_addr (*C var*), 3007
lorawan_join_abp.nwk_skey (*C var*), 3007
lorawan_join_config (*C struct*), 3007
lorawan_join_config.abp (*C var*), 3007
lorawan_join_config.dev_eui (*C var*), 3008
lorawan_join_config.mode (*C var*), 3008
lorawan_join_config.otaa (*C var*), 3007
lorawan_join_otaa (*C struct*), 3006
lorawan_join_otaa.app_key (*C var*), 3007
lorawan_join_otaa.dev_nonce (*C var*), 3007
lorawan_join_otaa.join_eui (*C var*), 3007
lorawan_join_otaa.nwk_key (*C var*), 3007
lorawan_message_type (*C enum*), 3003
lorawan_message_type.LORAWAN_MSG_CONFIRMED (*C enumerator*), 3004
lorawan_message_type.LORAWAN_MSG_UNCONFIRMED (*C enumerator*), 3004
lorawan_region (*C enum*), 3003
lorawan_region.LORAWAN_REGION_AS923 (*C enumerator*), 3003
lorawan_region.LORAWAN_REGION_AU915 (*C enumerator*), 3003
lorawan_region.LORAWAN_REGION_CN470 (*C enumerator*), 3003
lorawan_region.LORAWAN_REGION_CN779 (*C enumerator*), 3003
lorawan_region.LORAWAN_REGION_EU433 (*C enumerator*), 3003
lorawan_region.LORAWAN_REGION_EU868 (*C enumerator*), 3003
lorawan_region.LORAWAN_REGION_IN865 (*C enumerator*), 3003
lorawan_region.LORAWAN_REGION_KR920 (*C enumerator*), 3003
lorawan_region.LORAWAN_REGION_RU864 (*C enumerator*), 3003
lorawan_region.LORAWAN_REGION_US915 (*C enumerator*), 3003
lorawan_register_battery_level_callback (*C function*), 3004
lorawan_register_downlink_callback (*C function*), 3004
lorawan_register_dr_changed_callback (*C function*), 3004
lorawan_send (*C function*), 3004
lorawan_set_channels_mask (*C function*), 3005
lorawan_set_class (*C function*), 3005
lorawan_set_conf_msg_tries (*C function*), 3005

lorawan_set_datarate (*C function*), 3006
lorawan_set_region (*C function*), 3006
lorawan_start (*C function*), 3004
lpc_peripheral_opcode (*C enum*), 3314
lpc_peripheral_opcode.E8042_CLEAR_FLAG (*C enumerator*), 3315
lpc_peripheral_opcode.E8042_CLEAR_OBF (*C enumerator*), 3315
lpc_peripheral_opcode.E8042_IBF_HAS_CHAR (*C enumerator*), 3314
lpc_peripheral_opcode.E8042_OBF_HAS_CHAR (*C enumerator*), 3314
lpc_peripheral_opcode.E8042_PAUSE_IRQ (*C enumerator*), 3315
lpc_peripheral_opcode.E8042_READ_KB_STS (*C enumerator*), 3315
lpc_peripheral_opcode.E8042_RESUME_IRQ (*C enumerator*), 3315
lpc_peripheral_opcode.E8042_SET_FLAG (*C enumerator*), 3315
lpc_peripheral_opcode.E8042_WRITE_KB_CHAR (*C enumerator*), 3314
lpc_peripheral_opcode.E8042_WRITE_MB_CHAR (*C enumerator*), 3314
lpc_peripheral_opcode.EACPI_IBF_HAS_CHAR (*C enumerator*), 3315
lpc_peripheral_opcode.EACPI_OBF_HAS_CHAR (*C enumerator*), 3315
lpc_peripheral_opcode.EACPI_READ_STS (*C enumerator*), 3315
lpc_peripheral_opcode.EACPI_WRITE_CHAR (*C enumerator*), 3315
lpc_peripheral_opcode.EACPI_WRITE_STS (*C enumerator*), 3315
LSB_GET (*C macro*), 679
LW_RECV_PORT_ANY (*C macro*), 3000
lwm2m_acknowledge (*C function*), 2845
lwm2m_create_object_inst (*C function*), 2835
lwm2m_create_res_inst (*C function*), 2844
lwm2m_ctx (*C struct*), 2848
lwm2m_ctx_event_cb_t (*C type*), 2826
lwm2m_ctx.bootstrap_mode (*C var*), 2849
lwm2m_ctx.buffer_client_messages (*C var*), 2849
lwm2m_ctx.connection_suspended (*C var*), 2849
lwm2m_ctx.desthostname (*C var*), 2848
lwm2m_ctx.desthostnamelen (*C var*), 2848
lwm2m_ctx.event_cb (*C var*), 2849
lwm2m_ctx.fault_cb (*C var*), 2849
lwm2m_ctx.hostname_verify (*C var*), 2848
lwm2m_ctx.load_credentials (*C var*), 2848
lwm2m_ctx.observe_cb (*C var*), 2849
lwm2m_ctx.processed_req (*C var*), 2849
lwm2m_ctx.remote_addr (*C var*), 2848
lwm2m_ctx.sec_obj_inst (*C var*), 2849
lwm2m_ctx.set_socket_state (*C var*), 2850
lwm2m_ctx.set_socketoptions (*C var*), 2849
lwm2m_ctx.sock_fd (*C var*), 2849
lwm2m_ctx.srv_obj_inst (*C var*), 2849
lwm2m_ctx.tls_tag (*C var*), 2848
lwm2m_ctx.use_dtls (*C var*), 2849
lwm2m_ctx.validate_buf (*C var*), 2850
lwm2m_delete_object_inst (*C function*), 2835
lwm2m_delete_res_inst (*C function*), 2844
lwm2m_device_add_err (*C function*), 2831
LWM2M_DEVICE_BATTERY_STATUS_CHARGE_COMP (*C macro*), 2824
LWM2M_DEVICE_BATTERY_STATUS_CHARGING (*C macro*), 2824
LWM2M_DEVICE_BATTERY_STATUS_DAMAGED (*C macro*), 2824
LWM2M_DEVICE_BATTERY_STATUS_LOW (*C macro*), 2824
LWM2M_DEVICE_BATTERY_STATUS_NORMAL (*C macro*), 2824
LWM2M_DEVICE_BATTERY_STATUS_NOT_INST (*C macro*), 2824
LWM2M_DEVICE_BATTERY_STATUS_UNKNOWN (*C macro*), 2824
LWM2M_DEVICE_ERROR_EXT_POWER_SUPPLY_OFF (*C macro*), 2823
LWM2M_DEVICE_ERROR_GPS_FAILURE (*C macro*), 2823

LWM2M_DEVICE_ERROR_LOW_POWER (*C macro*), 2823
LWM2M_DEVICE_ERROR_LOW_SIGNAL_STRENGTH (*C macro*), 2823
LWM2M_DEVICE_ERROR_NETWORK_FAILURE (*C macro*), 2824
LWM2M_DEVICE_ERROR_NONE (*C macro*), 2823
LWM2M_DEVICE_ERROR_OUT_OF_MEMORY (*C macro*), 2823
LWM2M_DEVICE_ERROR_PERIPHERAL_FAILURE (*C macro*), 2824
LWM2M_DEVICE_ERROR_SMS_FAILURE (*C macro*), 2824
LWM2M_DEVICE_PWR_SRC_TYPE_AC_POWER (*C macro*), 2823
LWM2M_DEVICE_PWR_SRC_TYPE_BAT_EXT (*C macro*), 2823
LWM2M_DEVICE_PWR_SRC_TYPE_BAT_INT (*C macro*), 2823
LWM2M_DEVICE_PWR_SRC_TYPE_DC_POWER (*C macro*), 2822
LWM2M_DEVICE_PWR_SRC_TYPE_FUEL_CELL (*C macro*), 2823
LWM2M_DEVICE_PWR_SRC_TYPE_MAX (*C macro*), 2823
LWM2M_DEVICE_PWR_SRC_TYPE_PWR_OVER_ETH (*C macro*), 2823
LWM2M_DEVICE_PWR_SRC_TYPE_SOLAR (*C macro*), 2823
LWM2M_DEVICE_PWR_SRC_TYPE_USB (*C macro*), 2823
lwm2m_enable_cache (*C function*), 2847
lwm2m_engine_execute_cb_t (*C type*), 2828
lwm2m_engine_get_data_cb_t (*C type*), 2826
lwm2m_engine_pause (*C function*), 2846
lwm2m_engine_resume (*C function*), 2846
lwm2m_engine_set_data_cb_t (*C type*), 2827
lwm2m_engine_start (*C function*), 2845
lwm2m_engine_stop (*C function*), 2844
lwm2m_engine_user_cb_t (*C type*), 2828
lwm2m_event_log_set_read_log_data_cb (*C function*), 2834
lwm2m_firmware_get_cancel_cb (*C function*), 2832
lwm2m_firmware_get_cancel_cb_inst (*C function*), 2832
lwm2m_firmware_get_update_cb (*C function*), 2832
lwm2m_firmware_get_update_cb_inst (*C function*), 2833
lwm2m_firmware_get_write_cb (*C function*), 2831
lwm2m_firmware_get_write_cb_inst (*C function*), 2831
lwm2m_firmware_set_cancel_cb (*C function*), 2832
lwm2m_firmware_set_cancel_cb_inst (*C function*), 2832
lwm2m_firmware_set_update_cb (*C function*), 2832
lwm2m_firmware_set_update_cb_inst (*C function*), 2832
lwm2m_firmware_set_write_cb (*C function*), 2831
lwm2m_firmware_set_write_cb_inst (*C function*), 2831
lwm2m_get_bool (*C function*), 2840
lwm2m_get_f64 (*C function*), 2840
lwm2m_get_objlnk (*C function*), 2841
lwm2m_get_opaque (*C function*), 2838
lwm2m_get_res_buf (*C function*), 2843
lwm2m_get_s8 (*C function*), 2840
lwm2m_get_s16 (*C function*), 2840
lwm2m_get_s32 (*C function*), 2840
lwm2m_get_s64 (*C function*), 2840
lwm2m_get_string (*C function*), 2839
lwm2m_get_time (*C function*), 2841
lwm2m_get_u8 (*C function*), 2839
lwm2m_get_u16 (*C function*), 2839
lwm2m_get_u32 (*C function*), 2839
lwm2m_get_u64 (*C function*), 2839
LWM2M_HAS_RES_FLAG (*C macro*), 2826
LWM2M_MAX_PATH_STR_SIZE (*C macro*), 2826
lwm2m_obj_path (*C struct*), 2847
lwm2m_obj_path.level (*C var*), 2848
lwm2m_obj_path.obj_id (*C var*), 2847

lwm2m_obj_path.obj_inst_id (*C var*), 2848
lwm2m_obj_path.res_id (*C var*), 2848
lwm2m_obj_path.res_inst_id (*C var*), 2848
LWM2M_OBJECT_ACCESS_CONTROL_ID (*C macro*), 2821
LWM2M_OBJECT_BINARYAPPDATACONTAINER_ID (*C macro*), 2821
LWM2M_OBJECT_CONNECTIVITY_MONITORING_ID (*C macro*), 2821
LWM2M_OBJECT_CONNECTIVITY_STATISTICS_ID (*C macro*), 2821
LWM2M_OBJECT_DEVICE_ID (*C macro*), 2821
LWM2M_OBJECT_EVENT_LOG_ID (*C macro*), 2821
LWM2M_OBJECT_FIRMWARE_ID (*C macro*), 2821
LWM2M_OBJECT_GATEWAY_ID (*C macro*), 2821
LWM2M_OBJECT_LOCATION_ID (*C macro*), 2821
LWM2M_OBJECT_OSCORE_ID (*C macro*), 2821
LWM2M_OBJECT_PORTFOLIO_ID (*C macro*), 2821
LWM2M_OBJECT_SECURITY_ID (*C macro*), 2821
LWM2M_OBJECT_SERVER_ID (*C macro*), 2821
LWM2M_OBJECT_SOFTWARE_MANAGEMENT_ID (*C macro*), 2821
lwm2m_objlnk (*C struct*), 2850
LWM2M_OBJLNK_MAX_ID (*C macro*), 2826
lwm2m_objlnk.obj_id (*C var*), 2850
lwm2m_objlnk.obj_inst (*C var*), 2850
lwm2m_observe_cb_t (*C type*), 2826
lwm2m_observe_event (*C enum*), 2828
lwm2m_observe_event.LWM2M_OBSERVE_EVENT_NOTIFY_ACK (*C enumerator*), 2828
lwm2m_observe_event.LWM2M_OBSERVE_EVENT_NOTIFY_TIMEOUT (*C enumerator*), 2829
lwm2m_observe_event.LWM2M_OBSERVE_EVENT_OBSERVER_ADDED (*C enumerator*), 2828
lwm2m_observe_event.LWM2M_OBSERVE_EVENT_OBSERVER_REMOVED (*C enumerator*), 2828
lwm2m_path_is_observed (*C function*), 2844
lwm2m_path_log_buf (*C function*), 2846
lwm2m_rd_client_ctx (*C function*), 2847
lwm2m_rd_client_event (*C enum*), 2829
lwm2m_rd_client_event.LWM2M_RD_CLIENT_EVENT_BOOTSTRAP_REG_COMPLETE (*C enumerator*), 2829
lwm2m_rd_client_event.LWM2M_RD_CLIENT_EVENT_BOOTSTRAP_REG_FAILURE (*C enumerator*), 2829
lwm2m_rd_client_event.LWM2M_RD_CLIENT_EVENT_BOOTSTRAP_TRANSFER_COMPLETE (*C enumerator*), 2829
lwm2m_rd_client_event.LWM2M_RD_CLIENT_EVENT_DEREGISTER (*C enumerator*), 2830
lwm2m_rd_client_event.LWM2M_RD_CLIENT_EVENT_DEREGISTER_FAILURE (*C enumerator*), 2830
lwm2m_rd_client_event.LWM2M_RD_CLIENT_EVENT_DISCONNECT (*C enumerator*), 2830
lwm2m_rd_client_event.LWM2M_RD_CLIENT_EVENT_ENGINE_SUSPENDED (*C enumerator*), 2830
lwm2m_rd_client_event.LWM2M_RD_CLIENT_EVENT_NETWORK_ERROR (*C enumerator*), 2830
lwm2m_rd_client_event.LWM2M_RD_CLIENT_EVENT_NONE (*C enumerator*), 2829
lwm2m_rd_client_event.LWM2M_RD_CLIENT_EVENT_QUEUE_MODE_RX_OFF (*C enumerator*), 2830
lwm2m_rd_client_event.LWM2M_RD_CLIENT_EVENT_REG_TIMEOUT (*C enumerator*), 2830
lwm2m_rd_client_event.LWM2M_RD_CLIENT_EVENT_REG_UPDATE (*C enumerator*), 2830
lwm2m_rd_client_event.LWM2M_RD_CLIENT_EVENT_REG_UPDATE_COMPLETE (*C enumerator*), 2830
lwm2m_rd_client_event.LWM2M_RD_CLIENT_EVENT_REGISTRATION_COMPLETE (*C enumerator*), 2829
lwm2m_rd_client_event.LWM2M_RD_CLIENT_EVENT_REGISTRATION_FAILURE (*C enumerator*), 2829
lwm2m_rd_client_event.LWM2M_RD_CLIENT_EVENT_SERVER_DISABLED (*C enumerator*), 2830
LWM2M_RD_CLIENT_FLAG_BOOTSTRAP (*C macro*), 2826
lwm2m_rd_client_start (*C function*), 2845
lwm2m_rd_client_stop (*C function*), 2845
lwm2m_rd_client_update (*C function*), 2846
lwm2m_register_create_callback (*C function*), 2842
lwm2m_register_delete_callback (*C function*), 2843
lwm2m_register_exec_callback (*C function*), 2842
lwm2m_register_post_write_callback (*C function*), 2842
lwm2m_register_pre_write_callback (*C function*), 2841

lwm2m_register_read_callback (*C function*), 2841
lwm2m_register_validate_callback (*C function*), 2841
lwm2m_registry_lock (*C function*), 2835
lwm2m_registry_unlock (*C function*), 2836
LWM2M_RES_DATA_FLAG_RO (*C macro*), 2826
LWM2M_RES_DATA_READ_ONLY (*C macro*), 2826
lwm2m_res_item (*C struct*), 2850
lwm2m_res_item.path (*C var*), 2851
lwm2m_res_item.size (*C var*), 2851
lwm2m_res_item.value (*C var*), 2851
lwm2m_security_mode (*C function*), 2847
lwm2m_security_mode_e (*C enum*), 2830
lwm2m_security_mode_e.LWM2M_SECURITY_CERT (*C enumerator*), 2831
lwm2m_security_mode_e.LWM2M_SECURITY_CERT_EST (*C enumerator*), 2831
lwm2m_security_mode_e.LWM2M_SECURITY_NOSEC (*C enumerator*), 2831
lwm2m_security_mode_e.LWM2M_SECURITY_PSK (*C enumerator*), 2831
lwm2m_security_mode_e.LWM2M_SECURITY_RAW_PK (*C enumerator*), 2831
lwm2m_send_cb (*C function*), 2846
lwm2m_send_cb_t (*C type*), 2828
lwm2m_send_status (*C enum*), 2830
lwm2m_send_status.LWM2M_SEND_STATUS_FAILURE (*C enumerator*), 2830
lwm2m_send_status.LWM2M_SEND_STATUS_SUCCESS (*C enumerator*), 2830
lwm2m_send_status.LWM2M_SEND_STATUS_TIMEOUT (*C enumerator*), 2830
lwm2m_set_bool (*C function*), 2837
lwm2m_set_bulk (*C function*), 2838
lwm2m_set_default_sockopt (*C function*), 2847
lwm2m_set_f64 (*C function*), 2838
lwm2m_set_objlnk (*C function*), 2838
lwm2m_set_opaque (*C function*), 2836
lwm2m_set_res_buf (*C function*), 2843
lwm2m_set_res_data_len (*C function*), 2843
lwm2m_set_s8 (*C function*), 2837
lwm2m_set_s16 (*C function*), 2837
lwm2m_set_s32 (*C function*), 2837
lwm2m_set_s64 (*C function*), 2837
lwm2m_set_string (*C function*), 2836
lwm2m_set_time (*C function*), 2838
lwm2m_set_u8 (*C function*), 2836
lwm2m_set_u16 (*C function*), 2836
lwm2m_set_u32 (*C function*), 2836
lwm2m_set_u64 (*C function*), 2837
lwm2m_socket_fault_cb_t (*C type*), 2826
lwm2m_socket_states (*C enum*), 2829
lwm2m_socket_states.LWM2M_SOCKET_STATE_LAST (*C enumerator*), 2829
lwm2m_socket_states.LWM2M_SOCKET_STATE_NO_DATA (*C enumerator*), 2829
lwm2m_socket_states.LWM2M_SOCKET_STATE_ONE_RESPONSE (*C enumerator*), 2829
lwm2m_socket_states.LWM2M_SOCKET_STATE_ONGOING (*C enumerator*), 2829
lwm2m_swmgmt_install_completed (*C function*), 2834
lwm2m_swmgmt_set_activate_cb (*C function*), 2833
lwm2m_swmgmt_set_deactivate_cb (*C function*), 2833
lwm2m_swmgmt_set_delete_package_cb (*C function*), 2833
lwm2m_swmgmt_set_install_package_cb (*C function*), 2833
lwm2m_swmgmt_set_read_package_version_cb (*C function*), 2834
lwm2m_swmgmt_set_write_package_cb (*C function*), 2834
lwm2m_time_series_elem (*C struct*), 2850
lwm2m_time_series_elem.t (*C var*), 2850
lwm2m_update_device_service_period (*C function*), 2844
lwm2m_update_observer_max_period (*C function*), 2835

lwm2m_update_observer_min_period (*C function*), 2835

M

MACRO_MAP_CAT (*C macro*), 695
MACRO_MAP_CAT_N (*C macro*), 695
MAX (*C macro*), 683
MAX_COAP_MSG_LEN (*C macro*), 2775
MAX_REG_CHAN_NUM (*C macro*), 2717
MAX_SHARED_MULTI_HEAP_ATTR (*C macro*), 578
MB (*C macro*), 685
mb_display_get (*C function*), 3300
mb_display_image (*C function*), 3300
mb_display_mode (*C enum*), 3300
mb_display_mode.MB_DISPLAY_FLAG_LOOP (*C enumerator*), 3300
mb_display_mode.MB_DISPLAY_MODE_DEFAULT (*C enumerator*), 3300
mb_display_mode.MB_DISPLAY_MODE_SCROLL (*C enumerator*), 3300
mb_display_mode.MB_DISPLAY_MODE_SINGLE (*C enumerator*), 3300
mb_display_print (*C function*), 3301
mb_display_stop (*C function*), 3301
MB_IMAGE (*C macro*), 3299
mb_image (*C struct*), 3301
mbox_channel_id_t (*C type*), 3503
mbox_dt_spec (*C struct*), 3506
MBOX_DT_SPEC_GET (*C macro*), 3502
MBOX_DT_SPEC_INST_GET (*C macro*), 3503
mbox_dt_spec.channel_id (*C var*), 3506
mbox_dt_spec.dev (*C var*), 3506
mbox_is_ready_dt (*C function*), 3503
mbox_max_channels_get (*C function*), 3506
mbox_max_channels_get_dt (*C function*), 3506
mbox_msg (*C struct*), 3506
mbox_msg.data (*C var*), 3506
mbox_msg.size (*C var*), 3506
mbox_mtu_get (*C function*), 3504
mbox_mtu_get_dt (*C function*), 3505
mbox_register_callback (*C function*), 3504
mbox_register_callback_dt (*C function*), 3504
mbox_send (*C function*), 3503
mbox_send_dt (*C function*), 3504
mbox_set_enabled (*C function*), 3505
mbox_set_enabled_dt (*C function*), 3505
mcuboot_img_header (*C struct*), 824
mcuboot_img_header_v1 (*C struct*), 824
mcuboot_img_header_v1.image_size (*C var*), 824
mcuboot_img_header_v1.sem_ver (*C var*), 824
mcuboot_img_header.h (*C var*), 825
mcuboot_img_header.mcuboot_version (*C var*), 824
mcuboot_img_header.v1 (*C var*), 825
mcuboot_img_sem_ver (*C struct*), 824
mcuboot_swap_type (*C function*), 823
mcuboot_swap_type_multi (*C function*), 823
mcumgr_err_t (*C enum*), 752
mcumgr_err_t.MGMT_ERR_EACCESSDENIED (*C enumerator*), 753
mcumgr_err_t.MGMT_ERR_EBADSTATE (*C enumerator*), 753
mcumgr_err_t.MGMT_ERR_EBUSY (*C enumerator*), 753
mcumgr_err_t.MGMT_ERR_ECORRUPT (*C enumerator*), 753
mcumgr_err_t.MGMT_ERR_EINVAL (*C enumerator*), 753
mcumgr_err_t.MGMT_ERR_EMSGSIZE (*C enumerator*), 753

mcumgr_err_t.MGMT_ERR_ENOENT (*C enumerator*), 753
mcumgr_err_t.MGMT_ERR_ENOMEM (*C enumerator*), 753
mcumgr_err_t.MGMT_ERR_ENOTSUP (*C enumerator*), 753
mcumgr_err_t.MGMT_ERR_EOK (*C enumerator*), 752
mcumgr_err_t.MGMT_ERR_EPERUSER (*C enumerator*), 753
mcumgr_err_t.MGMT_ERR_ETIMEOUT (*C enumerator*), 753
mcumgr_err_t.MGMT_ERR_EUNKNOWN (*C enumerator*), 753
mcumgr_err_t.MGMT_ERR_UNSUPPORTED_TOO_NEW (*C enumerator*), 753
mcumgr_err_t.MGMT_ERR_UNSUPPORTED_TOO_OLD (*C enumerator*), 753
mcumgr_group_t (*C enum*), 751
mcumgr_group_t.MGMT_GROUP_ID_CRASH (*C enumerator*), 752
mcumgr_group_t.MGMT_GROUP_ID_FS (*C enumerator*), 752
mcumgr_group_t.MGMT_GROUP_ID_IMAGE (*C enumerator*), 752
mcumgr_group_t.MGMT_GROUP_ID_LOG (*C enumerator*), 752
mcumgr_group_t.MGMT_GROUP_ID_OS (*C enumerator*), 752
mcumgr_group_t.MGMT_GROUP_ID_PERUSER (*C enumerator*), 752
mcumgr_group_t.MGMT_GROUP_ID_RUN (*C enumerator*), 752
mcumgr_group_t.MGMT_GROUP_ID_SETTINGS (*C enumerator*), 752
mcumgr_group_t.MGMT_GROUP_ID_SHELL (*C enumerator*), 752
mcumgr_group_t.MGMT_GROUP_ID_SPLIT (*C enumerator*), 752
mcumgr_group_t.MGMT_GROUP_ID_STAT (*C enumerator*), 752
mcumgr_group_t.ZEPHYR_MGMT_GRP_BASIC (*C enumerator*), 752
mcumgr_op_t (*C enum*), 751
mcumgr_op_t.MGMT_OP_READ (*C enumerator*), 751
mcumgr_op_t.MGMT_OP_READ_RSP (*C enumerator*), 751
mcumgr_op_t.MGMT_OP_WRITE (*C enumerator*), 751
mcumgr_op_t.MGMT_OP_WRITE_RSP (*C enumerator*), 751
mdio_bus_disable (*C function*), 3481
mdio_bus_enable (*C function*), 3481
mdio_read (*C function*), 3481
mdio_read_c45 (*C function*), 3482
mdio_write (*C function*), 3481
mdio_write_c45 (*C function*), 3482
MEDIA_PROXY_CMD_CANNOT_BE_COMPLETED (*C macro*), 1848
MEDIA_PROXY_CMD_NOT_SUPPORTED (*C macro*), 1848
MEDIA_PROXY_CMD_PLAYER_INACTIVE (*C macro*), 1848
MEDIA_PROXY_CMD_SUCCESS (*C macro*), 1848
media_proxy_ctrl_cbs (*C struct*), 1860
media_proxy_ctrl_cbs.command_recv (*C var*), 1866
media_proxy_ctrl_cbs.command_send (*C var*), 1865
media_proxy_ctrl_cbs.commands_supported_recv (*C var*), 1866
media_proxy_ctrl_cbs.content_ctrl_id_recv (*C var*), 1867
media_proxy_ctrl_cbs.current_group_id_recv (*C var*), 1864
media_proxy_ctrl_cbs.current_group_id_write (*C var*), 1864
media_proxy_ctrl_cbs.current_track_id_recv (*C var*), 1863
media_proxy_ctrl_cbs.current_track_id_write (*C var*), 1863
media_proxy_ctrl_cbs.icon_id_recv (*C var*), 1861
media_proxy_ctrl_cbs.icon_url_recv (*C var*), 1861
media_proxy_ctrl_cbs.local_player_instance (*C var*), 1860
media_proxy_ctrl_cbs.media_state_recv (*C var*), 1865
media_proxy_ctrl_cbs.next_track_id_recv (*C var*), 1863
media_proxy_ctrl_cbs.next_track_id_write (*C var*), 1864
media_proxy_ctrl_cbs.parent_group_id_recv (*C var*), 1864
media_proxy_ctrl_cbs.playback_speed_recv (*C var*), 1862
media_proxy_ctrl_cbs.playback_speed_write (*C var*), 1862
media_proxy_ctrl_cbs.player_name_recv (*C var*), 1860
media_proxy_ctrl_cbs.playing_order_recv (*C var*), 1865
media_proxy_ctrl_cbs.playing_order_write (*C var*), 1865

media_proxy_ctrl_cbs.playing_orders_supported_recv (*C var*), 1865
media_proxy_ctrl_cbs.search_recv (*C var*), 1866
media_proxy_ctrl_cbs.search_results_id_recv (*C var*), 1867
media_proxy_ctrl_cbs.search_send (*C var*), 1866
media_proxy_ctrl_cbs.seeking_speed_recv (*C var*), 1863
media_proxy_ctrl_cbs.track_changed_recv (*C var*), 1861
media_proxy_ctrl_cbs.track_duration_recv (*C var*), 1861
media_proxy_ctrl_cbs.track_position_recv (*C var*), 1862
media_proxy_ctrl_cbs.track_position_write (*C var*), 1862
media_proxy_ctrl_cbs.track_segments_id_recv (*C var*), 1863
media_proxy_ctrl_cbs.track_title_recv (*C var*), 1861
media_proxy_ctrl_discover_player (*C function*), 1850
media_proxy_ctrl_get_commands_supported (*C function*), 1855
media_proxy_ctrl_get_content_ctrl_id (*C function*), 1856
media_proxy_ctrl_get_current_group_id (*C function*), 1854
media_proxy_ctrl_get_current_track_id (*C function*), 1852
media_proxy_ctrl_get_icon_id (*C function*), 1850
media_proxy_ctrl_get_icon_url (*C function*), 1850
media_proxy_ctrl_get_media_state (*C function*), 1855
media_proxy_ctrl_get_next_track_id (*C function*), 1853
media_proxy_ctrl_get_parent_group_id (*C function*), 1853
media_proxy_ctrl_get_playback_speed (*C function*), 1851
media_proxy_ctrl_get_player_name (*C function*), 1850
media_proxy_ctrl_get_playing_order (*C function*), 1854
media_proxy_ctrl_get_playing_orders_supported (*C function*), 1854
media_proxy_ctrl_get_search_results_id (*C function*), 1856
media_proxy_ctrl_get_seeking_speed (*C function*), 1852
media_proxy_ctrl_get_track_duration (*C function*), 1851
media_proxy_ctrl_get_track_position (*C function*), 1851
media_proxy_ctrl_get_track_segments_id (*C function*), 1852
media_proxy_ctrl_get_track_title (*C function*), 1851
media_proxy_ctrl_register (*C function*), 1850
media_proxy_ctrl_send_command (*C function*), 1855
media_proxy_ctrl_send_search (*C function*), 1855
media_proxy_ctrl_set_current_group_id (*C function*), 1854
media_proxy_ctrl_set_current_track_id (*C function*), 1853
media_proxy_ctrl_set_next_track_id (*C function*), 1853
media_proxy_ctrl_set_playback_speed (*C function*), 1852
media_proxy_ctrl_set_playing_order (*C function*), 1854
media_proxy_ctrl_set_track_position (*C function*), 1851
MEDIA_PROXY_GROUP_OBJECT_GROUP_TYPE (*C macro*), 1849
MEDIA_PROXY_GROUP_OBJECT_TRACK_TYPE (*C macro*), 1849
MEDIA_PROXY_OP_FAST_FORWARD (*C macro*), 1845
MEDIA_PROXY_OP_FAST_REWIND (*C macro*), 1845
MEDIA_PROXY_OP_FIRST_GROUP (*C macro*), 1846
MEDIA_PROXY_OP_FIRST_SEGMENT (*C macro*), 1846
MEDIA_PROXY_OP_FIRST_TRACK (*C macro*), 1846
MEDIA_PROXY_OP_GOTO_GROUP (*C macro*), 1847
MEDIA_PROXY_OP_GOTO_SEGMENT (*C macro*), 1846
MEDIA_PROXY_OP_GOTO_TRACK (*C macro*), 1846
MEDIA_PROXY_OP_LAST_GROUP (*C macro*), 1847
MEDIA_PROXY_OP_LAST_SEGMENT (*C macro*), 1846
MEDIA_PROXY_OP_LAST_TRACK (*C macro*), 1846
MEDIA_PROXY_OP_MOVE_RELATIVE (*C macro*), 1846
MEDIA_PROXY_OP_NEXT_GROUP (*C macro*), 1846
MEDIA_PROXY_OP_NEXT_SEGMENT (*C macro*), 1846
MEDIA_PROXY_OP_NEXT_TRACK (*C macro*), 1846
MEDIA_PROXY_OP_PAUSE (*C macro*), 1845

MEDIA_PROXY_OP_PLAY (*C macro*), 1845
MEDIA_PROXY_OP_PREV_GROUP (*C macro*), 1846
MEDIA_PROXY_OP_PREV_SEGMENT (*C macro*), 1846
MEDIA_PROXY_OP_PREV_TRACK (*C macro*), 1846
MEDIA_PROXY_OP_STOP (*C macro*), 1846
MEDIA_PROXY_OP_SUP_FAST_FORWARD (*C macro*), 1847
MEDIA_PROXY_OP_SUP_FAST_REWIND (*C macro*), 1847
MEDIA_PROXY_OP_SUP_FIRST_GROUP (*C macro*), 1848
MEDIA_PROXY_OP_SUP_FIRST_SEGMENT (*C macro*), 1847
MEDIA_PROXY_OP_SUP_FIRST_TRACK (*C macro*), 1848
MEDIA_PROXY_OP_SUP_GOTO_GROUP (*C macro*), 1848
MEDIA_PROXY_OP_SUP_GOTO_SEGMENT (*C macro*), 1847
MEDIA_PROXY_OP_SUP_GOTO_TRACK (*C macro*), 1848
MEDIA_PROXY_OP_SUP_LAST_GROUP (*C macro*), 1848
MEDIA_PROXY_OP_SUP_LAST_SEGMENT (*C macro*), 1847
MEDIA_PROXY_OP_SUP_LAST_TRACK (*C macro*), 1848
MEDIA_PROXY_OP_SUP_MOVE_RELATIVE (*C macro*), 1847
MEDIA_PROXY_OP_SUP_NEXT_GROUP (*C macro*), 1848
MEDIA_PROXY_OP_SUP_NEXT_SEGMENT (*C macro*), 1847
MEDIA_PROXY_OP_SUP_NEXT_TRACK (*C macro*), 1847
MEDIA_PROXY_OP_SUP_PAUSE (*C macro*), 1847
MEDIA_PROXY_OP_SUP_PLAY (*C macro*), 1847
MEDIA_PROXY_OP_SUP_PREV_GROUP (*C macro*), 1848
MEDIA_PROXY_OP_SUP_PREV_SEGMENT (*C macro*), 1847
MEDIA_PROXY_OP_SUP_PREV_TRACK (*C macro*), 1847
MEDIA_PROXY_OP_SUP_STOP (*C macro*), 1847
MEDIA_PROXY_OPCODES_SUPPORTED_LEN (*C macro*), 1850
media_proxy_pl_calls (*C struct*), 1867
media_proxy_pl_calls.get_commands_supported (*C var*), 1871
media_proxy_pl_calls.get_content_ctrl_id (*C var*), 1871
media_proxy_pl_calls.get_current_group_id (*C var*), 1870
media_proxy_pl_calls.get_current_track_id (*C var*), 1869
media_proxy_pl_calls.get_icon_id (*C var*), 1867
media_proxy_pl_calls.get_icon_url (*C var*), 1868
media_proxy_pl_calls.get_media_state (*C var*), 1870
media_proxy_pl_calls.get_next_track_id (*C var*), 1869
media_proxy_pl_calls.get_parent_group_id (*C var*), 1870
media_proxy_pl_calls.get_playback_speed (*C var*), 1868
media_proxy_pl_calls.get_player_name (*C var*), 1867
media_proxy_pl_calls.get_playing_order (*C var*), 1870
media_proxy_pl_calls.get_playing_orders_supported (*C var*), 1870
media_proxy_pl_calls.get_search_results_id (*C var*), 1871
media_proxy_pl_calls.get_seeking_speed (*C var*), 1869
media_proxy_pl_calls.get_track_duration (*C var*), 1868
media_proxy_pl_calls.get_track_position (*C var*), 1868
media_proxy_pl_calls.get_track_segments_id (*C var*), 1869
media_proxy_pl_calls.get_track_title (*C var*), 1868
media_proxy_pl_calls.send_command (*C var*), 1871
media_proxy_pl_calls.send_search (*C var*), 1871
media_proxy_pl_calls.set_current_group_id (*C var*), 1870
media_proxy_pl_calls.set_current_track_id (*C var*), 1869
media_proxy_pl_calls.set_next_track_id (*C var*), 1869
media_proxy_pl_calls.set_playback_speed (*C var*), 1868
media_proxy_pl_calls.set_playing_order (*C var*), 1870
media_proxy_pl_calls.set_track_position (*C var*), 1868
media_proxy_pl_command_cb (*C function*), 1858
media_proxy_pl_commands_supported_cb (*C function*), 1858
media_proxy_pl_current_group_id_cb (*C function*), 1858

media_proxy_pl_current_track_id_cb (*C function*), 1857
media_proxy_pl_icon_url_cb (*C function*), 1856
media_proxy_pl_init (*C function*), 1856
media_proxy_pl_media_state_cb (*C function*), 1858
media_proxy_pl_name_cb (*C function*), 1856
media_proxy_pl_next_track_id_cb (*C function*), 1858
media_proxy_pl_parent_group_id_cb (*C function*), 1858
media_proxy_pl_playback_speed_cb (*C function*), 1857
media_proxy_pl_playing_order_cb (*C function*), 1858
media_proxy_pl_register (*C function*), 1856
media_proxy_pl_search_cb (*C function*), 1859
media_proxy_pl_search_results_id_cb (*C function*), 1859
media_proxy_pl_seeking_speed_cb (*C function*), 1857
media_proxy_pl_track_changed_cb (*C function*), 1857
media_proxy_pl_track_duration_cb (*C function*), 1857
media_proxy_pl_track_position_cb (*C function*), 1857
media_proxy_pl_track_title_cb (*C function*), 1857
MEDIA_PROXY_PLAYBACK_SPEED_DOUBLE (*C macro*), 1843
MEDIA_PROXY_PLAYBACK_SPEED_HALF (*C macro*), 1843
MEDIA_PROXY_PLAYBACK_SPEED_MAX (*C macro*), 1843
MEDIA_PROXY_PLAYBACK_SPEED_MIN (*C macro*), 1843
MEDIA_PROXY_PLAYBACK_SPEED_QUARTER (*C macro*), 1843
MEDIA_PROXY_PLAYBACK_SPEED_UNITY (*C macro*), 1843
MEDIA_PROXY_PLAYING_ORDER_INORDER_ONCE (*C macro*), 1844
MEDIA_PROXY_PLAYING_ORDER_INORDER_REPEAT (*C macro*), 1844
MEDIA_PROXY_PLAYING_ORDER_NEWEST_ONCE (*C macro*), 1844
MEDIA_PROXY_PLAYING_ORDER_NEWEST_REPEAT (*C macro*), 1844
MEDIA_PROXY_PLAYING_ORDER_OLDEST_ONCE (*C macro*), 1844
MEDIA_PROXY_PLAYING_ORDER_OLDEST_REPEAT (*C macro*), 1844
MEDIA_PROXY_PLAYING_ORDER_SHUFFLE_ONCE (*C macro*), 1844
MEDIA_PROXY_PLAYING_ORDER_SHUFFLE_REPEAT (*C macro*), 1844
MEDIA_PROXY_PLAYING_ORDER_SINGLE_ONCE (*C macro*), 1844
MEDIA_PROXY_PLAYING_ORDER_SINGLE_REPEAT (*C macro*), 1844
MEDIA_PROXY_PLAYING_ORDERS_SUPPORTED_INORDER_ONCE (*C macro*), 1844
MEDIA_PROXY_PLAYING_ORDERS_SUPPORTED_INORDER_REPEAT (*C macro*), 1844
MEDIA_PROXY_PLAYING_ORDERS_SUPPORTED_NEWEST_ONCE (*C macro*), 1845
MEDIA_PROXY_PLAYING_ORDERS_SUPPORTED_NEWEST_REPEAT (*C macro*), 1845
MEDIA_PROXY_PLAYING_ORDERS_SUPPORTED_OLDEST_ONCE (*C macro*), 1844
MEDIA_PROXY_PLAYING_ORDERS_SUPPORTED_OLDEST_REPEAT (*C macro*), 1845
MEDIA_PROXY_PLAYING_ORDERS_SUPPORTED_SHUFFLE_ONCE (*C macro*), 1845
MEDIA_PROXY_PLAYING_ORDERS_SUPPORTED_SHUFFLE_REPEAT (*C macro*), 1845
MEDIA_PROXY_PLAYING_ORDERS_SUPPORTED_SINGLE_ONCE (*C macro*), 1844
MEDIA_PROXY_PLAYING_ORDERS_SUPPORTED_SINGLE_REPEAT (*C macro*), 1844
MEDIA_PROXY_SEARCH_FAILURE (*C macro*), 1849
MEDIA_PROXY_SEARCH_SUCCESS (*C macro*), 1849
MEDIA_PROXY_SEARCH_TYPE_ALBUM_NAME (*C macro*), 1849
MEDIA_PROXY_SEARCH_TYPE_ARTIST_NAME (*C macro*), 1849
MEDIA_PROXY_SEARCH_TYPE_EARLIEST_YEAR (*C macro*), 1849
MEDIA_PROXY_SEARCH_TYPE_GENRE (*C macro*), 1849
MEDIA_PROXY_SEARCH_TYPE_GROUP_NAME (*C macro*), 1849
MEDIA_PROXY_SEARCH_TYPE_LATEST_YEAR (*C macro*), 1849
MEDIA_PROXY_SEARCH_TYPE_ONLY_GROUPS (*C macro*), 1849
MEDIA_PROXY_SEARCH_TYPE_ONLY_TRACKS (*C macro*), 1849
MEDIA_PROXY_SEARCH_TYPE_TRACK_NAME (*C macro*), 1848
MEDIA_PROXY_SEEKING_SPEED_FACTOR_MAX (*C macro*), 1843
MEDIA_PROXY_SEEKING_SPEED_FACTOR_MIN (*C macro*), 1843
MEDIA_PROXY_SEEKING_SPEED_FACTOR_ZERO (*C macro*), 1843
MEDIA_PROXY_STATE_INACTIVE (*C macro*), 1845

MEDIA_PROXY_STATE_LAST (*C macro*), 1845
MEDIA_PROXY_STATE_PAUSED (*C macro*), 1845
MEDIA_PROXY_STATE_PLAYING (*C macro*), 1845
MEDIA_PROXY_STATE_SEEKING (*C macro*), 1845
mem_attr_check_buf (*C function*), 1014
mem_attr_get_regions (*C function*), 1014
mem_attr_heap_aligned_alloc (*C function*), 1016
mem_attr_heap_alloc (*C function*), 1016
mem_attr_heap_free (*C function*), 1016
mem_attr_heap_get_region (*C function*), 1016
mem_attr_heap_pool_init (*C function*), 1016
mem_attr_region_t (*C struct*), 1015
mem_attr_region_t.dt_addr (*C var*), 1015
mem_attr_region_t.dt_attr (*C var*), 1015
mem_attr_region_t.dt_name (*C var*), 1015
mem_attr_region_t.dt_size (*C var*), 1015
mem_xor_32 (*C function*), 699
mem_xor_128 (*C function*), 699
mem_xor_n (*C function*), 699
METAWARE_ROOT, 290
mgmt_alloc_rsp_fn (*C type*), 751
mgmt_callback (*C struct*), 772
mgmt_callback_notify (*C function*), 772
mgmt_callback_register (*C function*), 772
mgmt_callback_unregister (*C function*), 772
mgmt_callback.callback (*C var*), 772
mgmt_callback.event_id (*C var*), 772
mgmt_callback.node (*C var*), 772
mgmt_cb (*C type*), 768
MGMT_CB_ERROR_RET (*C macro*), 768
mgmt_cb_groups (*C enum*), 769
mgmt_cb_groups.MGMT_EVT_GRP_ALL (*C enumerator*), 769
mgmt_cb_groups.MGMT_EVT_GRP_FS (*C enumerator*), 769
mgmt_cb_groups.MGMT_EVT_GRP_IMG (*C enumerator*), 769
mgmt_cb_groups.MGMT_EVT_GRP_OS (*C enumerator*), 769
mgmt_cb_groups.MGMT_EVT_GRP_SETTINGS (*C enumerator*), 769
mgmt_cb_groups.MGMT_EVT_GRP_SMP (*C enumerator*), 769
mgmt_cb_groups.MGMT_EVT_GRP_USER_CUSTOM_START (*C enumerator*), 769
mgmt_cb_return (*C enum*), 769
mgmt_cb_return.MGMT_CB_ERROR_ERR (*C enumerator*), 769
mgmt_cb_return.MGMT_CB_ERROR_RC (*C enumerator*), 769
mgmt_cb_return.MGMT_CB_OK (*C enumerator*), 769
MGMT_CTXT_RC_RSN (*C macro*), 750
MGMT_CTXT_SET_RC_RSN (*C macro*), 750
MGMT_EVT_GET_GROUP (*C macro*), 768
MGMT_EVT_GET_ID (*C macro*), 768
mgmt_evt_get_index (*C function*), 772
mgmt_evt_op_cmd_arg (*C struct*), 773
mgmt_evt_op_cmd_arg.err (*C var*), 773
mgmt_evt_op_cmd_arg.group (*C var*), 773
mgmt_evt_op_cmd_arg.id (*C var*), 773
mgmt_evt_op_cmd_arg.op (*C var*), 773
mgmt_evt_op_cmd_arg.status (*C var*), 773
mgmt_find_group (*C function*), 754
mgmt_find_handler (*C function*), 754
mgmt_get_handler (*C function*), 754
mgmt_group (*C struct*), 754
mgmt_group.mg_group_id (*C var*), 755

mgmt_group.mg_handlers (*C var*), 754
mgmt_group.node (*C var*), 754
mgmt_handler (*C struct*), 754
mgmt_handler_fn (*C type*), 751
MGMT_HDR_SIZE (*C macro*), 750
mgmt_register_group (*C function*), 753
mgmt_reset_buf_fn (*C type*), 751
MGMT_RETURN_CHECK (*C macro*), 750
mgmt_unregister_group (*C function*), 754
MHZ (*C macro*), 685
MII_1KSTSR (*C macro*), 2655
MII_1KTCR (*C macro*), 2655
MII_ADVERTISE_10_FULL (*C macro*), 2658
MII_ADVERTISE_10_HALF (*C macro*), 2658
MII_ADVERTISE_100_FULL (*C macro*), 2657
MII_ADVERTISE_100_HALF (*C macro*), 2658
MII_ADVERTISE_100BASE_T4 (*C macro*), 2657
MII_ADVERTISE_1000_FULL (*C macro*), 2658
MII_ADVERTISE_1000_HALF (*C macro*), 2658
MII_ADVERTISE_ALL (*C macro*), 2658
MII_ADVERTISE_ASYM_PAUSE (*C macro*), 2657
MII_ADVERTISE_LPACK (*C macro*), 2657
MII_ADVERTISE_NEXT_PAGE (*C macro*), 2657
MII_ADVERTISE_PAUSE (*C macro*), 2657
MII_ADVERTISE_REMOTE_FAULT (*C macro*), 2657
MII_ADVERTISE_SEL_IEEE_802_3 (*C macro*), 2658
MII_ADVERTISE_SEL_MASK (*C macro*), 2658
MII_ANAR (*C macro*), 2655
MII_ANER (*C macro*), 2655
MII_ANLPAR (*C macro*), 2655
MII_ANLPRNPR (*C macro*), 2655
MII_ANNPTR (*C macro*), 2655
MII_BMCR (*C macro*), 2655
MII_BMCR_AUTONEG_ENABLE (*C macro*), 2656
MII_BMCR_AUTONEG_RESTART (*C macro*), 2656
MII_BMCR_DUPLEX_MODE (*C macro*), 2656
MII_BMCR_ISOLATE (*C macro*), 2656
MII_BMCR_LOOPBACK (*C macro*), 2655
MII_BMCR_POWER_DOWN (*C macro*), 2656
MII_BMCR_RESET (*C macro*), 2655
MII_BMCR_SPEED_10 (*C macro*), 2656
MII_BMCR_SPEED_100 (*C macro*), 2656
MII_BMCR_SPEED_1000 (*C macro*), 2656
MII_BMCR_SPEED_LSB (*C macro*), 2655
MII_BMCR_SPEED_MASK (*C macro*), 2656
MII_BMCR_SPEED_MSB (*C macro*), 2656
MII_BMSR (*C macro*), 2655
MII_BMSR_10_FULL (*C macro*), 2656
MII_BMSR_10_HALF (*C macro*), 2656
MII_BMSR_100BASE_T2_FULL (*C macro*), 2656
MII_BMSR_100BASE_T2_HALF (*C macro*), 2657
MII_BMSR_100BASE_T4 (*C macro*), 2656
MII_BMSR_100BASE_X_FULL (*C macro*), 2656
MII_BMSR_100BASE_X_HALF (*C macro*), 2656
MII_BMSR_AUTONEG_ABILITY (*C macro*), 2657
MII_BMSR_AUTONEG_COMPLETE (*C macro*), 2657
MII_BMSR_EXTEND_CAPAB (*C macro*), 2657
MII_BMSR_EXTEND_STATUS (*C macro*), 2657

MII_BMSR_JABBER_DETECT (*C macro*), 2657
MII_BMSR_LINK_STATUS (*C macro*), 2657
MII_BMSR_MF_PREAMB_SUPPR (*C macro*), 2657
MII_BMSR_REMOTE_FAULT (*C macro*), 2657
MII_ESTAT (*C macro*), 2655
MII_ESTAT_1000BASE_T_FULL (*C macro*), 2658
MII_ESTAT_1000BASE_T_HALF (*C macro*), 2658
MII_ESTAT_1000BASE_X_FULL (*C macro*), 2658
MII_ESTAT_1000BASE_X_HALF (*C macro*), 2658
MII_MMD_AADR (*C macro*), 2655
MII_MMD_ACR (*C macro*), 2655
MII_PHYID1R (*C macro*), 2655
MII_PHYID2R (*C macro*), 2655
MIN (*C macro*), 683
MIN_PER_HOUR (*C macro*), 476
mipi_dbi_command_read (*C function*), 3486
mipi_dbi_command_write (*C function*), 3486
mipi_dbi_config (*C struct*), 3488
MIPI_DBI_CONFIG_DT (*C macro*), 3483
MIPI_DBI_CONFIG_DT_INST (*C macro*), 3484
mipi_dbi_config.config (*C var*), 3488
mipi_dbi_config.mode (*C var*), 3488
mipi_dbi_driver_api (*C struct*), 3488
MIPI_DBI_MODE_6800_BUS_8_BIT (*C macro*), 3485
MIPI_DBI_MODE_6800_BUS_9_BIT (*C macro*), 3485
MIPI_DBI_MODE_6800_BUS_16_BIT (*C macro*), 3485
MIPI_DBI_MODE_8080_BUS_8_BIT (*C macro*), 3486
MIPI_DBI_MODE_8080_BUS_9_BIT (*C macro*), 3486
MIPI_DBI_MODE_8080_BUS_16_BIT (*C macro*), 3485
MIPI_DBI_MODE_SPI_3WIRE (*C macro*), 3484
MIPI_DBI_MODE_SPI_4WIRE (*C macro*), 3484
mipi_dbi_release (*C function*), 3487
mipi_dbi_reset (*C function*), 3487
MIPI_DBI_SPI_CONFIG_DT (*C macro*), 3483
MIPI_DBI_SPI_CONFIG_DT_INST (*C macro*), 3483
mipi_dbi_write_display (*C function*), 3487
mipi_dsi_attach (*C function*), 3490
MIPI_DSI_CLOCK_NON_CONTINUOUS (*C macro*), 3489
mipi_dsi_dcs_read (*C function*), 3491
mipi_dsi_dcs_write (*C function*), 3491
mipi_dsi_detach (*C function*), 3491
mipi_dsi_device (*C struct*), 3492
mipi_dsi_device.data_lanes (*C var*), 3492
mipi_dsi_device.mode_flags (*C var*), 3492
mipi_dsi_device.pixfmt (*C var*), 3492
mipi_dsi_device.timings (*C var*), 3492
mipi_dsi_driver_api (*C struct*), 3493
mipi_dsi_generic_read (*C function*), 3490
mipi_dsi_generic_write (*C function*), 3490
MIPI_DSI_MODE_EOT_PACKET (*C macro*), 3489
MIPI_DSI_MODE_LPM (*C macro*), 3489
MIPI_DSI_MODE_VIDEO (*C macro*), 3488
MIPI_DSI_MODE_VIDEO_AUTO_VERT (*C macro*), 3489
MIPI_DSI_MODE_VIDEO_BURST (*C macro*), 3488
MIPI_DSI_MODE_VIDEO_HBP (*C macro*), 3489
MIPI_DSI_MODE_VIDEO_HFP (*C macro*), 3489
MIPI_DSI_MODE_VIDEO_HSA (*C macro*), 3489
MIPI_DSI_MODE_VIDEO_HSE (*C macro*), 3489

MIPI_DSI_MODE_VIDEO_SYNC_PULSE (*C macro*), 3489
MIPI_DSI_MODE_VSYNC_FLUSH (*C macro*), 3489
mipi_dsi_msg (*C struct*), 3492
MIPI_DSI_MSG_USE_LPM (*C macro*), 3490
mipi_dsi_msg.cmd (*C var*), 3493
mipi_dsi_msg.flags (*C var*), 3493
mipi_dsi_msg.rx_buf (*C var*), 3493
mipi_dsi_msg.rx_len (*C var*), 3493
mipi_dsi_msg.tx_buf (*C var*), 3493
mipi_dsi_msg.tx_len (*C var*), 3493
mipi_dsi_msg.type (*C var*), 3492
MIPI_DSI_PIXFMT_RGB565 (*C macro*), 3489
MIPI_DSI_PIXFMT_RGB666 (*C macro*), 3489
MIPI_DSI_PIXFMT_RGB666_PACKED (*C macro*), 3489
MIPI_DSI_PIXFMT_RGB888 (*C macro*), 3489
mipi_dsi_timings (*C struct*), 3491
mipi_dsi_timings.hactive (*C var*), 3491
mipi_dsi_timings.hbp (*C var*), 3492
mipi_dsi_timings.hfp (*C var*), 3492
mipi_dsi_timings.hsync (*C var*), 3492
mipi_dsi_timings.vactive (*C var*), 3492
mipi_dsi_timings.vbp (*C var*), 3492
mipi_dsi_timings.vfp (*C var*), 3492
mipi_dsi_timings.vsync (*C var*), 3492
mipi_dsi_transfer (*C function*), 3490
mipi_stp_decoder_cb (*C type*), 744
mipi_stp_decoder_config (*C struct*), 746
mipi_stp_decoder_config.cb (*C var*), 746
mipi_stp_decoder_config.start_out_of_sync (*C var*), 746
mipi_stp_decoder_ctrl_type (*C enum*), 744
mipi_stp_decoder_ctrl_type.STP_DATA4 (*C enumerator*), 744
mipi_stp_decoder_ctrl_type.STP_DATA8 (*C enumerator*), 744
mipi_stp_decoder_ctrl_type.STP_DATA16 (*C enumerator*), 744
mipi_stp_decoder_ctrl_type.STP_DATA32 (*C enumerator*), 744
mipi_stp_decoder_ctrl_type.STP_DATA64 (*C enumerator*), 744
mipi_stp_decoder_ctrl_type.STP_DECODER_ASYNC (*C enumerator*), 745
mipi_stp_decoder_ctrl_type.STP_DECODER_CHANNEL (*C enumerator*), 744
mipi_stp_decoder_ctrl_type.STP_DECODER_FLAG (*C enumerator*), 745
mipi_stp_decoder_ctrl_type.STP_DECODER_FREQ (*C enumerator*), 744
mipi_stp_decoder_ctrl_type.STP_DECODER_GERROR (*C enumerator*), 745
mipi_stp_decoder_ctrl_type.STP_DECODER_MASTER (*C enumerator*), 744
mipi_stp_decoder_ctrl_type.STP_DECODER_MERROR (*C enumerator*), 744
mipi_stp_decoder_ctrl_type.STP_DECODER_NOT_SUPPORTED (*C enumerator*), 745
mipi_stp_decoder_ctrl_type.STP_DECODER_NULL (*C enumerator*), 744
mipi_stp_decoder_ctrl_type.STP_DECODER_VERSION (*C enumerator*), 744
mipi_stp_decoder_data (*C union*), 745
mipi_stp_decoder_data.data (*C var*), 746
mipi_stp_decoder_data.dummy (*C var*), 746
mipi_stp_decoder_data.err (*C var*), 746
mipi_stp_decoder_data.freq (*C var*), 745
mipi_stp_decoder_data.id (*C var*), 745
mipi_stp_decoder_data.ver (*C var*), 746
mipi_stp_decoder_decode (*C function*), 745
mipi_stp_decoder_init (*C function*), 745
mipi_stp_decoder_sync_loss (*C function*), 745
MissingProgram, 196
modbus_adu (*C struct*), 1025
modbus_adu.crc (*C var*), 1026

modbus_adu.data (*C var*), 1026
modbus_adu.fc (*C var*), 1026
modbus_adu.length (*C var*), 1026
modbus_adu.proto_id (*C var*), 1026
modbus_adu.trans_id (*C var*), 1026
modbus_adu.unit_id (*C var*), 1026
modbus_custom_cb_t (*C type*), 1019
MODBUS_CUSTOM_FC_DEFINE (*C macro*), 1018
modbus_disable (*C function*), 1024
MODBUS_EXC_ACK (*C macro*), 1018
MODBUS_EXC_GW_PATH_UNAVAILABLE (*C macro*), 1018
MODBUS_EXC_GW_TARGET_FAILED_TO_RESP (*C macro*), 1018
MODBUS_EXC_ILLEGAL_DATA_ADDR (*C macro*), 1018
MODBUS_EXC_ILLEGAL_DATA_VAL (*C macro*), 1018
MODBUS_EXC_ILLEGAL_FC (*C macro*), 1018
MODBUS_EXC_MEM_PARITY_ERROR (*C macro*), 1018
MODBUS_EXC_NONE (*C macro*), 1018
MODBUS_EXC_SERVER_DEVICE_BUSY (*C macro*), 1018
MODBUS_EXC_SERVER_DEVICE_FAILURE (*C macro*), 1018
modbus_iface_get_by_name (*C function*), 1024
modbus_iface_param (*C struct*), 1027
modbus_iface_param.mode (*C var*), 1027
modbus_iface_param.rawcb (*C var*), 1028
modbus_iface_param.rx_timeout (*C var*), 1027
modbus_iface_param.serial (*C var*), 1028
modbus_init_client (*C function*), 1024
modbus_init_server (*C function*), 1024
MODBUS_MBAP_AND_FC_LENGTH (*C macro*), 1018
MODBUS_MBAP_LENGTH (*C macro*), 1018
modbus_mode (*C enum*), 1019
modbus_mode.MODBUS_MODE_ASCII (*C enumerator*), 1019
modbus_mode.MODBUS_MODE_RAW (*C enumerator*), 1020
modbus_mode.MODBUS_MODE_RTU (*C enumerator*), 1019
modbus_raw_backend_txn (*C function*), 1025
modbus_raw_cb (*C struct*), 1027
modbus_raw_cb_t (*C type*), 1019
modbus_raw_get_header (*C function*), 1025
modbus_raw_put_header (*C function*), 1025
modbus_raw_set_server_failure (*C function*), 1025
modbus_raw_submit_rx (*C function*), 1024
modbus_read_coils (*C function*), 1020
modbus_read_dinputs (*C function*), 1020
modbus_read_holding_regs (*C function*), 1021
modbus_read_holding_regs_fp (*C function*), 1023
modbus_read_input_regs (*C function*), 1021
modbus_register_user_fc (*C function*), 1025
modbus_request_diagnostic (*C function*), 1022
modbus_serial_param (*C struct*), 1027
modbus_serial_param.baud (*C var*), 1027
modbus_serial_param.parity (*C var*), 1027
modbus_serial_param.stop_bits_client (*C var*), 1027
modbus_server_param (*C struct*), 1027
modbus_server_param.unit_id (*C var*), 1027
modbus_server_param.user_cb (*C var*), 1027
modbus_user_callbacks (*C struct*), 1026
modbus_user_callbacks.coil_rd (*C var*), 1026
modbus_user_callbacks.coil_wr (*C var*), 1026
modbus_user_callbacks.discrete_input_rd (*C var*), 1026

modbus_user_callbacks.holding_reg_rd (*C var*), 1026
modbus_user_callbacks.holding_reg_rd_fp (*C var*), 1026
modbus_user_callbacks.holding_reg_wr (*C var*), 1026
modbus_user_callbacks.holding_reg_wr_fp (*C var*), 1027
modbus_user_callbacks.input_reg_rd (*C var*), 1026
modbus_user_callbacks.input_reg_rd_fp (*C var*), 1026
modbus_write_coil (*C function*), 1021
modbus_write_coils (*C function*), 1022
modbus_write_holding_reg (*C function*), 1022
modbus_write_holding_regs (*C function*), 1023
modbus_write_holding_regs_fp (*C function*), 1023
modem_cmux_attach (*C function*), 1033
modem_cmux_callback (*C type*), 1032
modem_cmux_config (*C struct*), 1034
modem_cmux_config.callback (*C var*), 1035
modem_cmux_config.receive_buf (*C var*), 1035
modem_cmux_config.receive_buf_size (*C var*), 1035
modem_cmux_config.transmit_buf (*C var*), 1035
modem_cmux_config.transmit_buf_size (*C var*), 1035
modem_cmux_config.user_data (*C var*), 1035
modem_cmux_connect (*C function*), 1033
modem_cmux_connect_async (*C function*), 1033
modem_cmux_disconnect (*C function*), 1034
modem_cmux_disconnect_async (*C function*), 1034
modem_cmux_dlci_config (*C struct*), 1035
modem_cmux_dlci_config.dlci_address (*C var*), 1035
modem_cmux_dlci_config.receive_buf (*C var*), 1035
modem_cmux_dlci_config.receive_buf_size (*C var*), 1035
modem_cmux_dlci_init (*C function*), 1033
modem_cmux_event (*C enum*), 1033
modem_cmux_event.MODEM_CMUX_EVENT_CONNECTED (*C enumerator*), 1033
modem_cmux_event.MODEM_CMUX_EVENT_DISCONNECTED (*C enumerator*), 1033
modem_cmux_init (*C function*), 1033
modem_cmux_release (*C function*), 1034
modem_pipe_api_callback (*C type*), 1028
modem_pipe_attach (*C function*), 1029
modem_pipe_close (*C function*), 1030
modem_pipe_close_async (*C function*), 1031
modem_pipe_event (*C enum*), 1028
modem_pipe_event.MODEM_PIPE_EVENT_CLOSED (*C enumerator*), 1029
modem_pipe_event.MODEM_PIPE_EVENT_OPENED (*C enumerator*), 1028
modem_pipe_event.MODEM_PIPE_EVENT_RECEIVE_READY (*C enumerator*), 1028
modem_pipe_event.MODEM_PIPE_EVENT_TRANSMIT_IDLE (*C enumerator*), 1028
modem_pipe_open (*C function*), 1029
modem_pipe_open_async (*C function*), 1029
modem_pipe_receive (*C function*), 1030
modem_pipe_release (*C function*), 1030
modem_pipe_transmit (*C function*), 1030
modem_pipelink_attach (*C function*), 1037
modem_pipelink_callback (*C type*), 1036
MODEM_PIPELINK_DT_DECLARE (*C macro*), 1036
MODEM_PIPELINK_DT_DEFINE (*C macro*), 1036
MODEM_PIPELINK_DT_GET (*C macro*), 1036
MODEM_PIPELINK_DT_INST_DECLARE (*C macro*), 1036
MODEM_PIPELINK_DT_INST_DEFINE (*C macro*), 1036
MODEM_PIPELINK_DT_INST_GET (*C macro*), 1036
modem_pipelink_event (*C enum*), 1036
modem_pipelink_event.MODEM_PIPELINK_EVENT_CONNECTED (*C enumerator*), 1037

modem_pipelink_event.MODEM_PIPELINK_EVENT_DISCONNECTED (*C enumerator*), 1037
modem_pipelink_get_pipe (*C function*), 1037
modem_pipelink_is_connected (*C function*), 1037
modem_pipelink_release (*C function*), 1037
modem_ppp_attach (*C function*), 1032
MODEM_PPP_DEFINE (*C macro*), 1031
modem_ppp_get_iface (*C function*), 1032
modem_ppp_init_iface (*C type*), 1032
modem_ppp_release (*C function*), 1032
module
    runners.core, 195
mpl_cmd (*C struct*), 1859
mpl_cmd_ntf (*C struct*), 1859
mpl_cmd_ntf.requested_opcode (*C var*), 1859
mpl_cmd_ntf.result_code (*C var*), 1859
mpl_cmd.opcode (*C var*), 1859
mpl_cmd.param (*C var*), 1859
mpl_cmd.use_param (*C var*), 1859
mpl_sci (*C struct*), 1859
mpl_sci.len (*C var*), 1860
mpl_sci.param (*C var*), 1860
mpl_sci.type (*C var*), 1860
mpl_search (*C struct*), 1860
mpl_search.len (*C var*), 1860
mpl_search.search (*C var*), 1860
mpsc (*C struct*), 648
mpsc_init (*C function*), 647
MPSC_INIT (*C macro*), 647
mpsc_node (*C struct*), 647
mpsc_pop (*C function*), 647
mpsc_ptr_get (*C macro*), 647
mpsc_ptr_set (*C macro*), 647
mpsc_ptr_set_get (*C macro*), 647
mpsc_ptr_t (*C type*), 647
mpsc_push (*C function*), 647
mqtt_abort (*C function*), 2860
mqtt_binstr (*C struct*), 2862
mqtt_binstr.data (*C var*), 2862
mqtt_binstr.len (*C var*), 2862
mqtt_client (*C struct*), 2868
mqtt_client_init (*C function*), 2857
mqtt_client.broker (*C var*), 2868
mqtt_client.clean_session (*C var*), 2869
mqtt_client.client_id (*C var*), 2868
mqtt_client.evt_cb (*C var*), 2868
mqtt_client.internal (*C var*), 2868
mqtt_client.keepalive (*C var*), 2869
mqtt_client.password (*C var*), 2868
mqtt_client.protocol_version (*C var*), 2869
mqtt_client.rx_buf (*C var*), 2868
mqtt_client.rx_buf_size (*C var*), 2869
mqtt_client.transport (*C var*), 2868
mqtt_client.tx_buf (*C var*), 2869
mqtt_client.tx_buf_size (*C var*), 2869
mqtt_client.unacked_ping (*C var*), 2869
mqtt_client.user_data (*C var*), 2869
mqtt_client.user_name (*C var*), 2868
mqtt_client.will_message (*C var*), 2868

mqtt_client.will_retain (*C var*), 2869
mqtt_client.will_topic (*C var*), 2868
mqtt_conn_return_code (*C enum*), 2856
mqtt_conn_return_code.MQTT_BAD_USER_NAME_OR_PASSWORD (*C enumerator*), 2856
mqtt_conn_return_code.MQTT_CONNECTION_ACCEPTED (*C enumerator*), 2856
mqtt_conn_return_code.MQTT_IDENTIFIER_REJECTED (*C enumerator*), 2856
mqtt_conn_return_code.MQTT_NOT_AUTHORIZED (*C enumerator*), 2856
mqtt_conn_return_code.MQTT_SERVER_UNAVAILABLE (*C enumerator*), 2856
mqtt_conn_return_code.MQTT_UNACCEPTABLE_PROTOCOL_VERSION (*C enumerator*), 2856
mqtt_connack_param (*C struct*), 2863
mqtt_connack_param.return_code (*C var*), 2863
mqtt_connack_param.session_present_flag (*C var*), 2863
mqtt_connect (*C function*), 2857
mqtt_disconnect (*C function*), 2860
mqtt_evt (*C struct*), 2866
mqtt_evt_cb_t (*C type*), 2854
mqtt_evt_param (*C union*), 2865
mqtt_evt_param.connack (*C var*), 2865
mqtt_evt_param.puback (*C var*), 2865
mqtt_evt_param.pubcomp (*C var*), 2866
mqtt_evt_param.publish (*C var*), 2865
mqtt_evt_param.pubrec (*C var*), 2865
mqtt_evt_param.pubrel (*C var*), 2865
mqtt_evt_param.suback (*C var*), 2866
mqtt_evt_param.unsuback (*C var*), 2866
mqtt_evt_type (*C enum*), 2854
mqtt_evt_type.MQTT_EVT_CONNACK (*C enumerator*), 2854
mqtt_evt_type.MQTT_EVT_DISCONNECT (*C enumerator*), 2854
mqtt_evt_type.MQTT_EVT_PINGRESP (*C enumerator*), 2855
mqtt_evt_type.MQTT_EVT_PUBACK (*C enumerator*), 2855
mqtt_evt_type.MQTT_EVT_PUBCOMP (*C enumerator*), 2855
mqtt_evt_type.MQTT_EVT_PUBLISH (*C enumerator*), 2855
mqtt_evt_type.MQTT_EVT_PUBREC (*C enumerator*), 2855
mqtt_evt_type.MQTT_EVT_PUBREL (*C enumerator*), 2855
mqtt_evt_type.MQTT_EVT_SUBACK (*C enumerator*), 2855
mqtt_evt_type.MQTT_EVT_UNSUBACK (*C enumerator*), 2855
mqtt_evt.param (*C var*), 2866
mqtt_evt.result (*C var*), 2866
mqtt_evt.type (*C var*), 2866
mqtt_input (*C function*), 2860
mqtt_internal (*C struct*), 2867
mqtt_internal.last_activity (*C var*), 2867
mqtt_internal.mutex (*C var*), 2867
mqtt_internal.remaining_payload (*C var*), 2868
mqtt_internal.rx_buf_datalen (*C var*), 2867
mqtt_internal.state (*C var*), 2867
mqtt_keepalive_time_left (*C function*), 2860
mqtt_live (*C function*), 2860
mqtt_ping (*C function*), 2859
mqtt_puback_param (*C struct*), 2863
mqtt_puback_param.message_id (*C var*), 2863
mqtt_pubcomp_param (*C struct*), 2864
mqtt_pubcomp_param.message_id (*C var*), 2864
mqtt_publish (*C function*), 2858
mqtt_publish_message (*C struct*), 2863
mqtt_publish_message.payload (*C var*), 2863
mqtt_publish_message.topic (*C var*), 2863
mqtt_publish_param (*C struct*), 2864

mqtt_publish_param.dup_flag (*C var*), 2864
mqtt_publish_param.message (*C var*), 2864
mqtt_publish_param.message_id (*C var*), 2864
mqtt_publish_param.retain_flag (*C var*), 2865
mqtt_publish_qos1_ack (*C function*), 2858
mqtt_publish_qos2_complete (*C function*), 2859
mqtt_publish_qos2_receive (*C function*), 2858
mqtt_publish_qos2_release (*C function*), 2858
mqtt_pubrec_param (*C struct*), 2863
mqtt_pubrec_param.message_id (*C var*), 2863
mqtt_pubrel_param (*C struct*), 2863
mqtt_pubrel_param.message_id (*C var*), 2864
mqtt_qos (*C enum*), 2855
mqtt_qos.MQTT_QOS_0_AT_MOST_ONCE (*C enumerator*), 2855
mqtt_qos.MQTT_QOS_1_AT_LEAST_ONCE (*C enumerator*), 2856
mqtt_qos.MQTT_QOS_2_EXACTLY_ONCE (*C enumerator*), 2856
mqtt_read_publish_payload (*C function*), 2861
mqtt_read_publish_payload_blocking (*C function*), 2861
mqtt_readall_publish_payload (*C function*), 2861
mqtt_sec_config (*C struct*), 2866
mqtt_sec_config.cert_nocopy (*C var*), 2867
mqtt_sec_config.cipher_count (*C var*), 2866
mqtt_sec_config.cipher_list (*C var*), 2866
mqtt_sec_config.hostname (*C var*), 2867
mqtt_sec_config.peer_verify (*C var*), 2866
mqtt_sec_config.sec_tag_count (*C var*), 2866
mqtt_sec_config.sec_tag_list (*C var*), 2866
mqtt_sn_client (*C struct*), 2877
mqtt_sn_client_deinit (*C function*), 2874
mqtt_sn_client_init (*C function*), 2874
mqtt_sn_client.client_id (*C var*), 2877
mqtt_sn_client.evt_cb (*C var*), 2878
mqtt_sn_client.last_ping (*C var*), 2878
mqtt_sn_client.next_msg_id (*C var*), 2878
mqtt_sn_client.ping_retries (*C var*), 2878
mqtt_sn_client.process_work (*C var*), 2878
mqtt_sn_client.publish (*C var*), 2878
mqtt_sn_client.rx (*C var*), 2878
mqtt_sn_client.state (*C var*), 2878
mqtt_sn_client.topic (*C var*), 2878
mqtt_sn_client.transport (*C var*), 2878
mqtt_sn_client.tx (*C var*), 2878
mqtt_sn_client.will_msg (*C var*), 2878
mqtt_sn_client.will_qos (*C var*), 2878
mqtt_sn_client.will_retain (*C var*), 2878
mqtt_sn_client.will_topic (*C var*), 2877
mqtt_sn_connect (*C function*), 2874
mqtt_sn_data (*C struct*), 2876
MQTT_SN_DATA_BYTES (*C macro*), 2872
MQTT_SN_DATA_STRING_LITERAL (*C macro*), 2871
mqtt_sn_data.data (*C var*), 2876
mqtt_sn_data.size (*C var*), 2876
mqtt_sn_disconnect (*C function*), 2874
mqtt_sn_evt (*C struct*), 2876
mqtt_sn_evt_cb_t (*C type*), 2872
mqtt_sn_evt_param (*C union*), 2876
mqtt_sn_evt_param.data (*C var*), 2876
mqtt_sn_evt_param.publish (*C var*), 2876

mqtt_sn_evt_param.topic_id (*C var*), 2876
mqtt_sn_evt_param.topic_type (*C var*), 2876
mqtt_sn_evt_type (*C enum*), 2873
mqtt_sn_evt_type.MQTT_SN_EVT_ASLEEP (*C enumerator*), 2873
mqtt_sn_evt_type.MQTT_SN_EVT_AWAKE (*C enumerator*), 2873
mqtt_sn_evt_type.MQTT_SN_EVT_CONNECTED (*C enumerator*), 2873
mqtt_sn_evt_type.MQTT_SN_EVT_DISCONNECTED (*C enumerator*), 2873
mqtt_sn_evt_type.MQTT_SN_EVT_PINGRESP (*C enumerator*), 2873
mqtt_sn_evt_type.MQTT_SN_EVT_PUBLISH (*C enumerator*), 2873
mqtt_sn_evt.param (*C var*), 2877
mqtt_sn_evt.type (*C var*), 2877
mqtt_sn_get_topic_name (*C function*), 2876
mqtt_sn_input (*C function*), 2875
mqtt_sn_publish (*C function*), 2875
mqtt_sn_qos (*C enum*), 2872
mqtt_sn_qos.MQTT_SN_QOS_0 (*C enumerator*), 2872
mqtt_sn_qos.MQTT_SN_QOS_1 (*C enumerator*), 2872
mqtt_sn_qos.MQTT_SN_QOS_2 (*C enumerator*), 2872
mqtt_sn_qos.MQTT_SN_QOS_M1 (*C enumerator*), 2872
mqtt_sn_return_code (*C enum*), 2873
mqtt_sn_return_code.MQTT_SN_CODE_ACCEPTED (*C enumerator*), 2873
mqtt_sn_return_code.MQTT_SN_CODE_REJECTED_CONGESTION (*C enumerator*), 2873
mqtt_sn_return_code.MQTT_SN_CODE_REJECTED_NOTSUP (*C enumerator*), 2873
mqtt_sn_return_code.MQTT_SN_CODE_REJECTED_TOPIC_ID (*C enumerator*), 2873
mqtt_sn_sleep (*C function*), 2874
mqtt_sn_subscribe (*C function*), 2875
mqtt_sn_topic_type (*C enum*), 2872
mqtt_sn_topic_type.MQTT_SN_TOPIC_TYPE_NORMAL (*C enumerator*), 2872
mqtt_sn_topic_type.MQTT_SN_TOPIC_TYPE_PREDEF (*C enumerator*), 2873
mqtt_sn_topic_type.MQTT_SN_TOPIC_TYPE_SHORT (*C enumerator*), 2873
mqtt_sn_transport (*C struct*), 2877
mqtt_sn_transport.deinit (*C var*), 2877
mqtt_sn_transport.init (*C var*), 2877
mqtt_sn_transport.msg_send (*C var*), 2877
mqtt_sn_transport.poll (*C var*), 2877
mqtt_sn_transport.recv (*C var*), 2877
mqtt_sn_unsubscribe (*C function*), 2875
mqtt_suback_param (*C struct*), 2864
mqtt_suback_param.message_id (*C var*), 2864
mqtt_suback_param.return_codes (*C var*), 2864
mqtt_suback_return_code (*C enum*), 2856
mqtt_suback_return_code.MQTT_SUBACK_FAILURE (*C enumerator*), 2856
mqtt_suback_return_code.MQTT_SUBACK_SUCCESS_QoS_0 (*C enumerator*), 2856
mqtt_suback_return_code.MQTT_SUBACK_SUCCESS_QoS_1 (*C enumerator*), 2856
mqtt_suback_return_code.MQTT_SUBACK_SUCCESS_QoS_2 (*C enumerator*), 2856
mqtt_subscribe (*C function*), 2859
mqtt_subscription_list (*C struct*), 2865
mqtt_subscription_list.list (*C var*), 2865
mqtt_subscription_list.list_count (*C var*), 2865
mqtt_subscription_list.message_id (*C var*), 2865
mqtt_topic (*C struct*), 2862
mqtt_topic.qos (*C var*), 2862
mqtt_topic.topic (*C var*), 2862
mqtt_transport (*C struct*), 2867
mqtt_transport_type (*C enum*), 2857
mqtt_transport_type.MQTT_TRANSPORT_NON_SECURE (*C enumerator*), 2857
mqtt_transport_type.MQTT_TRANSPORT_NUM (*C enumerator*), 2857
mqtt_transport.sock (*C var*), 2867

mqtt_transport.tcp (*C var*), 2867
mqtt_transport.type (*C var*), 2867
mqtt_unsuback_param (*C struct*), 2864
mqtt_unsuback_param.message_id (*C var*), 2864
mqtt_unsubscribe (*C function*), 2859
mqtt_utf8 (*C struct*), 2862
MQTT_UTF8_LITERAL (*C macro*), 2854
mqtt_utf8.size (*C var*), 2862
mqtt_utf8.utf8 (*C var*), 2862
mqtt_version (*C enum*), 2855
mqtt_version.MQTT_VERSION_3_1_0 (*C enumerator*), 2855
mqtt_version.MQTT_VERSION_3_1_1 (*C enumerator*), 2855
MSEC_PER_SEC (*C macro*), 476
MSG_CTRUNC (*C macro*), 2490
MSG_DONTWAIT (*C macro*), 2490
MSG_PEEK (*C macro*), 2490
MSG_TRUNC (*C macro*), 2490
MSG_WAITALL (*C macro*), 2490
msghdr (*C struct*), 2527
msghdr.msg_control (*C var*), 2528
msghdr.msg_controllen (*C var*), 2528
msghdr.msg_flags (*C var*), 2528
msghdr.msg_iov (*C var*), 2527
msghdr.msg_iovlen (*C var*), 2527
msghdr.msg_name (*C var*), 2527
msghdr.msg_namelen (*C var*), 2527
mspi_api_config (*C type*), 3496
mspi_api_dev_config (*C type*), 3496
mspi_api_get_channel_status (*C type*), 3496
mspi_api_register_callback (*C type*), 3496
mspi_api_scramble_config (*C type*), 3497
mspi_api_timing_config (*C type*), 3497
mspi_api_transceive (*C type*), 3496
mspi_api_xip_config (*C type*), 3497
mspi_bus_event (*C enum*), 3499
mspi_bus_event_cb_mask (*C enum*), 3499
mspi_bus_event_cb_mask.MSPI_BUS_ERROR_CB (*C enumerator*), 3499
mspi_bus_event_cb_mask.MSPI_BUS_NO_CB (*C enumerator*), 3499
mspi_bus_event_cb_mask.MSPI_BUS_RESET_CB (*C enumerator*), 3499
mspi_bus_event_cb_mask.MSPI_BUS_XFER_COMPLETE_CB (*C enumerator*), 3499
mspi_bus_event.MSPI_BUS_ERROR (*C enumerator*), 3499
mspi_bus_event.MSPI_BUS_EVENT_MAX (*C enumerator*), 3499
mspi_bus_event.MSPI_BUS_RESET (*C enumerator*), 3499
mspi_bus_event.MSPI_BUS_XFER_COMPLETE (*C enumerator*), 3499
mspi_ce_polarity (*C enum*), 3499
mspi_ce_polarity.MSPI_CE_ACTIVE_HIGH (*C enumerator*), 3499
mspi_ce_polarity.MSPI_CE_ACTIVE_LOW (*C enumerator*), 3499
mspi_cpp_mode (*C enum*), 3498
mspi_cpp_mode.MSPI_CPP_MODE_0 (*C enumerator*), 3498
mspi_cpp_mode.MSPI_CPP_MODE_1 (*C enumerator*), 3498
mspi_cpp_mode.MSPI_CPP_MODE_2 (*C enumerator*), 3498
mspi_cpp_mode.MSPI_CPP_MODE_3 (*C enumerator*), 3499
mspi_data_rate (*C enum*), 3498
mspi_data_rate.MSPI_DATA_RATE_DUAL (*C enumerator*), 3498
mspi_data_rate.MSPI_DATA_RATE_MAX (*C enumerator*), 3498
mspi_data_rate.MSPI_DATA_RATE_S_D_D (*C enumerator*), 3498
mspi_data_rate.MSPI_DATA_RATE_S_S_D (*C enumerator*), 3498
mspi_data_rate.MSPI_DATA_RATE_SINGLE (*C enumerator*), 3498

mspi_dev_cfg_mask (*C enum*), 3500
mspi_dev_cfg_mask.MSPI_DEVICE_CONFIG_ADDR_LEN (*C enumerator*), 3501
mspi_dev_cfg_mask.MSPI_DEVICE_CONFIG_ALL (*C enumerator*), 3501
mspi_dev_cfg_mask.MSPI_DEVICE_CONFIG_BREAK_TIME (*C enumerator*), 3501
mspi_dev_cfg_mask.MSPI_DEVICE_CONFIG_CE_NUM (*C enumerator*), 3500
mspi_dev_cfg_mask.MSPI_DEVICE_CONFIG_CE_POL (*C enumerator*), 3500
mspi_dev_cfg_mask.MSPI_DEVICE_CONFIG_CMD_LEN (*C enumerator*), 3500
mspi_dev_cfg_mask.MSPI_DEVICE_CONFIG_CPP (*C enumerator*), 3500
mspi_dev_cfg_mask.MSPI_DEVICE_CONFIG_DATA_RATE (*C enumerator*), 3500
mspi_dev_cfg_mask.MSPI_DEVICE_CONFIG_DQS (*C enumerator*), 3500
mspi_dev_cfg_mask.MSPI_DEVICE_CONFIG_ENDIAN (*C enumerator*), 3500
mspi_dev_cfg_mask.MSPI_DEVICE_CONFIG_FREQUENCY (*C enumerator*), 3500
mspi_dev_cfg_mask.MSPI_DEVICE_CONFIG_IO_MODE (*C enumerator*), 3500
mspi_dev_cfg_mask.MSPI_DEVICE_CONFIG_MEM_BOUND (*C enumerator*), 3501
mspi_dev_cfg_mask.MSPI_DEVICE_CONFIG_NONE (*C enumerator*), 3500
mspi_dev_cfg_mask.MSPI_DEVICE_CONFIG_READ_CMD (*C enumerator*), 3500
mspi_dev_cfg_mask.MSPI_DEVICE_CONFIG_RX_DUMMY (*C enumerator*), 3500
mspi_dev_cfg_mask.MSPI_DEVICE_CONFIG_TX_DUMMY (*C enumerator*), 3500
mspi_dev_cfg_mask.MSPI_DEVICE_CONFIG_WRITE_CMD (*C enumerator*), 3500
mspi_driver_api (*C struct*), 3501
mspi_duplex (*C enum*), 3497
mspi_duplex.MSPI_FULL_DUPLEX (*C enumerator*), 3497
mspi_duplex.MSPI_HALF_DUPLEX (*C enumerator*), 3497
mspi_endian (*C enum*), 3499
mspi_endian.MSPI_XFER_BIG_ENDIAN (*C enumerator*), 3499
mspi_endian.MSPI_XFER_LITTLE_ENDIAN (*C enumerator*), 3499
mspi_io_mode (*C enum*), 3497
mspi_io_mode.MSPI_IO_MODE_DUAL (*C enumerator*), 3497
mspi_io_mode.MSPI_IO_MODE_DUAL_1_1_2 (*C enumerator*), 3497
mspi_io_mode.MSPI_IO_MODE_DUAL_1_2_2 (*C enumerator*), 3497
mspi_io_mode.MSPI_IO_MODE_HEX (*C enumerator*), 3498
mspi_io_mode.MSPI_IO_MODE_HEX_8_8_16 (*C enumerator*), 3498
mspi_io_mode.MSPI_IO_MODE_HEX_8_16_16 (*C enumerator*), 3498
mspi_io_mode.MSPI_IO_MODE_MAX (*C enumerator*), 3498
mspi_io_mode.MSPI_IO_MODE_OCTAL (*C enumerator*), 3498
mspi_io_mode.MSPI_IO_MODE_OCTAL_1_1_8 (*C enumerator*), 3498
mspi_io_mode.MSPI_IO_MODE_OCTAL_1_8_8 (*C enumerator*), 3498
mspi_io_mode.MSPI_IO_MODE_QUAD (*C enumerator*), 3497
mspi_io_mode.MSPI_IO_MODE_QUAD_1_1_4 (*C enumerator*), 3498
mspi_io_mode.MSPI_IO_MODE_QUAD_1_4_4 (*C enumerator*), 3498
mspi_io_mode.MSPI_IO_MODE_SINGLE (*C enumerator*), 3497
mspi_op_mode (*C enum*), 3497
mspi_op_mode.MSPI_OP_MODE_CONTROLLER (*C enumerator*), 3497
mspi_op_mode.MSPI_OP_MODE_PERIPHERAL (*C enumerator*), 3497
mspi_xfer_direction (*C enum*), 3500
mspi_xfer_direction.MSPI_RX (*C enumerator*), 3500
mspi_xfer_direction.MSPI_TX (*C enumerator*), 3500
mspi_xfer_mode (*C enum*), 3499
mspi_xfer_mode.MSPI_DMA (*C enumerator*), 3500
mspi_xfer_mode.MSPI_PIO (*C enumerator*), 3500
mspi_xip_permit (*C enum*), 3501
mspi_xip_permit.MSPI_XIP_READ_ONLY (*C enumerator*), 3501
mspi_xip_permit.MSPI_XIP_READ_WRITE (*C enumerator*), 3501

N

name() (*runners.core.ZephyrBinaryRunner class method*), 200
navigation_bearing (*C function*), 3351
navigation_data (*C struct*), 3352

navigation_data.altitude (*C var*), 3352
navigation_data.bearing (*C var*), 3352
navigation_data.latitude (*C var*), 3352
navigation_data.longitude (*C var*), 3352
navigation_data.speed (*C var*), 3352
navigation_distance (*C function*), 3351
net_addr_ntop (*C function*), 2522
net_addr_pton (*C function*), 2522
net_addr_state (*C enum*), 2512
net_addr_state.NET_ADDR_ANY_STATE (*C enumerator*), 2513
net_addr_state.NET_ADDR_DEPRECATED (*C enumerator*), 2513
net_addr_state.NET_ADDR_PREFERRED (*C enumerator*), 2513
net_addr_state.NET_ADDR_TENTATIVE (*C enumerator*), 2513
net_addr_type (*C enum*), 2513
net_addr_type.NET_ADDR_ANY (*C enumerator*), 2513
net_addr_type.NET_ADDR_AUTOCONF (*C enumerator*), 2513
net_addr_type.NET_ADDR_DHCP (*C enumerator*), 2513
net_addr_type.NET_ADDR_MANUAL (*C enumerator*), 2513
net_addr_type.NET_ADDR_OVERRIDABLE (*C enumerator*), 2513
net_buf (*C struct*), 2614
net_buf_add (*C function*), 2600
net_buf_add_be16 (*C function*), 2601
net_buf_add_be24 (*C function*), 2601
net_buf_add_be32 (*C function*), 2601
net_buf_add_be40 (*C function*), 2602
net_buf_add_be48 (*C function*), 2602
net_buf_add_be64 (*C function*), 2602
net_buf_add_le16 (*C function*), 2600
net_buf_add_le24 (*C function*), 2601
net_buf_add_le32 (*C function*), 2601
net_buf_add_le40 (*C function*), 2601
net_buf_add_le48 (*C function*), 2602
net_buf_add_le64 (*C function*), 2602
net_buf_add_mem (*C function*), 2600
net_buf_add_u8 (*C function*), 2600
net_buf_alloc (*C function*), 2596
net_buf_alloc_fixed (*C function*), 2596
net_buf_alloc_len (*C function*), 2597
net_buf_alloc_with_data (*C function*), 2597
net_buf_allocator_cb (*C type*), 2583
net_buf_append_bytes (*C function*), 2612
net_buf_clone (*C function*), 2599
net_buf_data_match (*C function*), 2612
net_buf_destroy (*C function*), 2598
NET_BUF_EXTERNAL_DATA (*C macro*), 2581
net_buf_frag_add (*C function*), 2611
net_buf_frag_del (*C function*), 2611
net_buf_frag_insert (*C function*), 2611
net_buf_frag_last (*C function*), 2611
net_buf_frags_len (*C function*), 2613
net_buf_get (*C function*), 2598
net_buf_headroom (*C function*), 2610
net_buf_id (*C function*), 2595
net_buf_linearize (*C function*), 2612
net_buf_max_len (*C function*), 2610
net_buf_pool (*C struct*), 2615
NET_BUF_POOL_DEFINE (*C macro*), 2582
NET_BUF_POOL_FIXED_DEFINE (*C macro*), 2582

net_buf_pool_get (*C function*), 2595
NET_BUF_POOL_HEAP_DEFINE (*C macro*), 2581
NET_BUF_POOL_VAR_DEFINE (*C macro*), 2582
net_buf_pool.alloc (*C var*), 2615
net_buf_pool.buf_count (*C var*), 2615
net_buf_pool.destroy (*C var*), 2615
net_buf_pool.free (*C var*), 2615
net_buf_pool.lock (*C var*), 2615
net_buf_pool.uninit_count (*C var*), 2615
net_buf_pool.user_data_size (*C var*), 2615
net_buf_pull (*C function*), 2608
net_buf_pull_be16 (*C function*), 2608
net_buf_pull_be24 (*C function*), 2609
net_buf_pull_be32 (*C function*), 2609
net_buf_pull_be40 (*C function*), 2609
net_buf_pull_be48 (*C function*), 2610
net_buf_pull_be64 (*C function*), 2610
net_buf_pull_le16 (*C function*), 2608
net_buf_pull_le24 (*C function*), 2608
net_buf_pull_le32 (*C function*), 2609
net_buf_pull_le40 (*C function*), 2609
net_buf_pull_le48 (*C function*), 2609
net_buf_pull_le64 (*C function*), 2610
net_buf_pull_mem (*C function*), 2608
net_buf_pull_u8 (*C function*), 2608
net_buf_push (*C function*), 2605
net_buf_push_be16 (*C function*), 2606
net_buf_push_be24 (*C function*), 2606
net_buf_push_be32 (*C function*), 2606
net_buf_push_be40 (*C function*), 2607
net_buf_push_be48 (*C function*), 2607
net_buf_push_be64 (*C function*), 2607
net_buf_push_le16 (*C function*), 2606
net_buf_push_le24 (*C function*), 2606
net_buf_push_le32 (*C function*), 2606
net_buf_push_le40 (*C function*), 2607
net_buf_push_le48 (*C function*), 2607
net_buf_push_le64 (*C function*), 2607
net_buf_push_mem (*C function*), 2605
net_buf_push_u8 (*C function*), 2606
net_buf_put (*C function*), 2599
net_buf_ref (*C function*), 2599
net_buf_remove_be16 (*C function*), 2603
net_buf_remove_be24 (*C function*), 2603
net_buf_remove_be32 (*C function*), 2604
net_buf_remove_be40 (*C function*), 2604
net_buf_remove_be48 (*C function*), 2605
net_buf_remove_be64 (*C function*), 2605
net_buf_remove_le16 (*C function*), 2603
net_buf_remove_le24 (*C function*), 2603
net_buf_remove_le32 (*C function*), 2604
net_buf_remove_le40 (*C function*), 2604
net_buf_remove_le48 (*C function*), 2604
net_buf_remove_le64 (*C function*), 2605
net_buf_remove_mem (*C function*), 2602
net_buf_remove_u8 (*C function*), 2603
net_buf_reserve (*C function*), 2600
net_buf_reset (*C function*), 2598

NET_BUF_SIMPLE (*C macro*), 2581
net_buf_simple (*C struct*), 2613
net_buf_simple_add (*C function*), 2584
net_buf_simple_add_be16 (*C function*), 2585
net_buf_simple_add_be24 (*C function*), 2585
net_buf_simple_add_be32 (*C function*), 2585
net_buf_simple_add_be40 (*C function*), 2586
net_buf_simple_add_be48 (*C function*), 2586
net_buf_simple_add_be64 (*C function*), 2586
net_buf_simple_add_le16 (*C function*), 2584
net_buf_simple_add_le24 (*C function*), 2585
net_buf_simple_add_le32 (*C function*), 2585
net_buf_simple_add_le40 (*C function*), 2585
net_buf_simple_add_le48 (*C function*), 2586
net_buf_simple_add_le64 (*C function*), 2586
net_buf_simple_add_mem (*C function*), 2584
net_buf_simple_add_u8 (*C function*), 2584
net_buf_simple_clone (*C function*), 2584
NET_BUF_SIMPLE_DEFINE (*C macro*), 2581
NET_BUF_SIMPLE_DEFINE_STATIC (*C macro*), 2581
net_buf_simple_headroom (*C function*), 2594
net_buf_simple_init (*C function*), 2583
net_buf_simple_init_with_data (*C function*), 2583
net_buf_simple_max_len (*C function*), 2595
net_buf_simple_pull (*C function*), 2592
net_buf_simple_pull_be16 (*C function*), 2592
net_buf_simple_pull_be24 (*C function*), 2593
net_buf_simple_pull_be32 (*C function*), 2593
net_buf_simple_pull_be40 (*C function*), 2593
net_buf_simple_pull_be48 (*C function*), 2594
net_buf_simple_pull_be64 (*C function*), 2594
net_buf_simple_pull_le16 (*C function*), 2592
net_buf_simple_pull_le24 (*C function*), 2592
net_buf_simple_pull_le32 (*C function*), 2593
net_buf_simple_pull_le40 (*C function*), 2593
net_buf_simple_pull_le48 (*C function*), 2594
net_buf_simple_pull_le64 (*C function*), 2594
net_buf_simple_pull_mem (*C function*), 2592
net_buf_simple_pull_u8 (*C function*), 2592
net_buf_simple_push (*C function*), 2589
net_buf_simple_push_be16 (*C function*), 2590
net_buf_simple_push_be24 (*C function*), 2590
net_buf_simple_push_be32 (*C function*), 2590
net_buf_simple_push_be40 (*C function*), 2591
net_buf_simple_push_be48 (*C function*), 2591
net_buf_simple_push_be64 (*C function*), 2591
net_buf_simple_push_le16 (*C function*), 2590
net_buf_simple_push_le24 (*C function*), 2590
net_buf_simple_push_le32 (*C function*), 2590
net_buf_simple_push_le40 (*C function*), 2591
net_buf_simple_push_le48 (*C function*), 2591
net_buf_simple_push_le64 (*C function*), 2591
net_buf_simple_push_mem (*C function*), 2589
net_buf_simple_push_u8 (*C function*), 2590
net_buf_simple_remove_be16 (*C function*), 2587
net_buf_simple_remove_be24 (*C function*), 2587
net_buf_simple_remove_be32 (*C function*), 2588
net_buf_simple_remove_be40 (*C function*), 2588

net_buf_simple_remove_be48 (*C function*), 2589
net_buf_simple_remove_be64 (*C function*), 2589
net_buf_simple_remove_le16 (*C function*), 2587
net_buf_simple_remove_le24 (*C function*), 2587
net_buf_simple_remove_le32 (*C function*), 2588
net_buf_simple_remove_le40 (*C function*), 2588
net_buf_simple_remove_le48 (*C function*), 2588
net_buf_simple_remove_le64 (*C function*), 2589
net_buf_simple_remove_mem (*C function*), 2586
net_buf_simple_remove_u8 (*C function*), 2587
net_buf_simple_reserve (*C function*), 2598
net_buf_simple_reset (*C function*), 2584
net_buf_simple_restore (*C function*), 2595
net_buf_simple_save (*C function*), 2595
net_buf_simple_state (*C struct*), 2613
net_buf_simple_state.len (*C var*), 2614
net_buf_simple_state.offset (*C var*), 2614
net_buf_simple_tail (*C function*), 2594
net_buf_simple_tailroom (*C function*), 2595
net_buf_simple.data (*C var*), 2613
net_buf_simple.len (*C var*), 2613
net_buf_simple.size (*C var*), 2613
net_buf_skip (*C function*), 2613
net_buf_slist_get (*C function*), 2598
net_buf_slist_put (*C function*), 2598
net_buf_tail (*C function*), 2611
net_buf_tailroom (*C function*), 2610
net_buf_unref (*C function*), 2599
net_buf_user_data (*C function*), 2599
net_buf_user_data_copy (*C function*), 2599
net_buf.data (*C var*), 2614
net_buf.flags (*C var*), 2614
net_buf.frags (*C var*), 2614
net_buf.len (*C var*), 2614
net_buf.node (*C var*), 2614
net_buf.pool_id (*C var*), 2614
net_buf.ref (*C var*), 2614
net_buf.size (*C var*), 2614
net_buf.user_data (*C var*), 2615
net_buf.user_data_size (*C var*), 2614
net_bytes_from_str (*C function*), 2524
net_can_ptr (*C function*), 2522
net_capture_cleanup (*C function*), 2578
net_capture_disable (*C function*), 2578
net_capture_enable (*C function*), 2578
net_capture_is_enabled (*C function*), 2578
net_capture_setup (*C function*), 2578
net_config_init (*C function*), 2888
net_config_init_app (*C function*), 2889
net_config_init_by_iface (*C function*), 2888
NET_CONFIG_NEED_IPV4 (*C macro*), 2888
NET_CONFIG_NEED_IPV6 (*C macro*), 2888
NET_CONFIG_NEED_ROUTER (*C macro*), 2888
NET_DEVICE_DT_DEFINE (*C macro*), 2899
NET_DEVICE_DT_DEFINE_INSTANCE (*C macro*), 2900
NET_DEVICE_DT_INST_DEFINE (*C macro*), 2899
NET_DEVICE_DT_INST_DEFINE_INSTANCE (*C macro*), 2901
NET_DEVICE_DT_INST_OFFLOAD_DEFINE (*C macro*), 2901

NET_DEVICE_DT_OFFLOAD_DEFINE (*C macro*), 2901
NET_DEVICE_INIT (*C macro*), 2899
NET_DEVICE_INIT_INSTANCE (*C macro*), 2900
NET_DEVICE_OFFLOAD_INIT (*C macro*), 2901
net_dhcpv4_add_option_callback (*C function*), 2891
net_dhcpv4_add_option_vendor_callback (*C function*), 2892
net_dhcpv4_init_option_callback (*C function*), 2891
net_dhcpv4_init_option_vendor_callback (*C function*), 2891
net_dhcpv4_msg_type (*C enum*), 2890
net_dhcpv4_msg_type_name (*C function*), 2893
net_dhcpv4_msg_type.NET_DHCPV4_MSG_TYPE_ACK (*C enumerator*), 2891
net_dhcpv4_msg_type.NET_DHCPV4_MSG_TYPE_DECLINE (*C enumerator*), 2890
net_dhcpv4_msg_type.NET_DHCPV4_MSG_TYPE_DISCOVER (*C enumerator*), 2890
net_dhcpv4_msg_type.NET_DHCPV4_MSG_TYPE_INFORM (*C enumerator*), 2891
net_dhcpv4_msg_type.NET_DHCPV4_MSG_TYPE_NAK (*C enumerator*), 2891
net_dhcpv4_msg_type.NET_DHCPV4_MSG_TYPE_OFFER (*C enumerator*), 2890
net_dhcpv4_msg_type.NET_DHCPV4_MSG_TYPE_RELEASE (*C enumerator*), 2891
net_dhcpv4_msg_type.NET_DHCPV4_MSG_TYPE_REQUEST (*C enumerator*), 2890
net_dhcpv4_option_callback_handler_t (*C type*), 2890
net_dhcpv4_remove_option_callback (*C function*), 2891
net_dhcpv4_remove_option_vendor_callback (*C function*), 2892
net_dhcpv4_restart (*C function*), 2892
net_dhcpv4_start (*C function*), 2892
net_dhcpv4_stop (*C function*), 2892
net_dhcpv6_params (*C struct*), 2894
net_dhcpv6_params.request_addr (*C var*), 2894
net_dhcpv6_params.request_prefix (*C var*), 2894
net_dhcpv6_restart (*C function*), 2893
net_dhcpv6_start (*C function*), 2893
net_dhcpv6_stop (*C function*), 2893
net_eth_addr (*C struct*), 2649
NET_ETH_ADDR_LEN (*C macro*), 2641
net_eth_addr.addr (*C var*), 2649
net_eth_broadcast_addr (*C function*), 2646
net_eth_carrier_off (*C function*), 2648
net_eth_carrier_on (*C function*), 2648
net_eth_get_hw_capabilities (*C function*), 2646
net_eth_get_hw_config (*C function*), 2646
net_eth_get_ptp_clock (*C function*), 2649
net_eth_get_ptp_clock_by_index (*C function*), 2649
net_eth_get_ptp_port (*C function*), 2649
net_eth_get_vlan_iface (*C function*), 2647
net_eth_get_vlan_main (*C function*), 2647
net_eth_get_vlan_status (*C function*), 2647
net_eth_get_vlan_tag (*C function*), 2647
net_eth_ipv4_mcast_to_mac_addr (*C function*), 2646
net_eth_ipv6_mcast_to_mac_addr (*C function*), 2646
net_eth_is_addr_all_zeroes (*C function*), 2645
net_eth_is_addr_broadcast (*C function*), 2644
net_eth_is_addr_group (*C function*), 2645
net_eth_is_addr_lldp_multicast (*C function*), 2645
net_eth_is_addr_multicast (*C function*), 2645
net_eth_is_addr_ptp_multicast (*C function*), 2645
net_eth_is_addr_unspecified (*C function*), 2645
net_eth_is_addr_valid (*C function*), 2645
net_eth_is_vlan_enabled (*C function*), 2647
net_eth_is_vlan_interface (*C function*), 2648
net_eth_mac_filter (*C function*), 2648

NET_ETH_MINIMAL_FRAME_SIZE (*C macro*), 2641
NET_ETH_MTU (*C macro*), 2641
net_eth_promisc_mode (*C function*), 2648
net_eth_set_ptp_port (*C function*), 2649
net_eth_txinjection_mode (*C function*), 2648
net_eth_type_is_wifi (*C function*), 2649
net_eth_vlan_disable (*C function*), 2646
net_eth_vlan_enable (*C function*), 2646
net_eth_vlan_get_dei (*C function*), 2633
net_eth_vlan_get_pcp (*C function*), 2634
net_eth_vlan_get_vid (*C function*), 2633
net_eth_vlan_set_dei (*C function*), 2634
net_eth_vlan_set_pcp (*C function*), 2634
net_eth_vlan_set_vid (*C function*), 2634
NET_EVENT_CAPTURE_STARTED (*C macro*), 2543
NET_EVENT_CAPTURE_STOPPED (*C macro*), 2543
net_event_coap_observer (*C struct*), 2787
NET_EVENT_COAP_OBSERVER_ADDED (*C macro*), 2787
NET_EVENT_COAP_OBSERVER_REMOVED (*C macro*), 2787
net_event_coap_observer.observer (*C var*), 2788
net_event_coap_observer.resource (*C var*), 2788
net_event_coap_service (*C struct*), 2787
NET_EVENT_COAP_SERVICE_STARTED (*C macro*), 2787
NET_EVENT_COAP_SERVICE_STOPPED (*C macro*), 2787
net_event_coap_service.service (*C var*), 2787
NET_EVENT_CONN_IF_FATAL_ERROR (*C macro*), 2978
NET_EVENT_CONN_IF_TIMEOUT (*C macro*), 2978
NET_EVENT_DNS_SERVER_ADD (*C macro*), 2543
NET_EVENT_DNS_SERVER_DEL (*C macro*), 2543
NET_EVENT_HOSTNAME_CHANGED (*C macro*), 2543
NET_EVENT_IEEE802154_SCAN_RESULT (*C macro*), 2662
NET_EVENT_IF_ADMIN_DOWN (*C macro*), 2540
NET_EVENT_IF_ADMIN_UP (*C macro*), 2540
NET_EVENT_IF_DOWN (*C macro*), 2540
NET_EVENT_IF_UP (*C macro*), 2540
NET_EVENT_IPV4_ACD_CONFLICT (*C macro*), 2542
NET_EVENT_IPV4_ACD_FAILED (*C macro*), 2542
NET_EVENT_IPV4_ACD_SUCCEED (*C macro*), 2542
NET_EVENT_IPV4_ADDR_ADD (*C macro*), 2542
NET_EVENT_IPV4_ADDR_DEL (*C macro*), 2542
NET_EVENT_IPV4_DHCP_BOUND (*C macro*), 2542
NET_EVENT_IPV4_DHCP_START (*C macro*), 2542
NET_EVENT_IPV4_DHCP_STOP (*C macro*), 2542
NET_EVENT_IPV4_MADDR_ADD (*C macro*), 2542
NET_EVENT_IPV4_MADDR_DEL (*C macro*), 2542
NET_EVENT_IPV4_MCAST_JOIN (*C macro*), 2542
NET_EVENT_IPV4_MCAST_LEAVE (*C macro*), 2542
NET_EVENT_IPV4_ROUTER_ADD (*C macro*), 2542
NET_EVENT_IPV4_ROUTER_DEL (*C macro*), 2542
net_event_ipv6_addr (*C struct*), 2547
NET_EVENT_IPV6_ADDR_ADD (*C macro*), 2540
NET_EVENT_IPV6_ADDR_DEL (*C macro*), 2540
NET_EVENT_IPV6_ADDR_DEPRECATED (*C macro*), 2541
net_event_ipv6_addr.addr (*C var*), 2547
NET_EVENT_IPV6_DAD_FAILED (*C macro*), 2541
NET_EVENT_IPV6_DAD_SUCCEED (*C macro*), 2541
NET_EVENT_IPV6_DHCP_BOUND (*C macro*), 2541
NET_EVENT_IPV6_DHCP_START (*C macro*), 2541

NET_EVENT_IPV6_DHCP_STOP (*C macro*), 2541
NET_EVENT_IPV6_MADDR_ADD (*C macro*), 2540
NET_EVENT_IPV6_MADDR_DEL (*C macro*), 2540
NET_EVENT_IPV6_MCAST_JOIN (*C macro*), 2540
NET_EVENT_IPV6_MCAST_LEAVE (*C macro*), 2541
net_event_ipv6_nbr (*C struct*), 2547
NET_EVENT_IPV6_NBR_ADD (*C macro*), 2541
NET_EVENT_IPV6_NBR_DEL (*C macro*), 2541
net_event_ipv6_nbr.addr (*C var*), 2547
net_event_ipv6_nbr.idx (*C var*), 2547
NET_EVENT_IPV6_PE_DISABLED (*C macro*), 2541
NET_EVENT_IPV6_PE_ENABLED (*C macro*), 2541
net_event_ipv6_pe_filter (*C struct*), 2548
NET_EVENT_IPV6_PE_FILTER_ADD (*C macro*), 2541
NET_EVENT_IPV6_PE_FILTER_DEL (*C macro*), 2542
net_event_ipv6_pe_filter.is_deny_list (*C var*), 2548
net_event_ipv6_pe_filter.prefix (*C var*), 2548
net_event_ipv6_prefix (*C struct*), 2548
NET_EVENT_IPV6_PREFIX_ADD (*C macro*), 2540
NET_EVENT_IPV6_PREFIX_DEL (*C macro*), 2540
net_event_ipv6_prefix.addr (*C var*), 2548
net_event_ipv6_prefix.len (*C var*), 2548
net_event_ipv6_prefix.lifetime (*C var*), 2548
net_event_ipv6_route (*C struct*), 2547
NET_EVENT_IPV6_ROUTE_ADD (*C macro*), 2541
NET_EVENT_IPV6_ROUTE_DEL (*C macro*), 2541
net_event_ipv6_route.addr (*C var*), 2547
net_event_ipv6_route.nexthop (*C var*), 2547
net_event_ipv6_route.prefix_len (*C var*), 2548
NET_EVENT_IPV6_ROUTER_ADD (*C macro*), 2541
NET_EVENT_IPV6_ROUTER_DEL (*C macro*), 2541
NET_EVENT_L4_CONNECTED (*C macro*), 2542
NET_EVENT_L4_DISCONNECTED (*C macro*), 2543
net_event_l4_hostname (*C struct*), 2548
net_event_l4_hostname.hostname (*C var*), 2548
NET_EVENT_L4_IPV4_CONNECTED (*C macro*), 2543
NET_EVENT_L4_IPV4_DISCONNECTED (*C macro*), 2543
NET_EVENT_L4_IPV6_CONNECTED (*C macro*), 2543
NET_EVENT_L4_IPV6_DISCONNECTED (*C macro*), 2543
NET_EVENT_WIFI_AP_DISABLE_RESULT (*C macro*), 2716
NET_EVENT_WIFI_AP_ENABLE_RESULT (*C macro*), 2716
NET_EVENT_WIFI_AP_STA_CONNECTED (*C macro*), 2717
NET_EVENT_WIFI_AP_STA_DISCONNECTED (*C macro*), 2717
net_event_wifi_cmd (*C enum*), 2727
net_event_wifi_cmd.NET_EVENT_WIFI_CMD_AP_DISABLE_RESULT (*C enumerator*), 2727
net_event_wifi_cmd.NET_EVENT_WIFI_CMD_AP_ENABLE_RESULT (*C enumerator*), 2727
net_event_wifi_cmd.NET_EVENT_WIFI_CMD_AP_STA_CONNECTED (*C enumerator*), 2727
net_event_wifi_cmd.NET_EVENT_WIFI_CMD_AP_STA_DISCONNECTED (*C enumerator*), 2727
net_event_wifi_cmd.NET_EVENT_WIFI_CMD_CONNECT_RESULT (*C enumerator*), 2727
net_event_wifi_cmd.NET_EVENT_WIFI_CMD_DISCONNECT_COMPLETE (*C enumerator*), 2727
net_event_wifi_cmd.NET_EVENT_WIFI_CMD_DISCONNECT_RESULT (*C enumerator*), 2727
net_event_wifi_cmd.NET_EVENT_WIFI_CMD_IFACE_STATUS (*C enumerator*), 2727
net_event_wifi_cmd.NET_EVENT_WIFI_CMD_RAW_SCAN_RESULT (*C enumerator*), 2727
net_event_wifi_cmd.NET_EVENT_WIFI_CMD_SCAN_DONE (*C enumerator*), 2727
net_event_wifi_cmd.NET_EVENT_WIFI_CMD_SCAN_RESULT (*C enumerator*), 2727
net_event_wifi_cmd.NET_EVENT_WIFI_CMD_TWT (*C enumerator*), 2727
net_event_wifi_cmd.NET_EVENT_WIFI_CMD_TWT_SLEEP_STATE (*C enumerator*), 2727
NET_EVENT_WIFI_CONNECT_RESULT (*C macro*), 2716

NET_EVENT_WIFI_DISCONNECT_COMPLETE (*C macro*), 2716
NET_EVENT_WIFI_DISCONNECT_RESULT (*C macro*), 2716
NET_EVENT_WIFI_IFACE_STATUS (*C macro*), 2716
NET_EVENT_WIFI_RAW_SCAN_RESULT (*C macro*), 2716
NET_EVENT_WIFI_SCAN_DONE (*C macro*), 2716
NET_EVENT_WIFI_SCAN_RESULT (*C macro*), 2716
NET_EVENT_WIFI_TWT (*C macro*), 2716
NET_EVENT_WIFI_TWT_SLEEP_STATE (*C macro*), 2716
net_family2str (*C function*), 2524
net_hostname_get (*C function*), 2895
net_hostname_init (*C function*), 2895
NET_HOSTNAME_MAX_LEN (*C macro*), 2895
net_hostname_set (*C function*), 2895
net_hostname_set_postfix (*C function*), 2895
net_if (*C struct*), 2936
net_if_addr (*C struct*), 2932
net_if_addr_ipv4 (*C struct*), 2934
net_if_addr_ipv4.ipv4 (*C var*), 2935
net_if_addr_ipv4.netmask (*C var*), 2935
net_if_addr_set_lf (*C function*), 2909
net_if_addr.addr_state (*C var*), 2932
net_if_addr.addr_type (*C var*), 2932
net_if_addr.address (*C var*), 2932
net_if_addr.atomic_ref (*C var*), 2932
net_if_addr.is_infinite (*C var*), 2932
net_if_addr.is_mesh_local (*C var*), 2932
net_if_addr.is_temporary (*C var*), 2932
net_if_addr.is_used (*C var*), 2932
net_if_are_pending_tx_packets (*C function*), 2930
net_if_call_link_cb (*C function*), 2927
net_if_carrier_off (*C function*), 2929
net_if_carrier_on (*C function*), 2929
net_if_cb_t (*C type*), 2903
net_if_checksum_type (*C enum*), 2904
net_if_checksum_type.NET_IF_CHECKSUM_IPV4_HEADER (*C enumerator*), 2905
net_if_checksum_type.NET_IF_CHECKSUM_IPV4_ICMP (*C enumerator*), 2905
net_if_checksum_type.NET_IF_CHECKSUM_IPV4_TCP (*C enumerator*), 2905
net_if_checksum_type.NET_IF_CHECKSUM_IPV4_UDP (*C enumerator*), 2905
net_if_checksum_type.NET_IF_CHECKSUM_IPV6_HEADER (*C enumerator*), 2905
net_if_checksum_type.NET_IF_CHECKSUM_IPV6_ICMP (*C enumerator*), 2905
net_if_checksum_type.NET_IF_CHECKSUM_IPV6_TCP (*C enumerator*), 2905
net_if_checksum_type.NET_IF_CHECKSUM_IPV6_UDP (*C enumerator*), 2905
net_if_config (*C struct*), 2935
net_if_config_get (*C function*), 2910
net_if_config_ipv4_get (*C function*), 2920
net_if_config_ipv4_put (*C function*), 2920
net_if_config_ipv6_get (*C function*), 2911
net_if_config_ipv6_put (*C function*), 2911
net_if_dev (*C struct*), 2936
net_if_dev.dev (*C var*), 2936
net_if_dev.flags (*C var*), 2936
net_if_dev.l2 (*C var*), 2936
net_if_dev.l2_data (*C var*), 2936
net_if_dev.link_addr (*C var*), 2936
net_if_dev.mtu (*C var*), 2936
net_if_dev.oper_state (*C var*), 2936
net_if_dormant_off (*C function*), 2930
net_if_dormant_on (*C function*), 2929

net_if_down (*C function*), 2929
net_if_flag (*C enum*), 2903
net_if_flag_clear (*C function*), 2905
net_if_flag_is_set (*C function*), 2906
net_if_flag_set (*C function*), 2905
net_if_flag_test_and_clear (*C function*), 2906
net_if_flag_test_and_set (*C function*), 2905
net_if_flag.NET_IF_DORMANT (*C enumerator*), 2904
net_if_flag.NET_IF_FORWARD_MULTICASTS (*C enumerator*), 2903
net_if_flag.NET_IF_IPV4 (*C enumerator*), 2903
net_if_flag.NET_IF_IPV6 (*C enumerator*), 2904
net_if_flag.NET_IF_IPV6_NO_MLD (*C enumerator*), 2904
net_if_flag.NET_IF_IPV6_NO_ND (*C enumerator*), 2904
net_if_flag.NET_IF_LOWER_UP (*C enumerator*), 2904
net_if_flag.NET_IF_NO_AUTO_START (*C enumerator*), 2903
net_if_flag.NET_IF_NO_TX_LOCK (*C enumerator*), 2904
net_if_flag.NET_IF_POINTOPOINT (*C enumerator*), 2903
net_if_flag.NET_IF_PROMISC (*C enumerator*), 2903
net_if_flag.NET_IF_RUNNING (*C enumerator*), 2904
net_if_flag.NET_IF_SUSPENDED (*C enumerator*), 2903
net_if_flag.NET_IF_UP (*C enumerator*), 2903
net_if_foreach (*C function*), 2928
net_if_get_by_iface (*C function*), 2928
net_if_get_by_index (*C function*), 2928
net_if_get_by_link_addr (*C function*), 2910
net_if_get_by_name (*C function*), 2931
net_if_get_config (*C function*), 2908
net_if_get_default (*C function*), 2910
net_if_get_device (*C function*), 2907
net_if_get_first_by_type (*C function*), 2910
net_if_get_first_up (*C function*), 2910
net_if_get_first_wifi (*C function*), 2931
net_if_get_link_addr (*C function*), 2908
net_if_get_mtu (*C function*), 2909
net_if_get_name (*C function*), 2931
net_if_get_wifi_sap (*C function*), 2931
net_if_get_wifi_sta (*C function*), 2931
net_if_ip (*C struct*), 2935
net_if_ip_addr_cb_t (*C type*), 2902
net_if_ip_maddr_cb_t (*C type*), 2902
net_if_ipv4 (*C struct*), 2935
net_if_ipv4_addr_add (*C function*), 2921
net_if_ipv4_addr_add_by_index (*C function*), 2922
net_if_ipv4_addr_foreach (*C function*), 2922
net_if_ipv4_addr_lookup (*C function*), 2517, 2921
net_if_ipv4_addr_lookup_by_index (*C function*), 2921
net_if_ipv4_addr_mask_cmp (*C function*), 2517, 2924
net_if_ipv4_addr_rm (*C function*), 2921
net_if_ipv4_addr_rm_by_index (*C function*), 2922
net_if_ipv4_get_global_addr (*C function*), 2925
net_if_ipv4_get_ll (*C function*), 2925
net_if_ipv4_get_mcast_ttl (*C function*), 2920
net_if_ipv4_get_netmask (*C function*), 2926
net_if_ipv4_get_netmask_by_addr (*C function*), 2925
net_if_ipv4_get_ttl (*C function*), 2920
net_if_ipv4_is_addr_bcast (*C function*), 2517, 2924
net_if_ipv4_maddr_add (*C function*), 2922
net_if_ipv4_maddr_foreach (*C function*), 2922

net_if_ipv4_maddr_is_joined (*C function*), 2923
net_if_ipv4_maddr_join (*C function*), 2923
net_if_ipv4_maddr_leave (*C function*), 2923
net_if_ipv4_maddr_lookup (*C function*), 2923
net_if_ipv4_maddr_rm (*C function*), 2922
net_if_ipv4_router_add (*C function*), 2924
net_if_ipv4_router_find_default (*C function*), 2924
net_if_ipv4_router_lookup (*C function*), 2923
net_if_ipv4_router_rm (*C function*), 2924
net_if_ipv4_select_src_addr (*C function*), 2925
net_if_ipv4_select_src_iface (*C function*), 2925
net_if_ipv4_set_gw (*C function*), 2927
net_if_ipv4_set_gw_by_index (*C function*), 2927
net_if_ipv4_set_mcast_ttl (*C function*), 2921
net_if_ipv4_set_netmask (*C function*), 2926
net_if_ipv4_set_netmask_by_addr (*C function*), 2926
net_if_ipv4_set_netmask_by_addr_by_index (*C function*), 2926
net_if_ipv4_set_netmask_by_index (*C function*), 2926
net_if_ipv4_set_ttl (*C function*), 2920
net_if_ipv4.gw (*C var*), 2935
net_if_ipv4.mcast (*C var*), 2935
net_if_ipv4.mcast_ttl (*C var*), 2935
net_if_ipv4.ttl (*C var*), 2935
net_if_ipv4.unicast (*C var*), 2935
net_if_ipv6 (*C struct*), 2934
net_if_ipv6_addr_add (*C function*), 2911
net_if_ipv6_addr_add_by_index (*C function*), 2912
net_if_ipv6_addr_foreach (*C function*), 2912
net_if_ipv6_addr_lookup (*C function*), 2514, 2911
net_if_ipv6_addr_lookup_by_iface (*C function*), 2911
net_if_ipv6_addr_lookup_by_index (*C function*), 2911
net_if_ipv6_addr_onlink (*C function*), 2916
net_if_ipv6_addr_rm (*C function*), 2912
net_if_ipv6_addr_rm_by_index (*C function*), 2912
net_if_ipv6_addr_update_lifetime (*C function*), 2912
net_if_ipv6_calc_reachable_time (*C function*), 2918
net_if_ipv6_dad_failed (*C function*), 2919
net_if_ipv6_get_global_addr (*C function*), 2920
net_if_ipv6_get_hop_limit (*C function*), 2917
net_if_ipv6_get_ll (*C function*), 2919
net_if_ipv6_get_ll_addr (*C function*), 2919
net_if_ipv6_get_mcast_hop_limit (*C function*), 2917
net_if_ipv6_get_reachable_time (*C function*), 2918
net_if_ipv6_get_retrans_timer (*C function*), 2918
net_if_ipv6_maddr_add (*C function*), 2913
net_if_ipv6_maddr_foreach (*C function*), 2913
net_if_ipv6_maddr_is_joined (*C function*), 2914
net_if_ipv6_maddr_join (*C function*), 2914
net_if_ipv6_maddr_leave (*C function*), 2914
net_if_ipv6_maddr_lookup (*C function*), 2514, 2913
net_if_ipv6_maddr_rm (*C function*), 2913
net_if_ipv6_prefix (*C struct*), 2933
net_if_ipv6_prefix_add (*C function*), 2915
net_if_ipv6_prefix_get (*C function*), 2914
net_if_ipv6_prefix_lookup (*C function*), 2915
net_if_ipv6_prefix_rm (*C function*), 2915
net_if_ipv6_prefix_set_lf (*C function*), 2915
net_if_ipv6_prefix_set_timer (*C function*), 2915

net_if_ipv6_prefix_unset_timer (*C function*), 2915
net_if_ipv6_prefix.iface (*C var*), 2933
net_if_ipv6_prefix.is_infinite (*C var*), 2933
net_if_ipv6_prefix.is_used (*C var*), 2933
net_if_ipv6_prefix.len (*C var*), 2933
net_if_ipv6_prefix.lifetime (*C var*), 2933
net_if_ipv6_prefix.prefix (*C var*), 2933
net_if_ipv6_router_add (*C function*), 2916
net_if_ipv6_router_find_default (*C function*), 2916
net_if_ipv6_router_lookup (*C function*), 2916
net_if_ipv6_router_rm (*C function*), 2917
net_if_ipv6_router_update_lifetime (*C function*), 2916
net_if_ipv6_select_src_addr (*C function*), 2918
net_if_ipv6_select_src_addr_hint (*C function*), 2919
net_if_ipv6_select_src_iface (*C function*), 2919
net_if_ipv6_set_base_reachable_time (*C function*), 2917
net_if_ipv6_set_hop_limit (*C function*), 2917
net_if_ipv6_set_mcast_hop_limit (*C function*), 2917
net_if_ipv6_set_reachable_time (*C function*), 2918
net_if_ipv6_set_retrans_timer (*C function*), 2918
net_if_ipv6.base_reachable_time (*C var*), 2934
net_if_ipv6.hop_limit (*C var*), 2934
net_if_ipv6.mcast (*C var*), 2934
net_if_ipv6.mcast_hop_limit (*C var*), 2934
net_if_ipv6.prefix (*C var*), 2934
net_if_ipv6.reachable_time (*C var*), 2934
net_if_ipv6.retrans_timer (*C var*), 2934
net_if_ipv6.unicast (*C var*), 2934
net_if_is_admin_up (*C function*), 2929
net_if_is_carrier_ok (*C function*), 2929
net_if_is_dormant (*C function*), 2930
net_if_is_ip_offloaded (*C function*), 2907
net_if_is_offloaded (*C function*), 2907
net_if_is_promisc (*C function*), 2930
net_if_is_socket_offloaded (*C function*), 2908
net_if_is_up (*C function*), 2929
net_if_is_wifi (*C function*), 2931
net_if_l2 (*C function*), 2906
net_if_l2_data (*C function*), 2907
net_if_link_callback_t (*C type*), 2903
net_if_link_cb (*C struct*), 2937
net_if_link_cb.cb (*C var*), 2938
net_if_link_cb.node (*C var*), 2938
net_if_lookup_by_dev (*C function*), 2910
net_if_mcast_addr (*C struct*), 2932
net_if_mcast_addr.address (*C var*), 2933
net_if_mcast_addr.is_joined (*C var*), 2933
net_if_mcast_addr.is_used (*C var*), 2933
net_if_mcast_callback_t (*C type*), 2902
net_if_mcast_mon_register (*C function*), 2913
net_if_mcast_mon_unregister (*C function*), 2914
net_if_mcast_monitor (*C function*), 2914
net_if_mcast_monitor (*C struct*), 2937
net_if_mcast_monitor.cb (*C var*), 2937
net_if_mcast_monitor.iface (*C var*), 2937
net_if_mcast_monitor.node (*C var*), 2937
net_if_nbr_reachability_hint (*C function*), 2909
net_if_need_calc_rx_checksum (*C function*), 2928

net_if_need_calc_tx_checksum (*C function*), 2928
net_if_offload (*C function*), 2907
net_if_oper_state (*C enum*), 2904
net_if_oper_state (*C function*), 2906
net_if_oper_state_set (*C function*), 2906
net_if_oper_state.NET_IF_OPER_DORMANT (*C enumerator*), 2904
net_if_oper_state.NET_IF_OPER_DOWN (*C enumerator*), 2904
net_if_oper_state.NET_IF_OPER_LOWERLAYERDOWN (*C enumerator*), 2904
net_if_oper_state.NET_IF_OPER_NOTPRESENT (*C enumerator*), 2904
net_if_oper_state.NET_IF_OPER_TESTING (*C enumerator*), 2904
net_if_oper_state.NET_IF_OPER_UNKNOWN (*C enumerator*), 2904
net_if_oper_state.NET_IF_OPER_UP (*C enumerator*), 2904
net_if_queue_tx (*C function*), 2907
net_if_recv_data (*C function*), 2907
net_if_register_link_cb (*C function*), 2927
net_if_router (*C struct*), 2933
net_if_router_ipv4 (*C function*), 2923
net_if_router_ipv6 (*C function*), 2916
net_if_router_rm (*C function*), 2910
net_if_router.address (*C var*), 2933
net_if_router.iface (*C var*), 2933
net_if_router.is_default (*C var*), 2934
net_if_router.is_infinite (*C var*), 2934
net_if_router.is_used (*C var*), 2934
net_if_router.life_start (*C var*), 2934
net_if_router.lifetime (*C var*), 2934
net_if_router.node (*C var*), 2933
net_if_select_src_iface (*C function*), 2927
net_if_send_data (*C function*), 2906
net_if_set_default (*C function*), 2910
net_if_set_link_addr (*C function*), 2909
net_if_set_mtu (*C function*), 2909
net_if_set_name (*C function*), 2931
net_if_set_promisc (*C function*), 2930
net_if_socket_offload (*C function*), 2908
net_if_socket_offload_set (*C function*), 2908
net_if_start_dad (*C function*), 2908
net_if_start_rs (*C function*), 2908
net_if_stop_rs (*C function*), 2909
net_if_unregister_link_cb (*C function*), 2927
net_if_unset_promisc (*C function*), 2930
net_if_up (*C function*), 2928
NET_IFACE_COUNT (*C macro*), 2902
net_if.config (*C var*), 2937
net_if.if_dev (*C var*), 2937
net_if.lock (*C var*), 2937
net_if.pe_enabled (*C var*), 2937
net_if.pe_prefer_public (*C var*), 2937
net_if.tx_lock (*C var*), 2937
net_ip_mtu (*C enum*), 2512
net_ip_mtu.NET_IPV4_MTU (*C enumerator*), 2512
net_ip_mtu.NET_IPV6_MTU (*C enumerator*), 2512
net_ip_protocol (*C enum*), 2510
net_ip_protocol_secure (*C enum*), 2511
net_ip_protocol_secure.IPPROTO_DTLS_1_0 (*C enumerator*), 2511
net_ip_protocol_secure.IPPROTO_DTLS_1_2 (*C enumerator*), 2511
net_ip_protocol_secure.IPPROTO_TLS_1_0 (*C enumerator*), 2511
net_ip_protocol_secure.IPPROTO_TLS_1_1 (*C enumerator*), 2511

net_ip_protocol_secure.IPPROTO_TLS_1_2 (*C enumerator*), 2511
net_ip_protocol.IPPROTO_ICMP (*C enumerator*), 2510
net_ip_protocol.IPPROTO_ICMPV6 (*C enumerator*), 2511
net_ip_protocol.IPPROTO_IGMP (*C enumerator*), 2511
net_ip_protocol.IPPROTO_IP (*C enumerator*), 2510
net_ip_protocol.IPPROTO_IPIP (*C enumerator*), 2511
net_ip_protocol.IPPROTO_IPV6 (*C enumerator*), 2511
net_ip_protocol.IPPROTO_RAW (*C enumerator*), 2511
net_ip_protocol.IPPROTO_TCP (*C enumerator*), 2511
net_ip_protocol.IPPROTO_UDP (*C enumerator*), 2511
net_ipaddr_copy (*C macro*), 2510
net_ipaddr_parse (*C function*), 2523
net_ipv4_addr_cmp (*C function*), 2515
net_ipv4_addr_cmp_raw (*C function*), 2515
net_ipv4_addr_copy_raw (*C function*), 2515
net_ipv4_addr_mask_cmp (*C function*), 2517
NET_IPV4_ADDR_SIZE (*C macro*), 2509
net_ipv4_broadcast_address (*C function*), 2517
net_ipv4_is_addr_bcast (*C function*), 2517
net_ipv4_is_addr_loopback (*C function*), 2514
net_ipv4_is_addr_mcast (*C function*), 2515
net_ipv4_is_addr_unspecified (*C function*), 2514
net_ipv4_is_ll_addr (*C function*), 2515
net_ipv4_is_my_addr (*C function*), 2517
net_ipv4_is_private_addr (*C function*), 2515
net_ipv4_unspecified_address (*C function*), 2517
net_ipv6_addr_based_on_ll (*C function*), 2521
net_ipv6_addr_cmp (*C function*), 2516
net_ipv6_addr_cmp_raw (*C function*), 2516
net_ipv6_addr_copy_raw (*C function*), 2515
net_ipv6_addr_create (*C function*), 2520
net_ipv6_addr_create_iid (*C function*), 2521
net_ipv6_addr_create_ll_allnodes_mcast (*C function*), 2520
net_ipv6_addr_create_ll_allrouters_mcast (*C function*), 2520
net_ipv6_addr_create_solicited_node (*C function*), 2520
net_ipv6_addr_create_v4_mapped (*C function*), 2521
net_ipv6_addr_is_v4_mapped (*C function*), 2521
NET_IPV6_ADDR_SIZE (*C macro*), 2509
net_ipv6_is_addr_loopback (*C function*), 2513
net_ipv6_is_addr_mcast (*C function*), 2513
net_ipv6_is_addr_mcast_all_nodes_group (*C function*), 2519
net_ipv6_is_addr_mcast_global (*C function*), 2518
net_ipv6_is_addr_mcast_group (*C function*), 2519
net_ipv6_is_addr_mcast_iface (*C function*), 2518
net_ipv6_is_addr_mcast_iface_all_nodes (*C function*), 2519
net_ipv6_is_addr_mcast_link (*C function*), 2518
net_ipv6_is_addr_mcast_link_all_nodes (*C function*), 2520
net_ipv6_is_addr_mcast_mesh (*C function*), 2519
net_ipv6_is_addr_mcast_org (*C function*), 2519
net_ipv6_is_addr_mcast_scope (*C function*), 2518
net_ipv6_is_addr_mcast_site (*C function*), 2519
net_ipv6_is_addr_solicited_node (*C function*), 2518
net_ipv6_is_addr_unspecified (*C function*), 2518
net_ipv6_is_global_addr (*C function*), 2516
net_ipv6_is_ll_addr (*C function*), 2516
net_ipv6_is_my_addr (*C function*), 2514
net_ipv6_is_my_maddr (*C function*), 2514
net_ipv6_is_prefix (*C function*), 2514

net_ipv6_is_private_addr (*C function*), 2516
net_ipv6_is_same_mcast_scope (*C function*), 2518
net_ipv6_is_sl_addr (*C function*), 2516
net_ipv6_is_ula_addr (*C function*), 2516
net_ipv6_pe_add_filter (*C function*), 2525
net_ipv6_pe_del_filter (*C function*), 2525
net_ipv6_unspecified_address (*C function*), 2517
net_l2 (*C struct*), 2941
net_l2_flags (*C enum*), 2940
net_l2_flags.NET_L2_MULTICAST (*C enumerator*), 2940
net_l2_flags.NET_L2_MULTICAST_SKIP_JOIN_SOLICIT_NODE (*C enumerator*), 2940
net_l2_flags.NET_L2_POINT_TO_POINT (*C enumerator*), 2941
net_l2_flags.NET_L2_PROMISC_MODE (*C enumerator*), 2941
net_l2.enable (*C var*), 2941
net_l2.get_flags (*C var*), 2941
net_l2.recv (*C var*), 2941
net_l2.send (*C var*), 2941
NET_LINK_ADDR_MAX_LENGTH (*C macro*), 2942
net_link_type (*C enum*), 2942
net_link_type.NET_LINK_BLUETOOTH (*C enumerator*), 2943
net_link_type.NET_LINK_CANBUS_RAW (*C enumerator*), 2943
net_link_type.NET_LINK_DUMMY (*C enumerator*), 2943
net_link_type.NET_LINK_ETHERNET (*C enumerator*), 2943
net_link_type.NET_LINK_IEEE802154 (*C enumerator*), 2943
net_link_type.NET_LINK_UNKNOWN (*C enumerator*), 2943
net_linkaddr (*C struct*), 2943
net_linkaddr_cmp (*C function*), 2943
net_linkaddr_set (*C function*), 2943
net_linkaddr_storage (*C struct*), 2944
net_linkaddr_storage.addr (*C var*), 2944
net_linkaddr_storage.len (*C var*), 2944
net_linkaddr_storage.type (*C var*), 2944
net_linkaddr.addr (*C var*), 2944
net_linkaddr.len (*C var*), 2944
net_linkaddr.type (*C var*), 2944
net_lldp_chassis_tlv (*C struct*), 2637
net_lldp_chassis_tlv.subtype (*C var*), 2637
net_lldp_chassis_tlv.type_length (*C var*), 2637
net_lldp_chassis_tlv.value (*C var*), 2637
net_lldp_config (*C function*), 2636
net_lldp_config_optional (*C function*), 2636
net_lldp_init (*C function*), 2637
net_lldp_port_tlv (*C struct*), 2637
net_lldp_port_tlv.subtype (*C var*), 2638
net_lldp_port_tlv.type_length (*C var*), 2638
net_lldp_port_tlv.value (*C var*), 2638
net_lldp_recv (*C function*), 2637
net_lldp_recv_cb_t (*C type*), 2635
net_lldp_register_callback (*C function*), 2637
net_lldp_set_lldpdu (*C macro*), 2635
net_lldp_time_to_live_tlv (*C struct*), 2638
net_lldp_time_to_live_tlv.ttl (*C var*), 2638
net_lldp_time_to_live_tlv.type_length (*C var*), 2638
net_lldp_tlv_type (*C enum*), 2635
net_lldp_tlv_type.LLDP_TLV_CHASSIS_ID (*C enumerator*), 2636
net_lldp_tlv_type.LLDP_TLV_END_LLDPDU (*C enumerator*), 2636
net_lldp_tlv_type.LLDP_TLV_MANAGEMENT_ADDR (*C enumerator*), 2636
net_lldp_tlv_type.LLDP_TLV_ORG_SPECIFIC (*C enumerator*), 2636

net_lldp_tlv_type.LLDP_TLV_PORT_DESC (*C enumerator*), 2636
net_lldp_tlv_type.LLDP_TLV_PORT_ID (*C enumerator*), 2636
net_lldp_tlv_type.LLDP_TLV_SYSTEM_CAPABILITIES (*C enumerator*), 2636
net_lldp_tlv_type.LLDP_TLV_SYSTEM_DESC (*C enumerator*), 2636
net_lldp_tlv_type.LLDP_TLV_SYSTEM_NAME (*C enumerator*), 2636
net_lldp_tlv_type.LLDP_TLV_TTL (*C enumerator*), 2636
net_lldp_unset_lldpdu (*C macro*), 2635
net_lldpdu (*C struct*), 2638
net_lldpdu.chassis_id (*C var*), 2638
net_lldpdu.port_id (*C var*), 2638
net_lldpdu.ttl (*C var*), 2638
NET_MAX_PRIORITIES (*C macro*), 2510
net_mgmt (*C macro*), 2543
net_mgmt_add_event_callback (*C function*), 2545
NET_MGMT_DEFINE_REQUEST_HANDLER (*C macro*), 2543
net_mgmt_del_event_callback (*C function*), 2545
net_mgmt_event_callback (*C struct*), 2549
net_mgmt_event_callback.event_mask (*C var*), 2549
net_mgmt_event_callback.handler (*C var*), 2549
net_mgmt_event_callback.node (*C var*), 2549
net_mgmt_event_callback.raised_event (*C var*), 2549
net_mgmt_event_callback.sync_call (*C var*), 2549
net_mgmt_event_handler_t (*C type*), 2544
net_mgmt_event_init (*C function*), 2547
net_mgmt_event_notify (*C function*), 2546
net_mgmt_event_notify_with_info (*C function*), 2545
net_mgmt_event_static_handler_t (*C type*), 2544
net_mgmt_event_wait (*C function*), 2546
net_mgmt_event_wait_on_iface (*C function*), 2546
net_mgmt_init_event_callback (*C function*), 2545
NET_MGMT_REGISTER_EVENT_HANDLER (*C macro*), 2544
NET_MGMT_REGISTER_REQUEST_HANDLER (*C macro*), 2544
net_mgmt_request_handler_t (*C type*), 2544
net_pkt (*C struct*), 2631
net_pkt_acknowledge_data (*C function*), 2631
net_pkt_alloc (*C function*), 2623
net_pkt_alloc_buffer (*C function*), 2623
net_pkt_alloc_buffer_raw (*C function*), 2624
net_pkt_alloc_from_slab (*C function*), 2623
net_pkt_alloc_on_iface (*C function*), 2623
net_pkt_alloc_with_buffer (*C function*), 2624
net_pkt_append_buffer (*C function*), 2624
net_pkt_available_buffer (*C function*), 2624
net_pkt_available_payload_buffer (*C function*), 2625
net_pkt_clone (*C function*), 2627
net_pkt_compact (*C function*), 2622
net_pkt_copy (*C function*), 2627
net_pkt_cursor_backup (*C function*), 2626
net_pkt_cursor_get_pos (*C function*), 2626
net_pkt_cursor_init (*C function*), 2625
net_pkt_cursor_restore (*C function*), 2626
NET_PKT_DATA_POOL_DEFINE (*C macro*), 2620
net_pkt_frag_add (*C function*), 2622
net_pkt_frag_del (*C function*), 2622
net_pkt_frag_insert (*C function*), 2622
net_pkt_frag_ref (*C function*), 2622
net_pkt_frag_unref (*C function*), 2622
net_pkt_get_contiguous_len (*C function*), 2631

net_pkt_get_current_offset (*C function*), 2630
net_pkt_get_data (*C function*), 2631
net_pkt_get_frag (*C function*), 2621
net_pkt_get_info (*C function*), 2623
net_pkt_get_reserve_data (*C function*), 2620
net_pkt_get_reserve_rx_data (*C function*), 2621
net_pkt_get_reserve_tx_data (*C function*), 2621
net_pkt_is_contiguous (*C function*), 2631
net_pkt_memset (*C function*), 2626
net_pkt_print_frags (*C macro*), 2620
net_pkt_pull (*C function*), 2630
net_pkt_read (*C function*), 2627
net_pkt_read_be16 (*C function*), 2628
net_pkt_read_be32 (*C function*), 2628
net_pkt_read_le16 (*C function*), 2628
net_pkt_read_u8 (*C function*), 2628
net_pkt_ref (*C function*), 2621
net_pkt_remaining_data (*C function*), 2630
net_pkt_remove_tail (*C function*), 2625
net_pkt_rx_alloc (*C function*), 2623
net_pkt_rx_clone (*C function*), 2627
net_pkt_set_data (*C function*), 2631
net_pkt_shallow_clone (*C function*), 2627
net_pkt_skip (*C function*), 2626
NET_PKT_SLAB_DEFINE (*C macro*), 2620
net_pkt_trim_buffer (*C function*), 2625
net_pkt_unref (*C function*), 2621
net_pkt_update_length (*C function*), 2630
net_pkt_write (*C function*), 2628
net_pkt_write_be16 (*C function*), 2629
net_pkt_write_be32 (*C function*), 2629
net_pkt_write_le16 (*C function*), 2630
net_pkt_write_le32 (*C function*), 2629
net_pkt_write_u8 (*C function*), 2629
net_pkt.buffer (*C var*), 2632
net_pkt.context (*C var*), 2632
net_pkt.cursor (*C var*), 2632
net_pkt.fifo (*C var*), 2632
net_pkt.frags (*C var*), 2632
net_pkt.iface (*C var*), 2632
net_pkt.slab (*C var*), 2632
net_port_set_default (*C function*), 2523
net_priority (*C enum*), 2512
net_priority2vlan (*C function*), 2524
net_priority.NET_PRIORITY_BE (*C enumerator*), 2512
net_priority.NET_PRIORITY_BK (*C enumerator*), 2512
net_priority.NET_PRIORITY_CA (*C enumerator*), 2512
net_priority.NET_PRIORITY_EE (*C enumerator*), 2512
net_priority.NET_PRIORITY_IC (*C enumerator*), 2512
net_priority.NET_PRIORITY_NC (*C enumerator*), 2512
net_priority.NET_PRIORITY_VI (*C enumerator*), 2512
net_priority.NET_PRIORITY_VO (*C enumerator*), 2512
net_promisc_mode_off (*C function*), 2565
net_promisc_mode_on (*C function*), 2565
net_promisc_mode_wait_data (*C function*), 2565
net_ptp_extended_time (*C struct*), 2967
net_ptp_extended_time.fract_nsecond (*C var*), 2967
net_ptp_extended_time.second (*C var*), 2967

net_ptp_time (*C struct*), 2966
net_ptp_time_to_ns (*C function*), 2965
net_ptp_time.nanosecond (*C var*), 2967
net_ptp_time.second (*C var*), 2967
net_recv_data (*C function*), 2896
NET_REQUEST_IEEE802154_ACTIVE_SCAN (*C macro*), 2661
NET_REQUEST_IEEE802154_ASSOCIATE (*C macro*), 2661
NET_REQUEST_IEEE802154_CANCEL_SCAN (*C macro*), 2661
NET_REQUEST_IEEE802154_DISASSOCIATE (*C macro*), 2661
NET_REQUEST_IEEE802154_GET_CHANNEL (*C macro*), 2662
NET_REQUEST_IEEE802154_GET_EXT_ADDR (*C macro*), 2662
NET_REQUEST_IEEE802154_GET_PAN_ID (*C macro*), 2662
NET_REQUEST_IEEE802154_GET_SECURITY_SETTINGS (*C macro*), 2662
NET_REQUEST_IEEE802154_GET_SHORT_ADDR (*C macro*), 2662
NET_REQUEST_IEEE802154_GET_TX_POWER (*C macro*), 2662
NET_REQUEST_IEEE802154_PASSIVE_SCAN (*C macro*), 2661
NET_REQUEST_IEEE802154_SET_ACK (*C macro*), 2661
NET_REQUEST_IEEE802154_SET_CHANNEL (*C macro*), 2661
NET_REQUEST_IEEE802154_SET_EXT_ADDR (*C macro*), 2662
NET_REQUEST_IEEE802154_SET_PAN_ID (*C macro*), 2662
NET_REQUEST_IEEE802154_SET_SECURITY_SETTINGS (*C macro*), 2662
NET_REQUEST_IEEE802154_SET_SHORT_ADDR (*C macro*), 2662
NET_REQUEST_IEEE802154_SET_TX_POWER (*C macro*), 2662
NET_REQUEST_IEEE802154_UNSET_ACK (*C macro*), 2661
NET_REQUEST_WIFI_AP_CONFIG_PARAM (*C macro*), 2716
NET_REQUEST_WIFI_AP_DISABLE (*C macro*), 2715
NET_REQUEST_WIFI_AP_ENABLE (*C macro*), 2715
NET_REQUEST_WIFI_AP_STA_DISCONNECT (*C macro*), 2716
NET_REQUEST_WIFI_CHANNEL (*C macro*), 2716
net_request_wifi_cmd (*C enum*), 2725
net_request_wifi_cmd.NET_REQUEST_WIFI_CMD_AP_CONFIG_PARAM (*C enumerator*), 2726
net_request_wifi_cmd.NET_REQUEST_WIFI_CMD_AP_DISABLE (*C enumerator*), 2726
net_request_wifi_cmd.NET_REQUEST_WIFI_CMD_AP_ENABLE (*C enumerator*), 2726
net_request_wifi_cmd.NET_REQUEST_WIFI_CMD_AP_STA_DISCONNECT (*C enumerator*), 2726
net_request_wifi_cmd.NET_REQUEST_WIFI_CMD_CHANNEL (*C enumerator*), 2726
net_request_wifi_cmd.NET_REQUEST_WIFI_CMD_CONNECT (*C enumerator*), 2726
net_request_wifi_cmd.NET_REQUEST_WIFI_CMD_DISCONNECT (*C enumerator*), 2726
net_request_wifi_cmd.NET_REQUEST_WIFI_CMD_IFACE_STATUS (*C enumerator*), 2726
net_request_wifi_cmd.NET_REQUEST_WIFI_CMD_MODE (*C enumerator*), 2726
net_request_wifi_cmd.NET_REQUEST_WIFI_CMD_PACKET_FILTER (*C enumerator*), 2726
net_request_wifi_cmd.NET_REQUEST_WIFI_CMD_PS (*C enumerator*), 2726
net_request_wifi_cmd.NET_REQUEST_WIFI_CMD_PS_CONFIG (*C enumerator*), 2726
net_request_wifi_cmd.NET_REQUEST_WIFI_CMD_REG_DOMAIN (*C enumerator*), 2726
net_request_wifi_cmd.NET_REQUEST_WIFI_CMD_RTS_THRESHOLD (*C enumerator*), 2726
net_request_wifi_cmd.NET_REQUEST_WIFI_CMD_SCAN (*C enumerator*), 2725
net_request_wifi_cmd.NET_REQUEST_WIFI_CMD_TWT (*C enumerator*), 2726
net_request_wifi_cmd.NET_REQUEST_WIFI_CMD_VERSION (*C enumerator*), 2726
NET_REQUEST_WIFI_CONNECT (*C macro*), 2715
NET_REQUEST_WIFI_DISCONNECT (*C macro*), 2715
NET_REQUEST_WIFI_IFACE_STATUS (*C macro*), 2715
NET_REQUEST_WIFI_MODE (*C macro*), 2715
NET_REQUEST_WIFI_PACKET_FILTER (*C macro*), 2715
NET_REQUEST_WIFI_PS (*C macro*), 2715
NET_REQUEST_WIFI_PS_CONFIG (*C macro*), 2715
NET_REQUEST_WIFI_REG_DOMAIN (*C macro*), 2715
NET_REQUEST_WIFI_RTS_THRESHOLD (*C macro*), 2716
NET_REQUEST_WIFI_SCAN (*C macro*), 2715
NET_REQUEST_WIFI_TWT (*C macro*), 2715

NET_REQUEST_WIFI_VERSION (*C macro*), 2716
net_rx_priority2tc (*C function*), 2524
net_send_data (*C function*), 2896
net_sin (*C function*), 2521
net_sin6 (*C function*), 2521
net_sin6_ptr (*C function*), 2521
net_sin_ptr (*C function*), 2522
net_sll_ptr (*C function*), 2522
net_sock_type (*C enum*), 2511
net_sock_type.SOCK_DGRAM (*C enumerator*), 2512
net_sock_type.SOCK_RAW (*C enumerator*), 2512
net_sock_type.SOCK_STREAM (*C enumerator*), 2511
net_socket_create_t (*C type*), 2902
net_stats (*C struct*), 2556
net_stats_bytes (*C struct*), 2550
net_stats_bytes.received (*C var*), 2550
net_stats_bytes.sent (*C var*), 2550
net_stats_eth (*C struct*), 2558
net_stats_eth_csum (*C struct*), 2558
net_stats_eth_csum.rx_csum_offload_errors (*C var*), 2558
net_stats_eth_csum.rx_csum_offload_good (*C var*), 2558
net_stats_eth_errors (*C struct*), 2556
net_stats_eth_errors.corr_ecc_errors (*C var*), 2557
net_stats_eth_errors.rx_align_errors (*C var*), 2557
net_stats_eth_errors.rx_buf_alloc_failed (*C var*), 2557
net_stats_eth_errors.rx_crc_errors (*C var*), 2556
net_stats_eth_errors.rx_dma_failed (*C var*), 2557
net_stats_eth_errors.rx_frame_errors (*C var*), 2556
net_stats_eth_errors.rx_length_errors (*C var*), 2556
net_stats_eth_errors.rx_long_length_errors (*C var*), 2556
net_stats_eth_errors.rx_missed_errors (*C var*), 2556
net_stats_eth_errors.rx_no_buffer_count (*C var*), 2556
net_stats_eth_errors.rx_over_errors (*C var*), 2556
net_stats_eth_errors.rx_short_length_errors (*C var*), 2556
net_stats_eth_errors.tx_aborted_errors (*C var*), 2557
net_stats_eth_errors.tx_carrier_errors (*C var*), 2557
net_stats_eth_errors.tx_dma_failed (*C var*), 2557
net_stats_eth_errors.tx_fifo_errors (*C var*), 2557
net_stats_eth_errors.tx_heartbeat_errors (*C var*), 2557
net_stats_eth_errors.tx_window_errors (*C var*), 2557
net_stats_eth_errors.uncorr_ecc_errors (*C var*), 2557
net_stats_eth_flow (*C struct*), 2557
net_stats_eth_flow.rx_flow_control_xoff (*C var*), 2557
net_stats_eth_flow.rx_flow_control_xon (*C var*), 2557
net_stats_eth_flow.tx_flow_control_xoff (*C var*), 2558
net_stats_eth_flow.tx_flow_control_xon (*C var*), 2557
net_stats_eth_hw_timestamp (*C struct*), 2558
net_stats_eth_hw_timestamp.rx_hwtstamp_cleared (*C var*), 2558
net_stats_eth_hw_timestamp.tx_hwtstamp_skipped (*C var*), 2558
net_stats_eth_hw_timestamp.tx_hwtstamp_timeouts (*C var*), 2558
net_stats_eth.broadcast (*C var*), 2558
net_stats_eth.bytes (*C var*), 2558
net_stats_eth.collisions (*C var*), 2559
net_stats_eth.csum (*C var*), 2559
net_stats_eth.error_details (*C var*), 2559
net_stats_eth.errors (*C var*), 2559
net_stats_eth.flow_control (*C var*), 2559
net_stats_eth.hw_timestamp (*C var*), 2559

net_stats_eth.multicast (*C var*), 2558
net_stats_eth.pkts (*C var*), 2558
net_stats_eth.tx_dropped (*C var*), 2559
net_stats_eth.tx_restart_queue (*C var*), 2559
net_stats_eth.tx_timeout_count (*C var*), 2559
net_stats_eth.unknown_protocol (*C var*), 2559
net_stats_icmp (*C struct*), 2551
net_stats_icmp.chkerr (*C var*), 2552
net_stats_icmp.drop (*C var*), 2552
net_stats_icmp.recv (*C var*), 2552
net_stats_icmp.sent (*C var*), 2552
net_stats_icmp.typeerr (*C var*), 2552
net_stats_ip (*C struct*), 2551
net_stats_ip_errors (*C struct*), 2551
net_stats_ip_errors.chkerr (*C var*), 2551
net_stats_ip_errors.fragerr (*C var*), 2551
net_stats_ip_errors.hblenerr (*C var*), 2551
net_stats_ip_errors.lblenerr (*C var*), 2551
net_stats_ip_errors.protoerr (*C var*), 2551
net_stats_ip_errors.vhlerr (*C var*), 2551
net_stats_ip.drop (*C var*), 2551
net_stats_ip.forwarded (*C var*), 2551
net_stats_ip.recv (*C var*), 2551
net_stats_ip.sent (*C var*), 2551
net_stats_ipv4_igmp (*C struct*), 2554
net_stats_ipv4_igmp.drop (*C var*), 2554
net_stats_ipv4_igmp.recv (*C var*), 2554
net_stats_ipv4_igmp.sent (*C var*), 2554
net_stats_ipv6_mld (*C struct*), 2554
net_stats_ipv6_mld.drop (*C var*), 2554
net_stats_ipv6_mld.recv (*C var*), 2554
net_stats_ipv6_mld.sent (*C var*), 2554
net_stats_ipv6_nd (*C struct*), 2553
net_stats_ipv6_nd.drop (*C var*), 2553
net_stats_ipv6_nd.recv (*C var*), 2553
net_stats_ipv6_nd.sent (*C var*), 2553
net_stats_pkts (*C struct*), 2550
net_stats_pkts.rx (*C var*), 2551
net_stats_pkts.tx (*C var*), 2551
net_stats_pm (*C struct*), 2555
net_stats_pm.last_suspend_time (*C var*), 2555
net_stats_pm.overall_suspend_time (*C var*), 2555
net_stats_pm.start_time (*C var*), 2556
net_stats_pm.suspend_count (*C var*), 2555
net_stats_ppp (*C struct*), 2559
net_stats_ppp.bytes (*C var*), 2559
net_stats_ppp.chkerr (*C var*), 2559
net_stats_ppp.drop (*C var*), 2559
net_stats_ppp.pkts (*C var*), 2559
net_stats_rx_time (*C struct*), 2554
net_stats_rx_time.count (*C var*), 2555
net_stats_rx_time.sum (*C var*), 2555
net_stats_sta_mgmt (*C struct*), 2560
net_stats_sta_mgmt.beacons_miss (*C var*), 2560
net_stats_sta_mgmt.beacons_rx (*C var*), 2560
net_stats_t (*C type*), 2550
net_stats_tc (*C struct*), 2555
net_stats_tc.bytes (*C var*), 2555

net_stats_tcp (*C struct*), 2552
net_stats_tcp.ackerr (*C var*), 2552
net_stats_tcp.bytes (*C var*), 2552
net_stats_tcp.chkerr (*C var*), 2552
net_stats_tcp.conndrop (*C var*), 2553
net_stats_tcp.connrst (*C var*), 2553
net_stats_tcp.drop (*C var*), 2552
net_stats_tc.pkts (*C var*), 2555
net_stats_tcp.recv (*C var*), 2552
net_stats_tcp.resent (*C var*), 2552
net_stats_tcp.rexmit (*C var*), 2553
net_stats_tc.priority (*C var*), 2555
net_stats_tcp.rst (*C var*), 2553
net_stats_tcp.rsterr (*C var*), 2553
net_stats_tcp.seg_drop (*C var*), 2552
net_stats_tcp.sent (*C var*), 2552
net_stats_tc.recv (*C var*), 2555
net_stats_tc.rx_time (*C var*), 2555
net_stats_tc.sent (*C var*), 2555
net_stats_tc.tx_time (*C var*), 2555
net_stats_tx_time (*C struct*), 2554
net_stats_tx_time.count (*C var*), 2554
net_stats_tx_time.sum (*C var*), 2554
net_stats_udp (*C struct*), 2553
net_stats_udp.chkerr (*C var*), 2553
net_stats_udp.drop (*C var*), 2553
net_stats_udp.recv (*C var*), 2553
net_stats_udp.sent (*C var*), 2553
net_stats_wifi (*C struct*), 2560
net_stats_wifi.broadcast (*C var*), 2560
net_stats_wifi.bytes (*C var*), 2560
net_stats_wifi.errors (*C var*), 2560
net_stats_wifi.multicast (*C var*), 2560
net_stats_wifi.pkts (*C var*), 2560
net_stats_wifi.sta_mgmt (*C var*), 2560
net_stats_wifi.unicast (*C var*), 2560
net_stats.bytes (*C var*), 2556
net_stats.ip_errors (*C var*), 2556
net_stats.processing_error (*C var*), 2556
net_tcp_seq_cmp (*C function*), 2523
net_tcp_seq_greater (*C function*), 2523
NET_TIME_MAX (*C macro*), 2963
NET_TIME_MIN (*C macro*), 2963
NET_TIME_SEC_MAX (*C macro*), 2963
NET_TIME_SEC_MIN (*C macro*), 2963
net_time_t (*C type*), 2964
net_timeout (*C struct*), 2563
net_timeout_deadline (*C function*), 2562
net_timeout_evaluate (*C function*), 2563
NET_TIMEOUT_MAX_VALUE (*C macro*), 2561
net_timeout_remaining (*C function*), 2562
net_timeout_set (*C function*), 2562
net_timeout.node (*C var*), 2563
net_timeout.timer_start (*C var*), 2563
net_timeout.timer_timeout (*C var*), 2563
net_timeout.wrap_counter (*C var*), 2563
net_traffic_class (*C struct*), 2935
net_traffic_class.fifo (*C var*), 2936

net_traffic_class.handler (*C var*), 2936
net_traffic_class.stack (*C var*), 2936
net_trickle (*C struct*), 2570
net_trickle_cb_t (*C type*), 2569
net_trickle_consistency (*C function*), 2569
net_trickle_create (*C function*), 2569
net_trickle_inconsistency (*C function*), 2569
net_trickle_is_running (*C function*), 2570
net_trickle_start (*C function*), 2569
net_trickle_stop (*C function*), 2569
net_trickle.c (*C var*), 2570
net_trickle.cb (*C var*), 2570
net_trickle.double_to (*C var*), 2570
net_trickle.I (*C var*), 2570
net_trickle.Imax (*C var*), 2570
net_trickle.Imax_abs (*C var*), 2570
net_trickle.Imin (*C var*), 2570
net_trickle.Istart (*C var*), 2570
net_trickle.k (*C var*), 2570
net_trickle.timer (*C var*), 2570
net_trickle.user_data (*C var*), 2571
net_tuple (*C struct*), 2528
net_tuple.ip_proto (*C var*), 2529
net_tuple.local_addr (*C var*), 2528
net_tuple.local_port (*C var*), 2529
net_tuple.remote_addr (*C var*), 2528
net_tuple.remote_port (*C var*), 2529
net_tx_priority2tc (*C function*), 2524
net_verdict (*C enum*), 2896
net_verdict.NET_CONTINUE (*C enumerator*), 2896
net_verdict.NET_DROP (*C enumerator*), 2896
net_verdict.NET_OK (*C enumerator*), 2896
net_vlan2priority (*C function*), 2524
NET_VLAN_TAG_UNSPEC (*C macro*), 2633
net_wifi_mgmt_offload (*C struct*), 2745
net_wifi_mgmt_offload.wifi_drv_ops (*C var*), 2746
net_wifi_mgmt_offload.wifi_iface (*C var*), 2745
net_wifi_mgmt_offload.wifi_mgmt_api (*C var*), 2745
NetworkPortHelper (*class in runners.core*), 196
NHPOT (*C macro*), 685
NI_DGRAM (*C macro*), 2493
NI_MAXHOST (*C macro*), 2493
NI_NAMEREQD (*C macro*), 2493
NI_NOFQDN (*C macro*), 2493
NI_NUMERICHOST (*C macro*), 2493
NI_NUMERICSERV (*C macro*), 2493
no_bus_emul (*C struct*), 3146
npf_append_rule (*C function*), 2948
npf_default_drop (*C var*), 2948
npf_default_ok (*C var*), 2948
NPF_ETH_DST_ADDR_MASK_MATCH (*C macro*), 2952
NPF_ETH_DST_ADDR_MATCH (*C macro*), 2951
NPF_ETH_DST_ADDR_UNMATCH (*C macro*), 2952
NPF_ETH_SRC_ADDR_MASK_MATCH (*C macro*), 2952
NPF_ETH_SRC_ADDR_MATCH (*C macro*), 2951
NPF_ETH_SRC_ADDR_UNMATCH (*C macro*), 2951
NPF_ETH_TYPE_MATCH (*C macro*), 2952
NPF_ETH_TYPE_UNMATCH (*C macro*), 2952

NPF_IFACE_MATCH (*C macro*), 2950
NPF_IFACE_UNMATCH (*C macro*), 2950
npf_insert_rule (*C function*), 2948
NPF_IP_SRC_ADDR_ALLOWLIST (*C macro*), 2950
NPF_IP_SRC_ADDR_BLOCKLIST (*C macro*), 2951
npf_ipv4_recv_rules (*C var*), 2949
npf_ipv6_recv_rules (*C var*), 2949
npf_local_in_recv_rules (*C var*), 2948
NPF_ORIG_IFACE_MATCH (*C macro*), 2950
NPF_ORIG_IFACE_UNMATCH (*C macro*), 2950
npf_recv_rules (*C var*), 2948
npf_remove_all_rules (*C function*), 2948
npf_remove_rule (*C function*), 2948
NPF_RULE (*C macro*), 2947
npf_rule (*C struct*), 2949
npf_rule_list (*C struct*), 2949
npf_rule_list.lock (*C var*), 2949
npf_rule_list.rule_head (*C var*), 2949
npf_rule.nb_tests (*C var*), 2949
npf_rule.node (*C var*), 2949
npf_rule.result (*C var*), 2949
npf_rule.tests (*C var*), 2949
npf_send_rules (*C var*), 2948
NPF_SIZE_BOUNDS (*C macro*), 2950
NPF_SIZE_MAX (*C macro*), 2950
NPF_SIZE_MIN (*C macro*), 2950
npf_test (*C struct*), 2949
npf_test.fn (*C var*), 2949
ns_to_net_ptp_time (*C function*), 2965
NSEC_PER_MSEC (*C macro*), 476
NSEC_PER_SEC (*C macro*), 477
NSEC_PER_USEC (*C macro*), 476
ntohl (*C macro*), 2508
ntohll (*C macro*), 2508
ntohs (*C macro*), 2508
NUM_SOP_STAR_TYPES (*C macro*), 3678
NUM_VA_ARGS (*C macro*), 695
NUM_VA_ARGS_LESS_1 (*C macro*), 694
nvs_calc_free_space (*C function*), 1180
nvs_clear (*C function*), 1178
nvs_delete (*C function*), 1179
nvs_fs (*C struct*), 1177
nvs_fs.ate_wra (*C var*), 1177
nvs_fs.data_wra (*C var*), 1177
nvs_fs.flash_device (*C var*), 1178
nvs_fs.flash_parameters (*C var*), 1178
nvs_fs.nvs_lock (*C var*), 1178
nvs_fs.offset (*C var*), 1177
nvs_fs.ready (*C var*), 1177
nvs_fs.sector_count (*C var*), 1177
nvs_fs.sector_size (*C var*), 1177
nvs_mount (*C function*), 1178
nvs_read (*C function*), 1179
nvs_read_hist (*C function*), 1179
nvs_write (*C function*), 1178

O

onoff_cancel (*C function*), 1006

onoff_cancel_or_release (*C function*), 1007
onoff_client (*C struct*), 1010
onoff_client_callback (*C type*), 1004
ONOFF_CLIENT_EXTENSION_POS (*C macro*), 1003
onoff_client.notify (*C var*), 1010
ONOFF_FLAG_ERROR (*C macro*), 1002
onoff_has_error (*C function*), 1005
onoff_manager (*C struct*), 1009
onoff_manager_init (*C function*), 1005
onoff_manager.clients (*C var*), 1010
onoff_manager.flags (*C var*), 1010
onoff_manager.last_res (*C var*), 1010
onoff_manager.lock (*C var*), 1010
onoff_manager.monitors (*C var*), 1010
onoff_manager.refs (*C var*), 1010
onoff_manager.transitions (*C var*), 1010
onoff_monitor (*C struct*), 1010
onoff_monitor_callback (*C type*), 1004
onoff_monitor_register (*C function*), 1008
onoff_monitor_unregister (*C function*), 1008
onoff_monitor.callback (*C var*), 1011
onoff_monitor.node (*C var*), 1011
onoff_notify_fn (*C type*), 1003
onoff_release (*C function*), 1006
onoff_request (*C function*), 1005
onoff_reset (*C function*), 1007
ONOFF_STATE_ERROR (*C macro*), 1003
ONOFF_STATE_MASK (*C macro*), 1002
ONOFF_STATE_OFF (*C macro*), 1003
ONOFF_STATE_ON (*C macro*), 1003
ONOFF_STATE_RESETTING (*C macro*), 1003
ONOFF_STATE_TO_OFF (*C macro*), 1003
ONOFF_STATE_TO_ON (*C macro*), 1003
onoff_sync_finalize (*C function*), 1009
onoff_sync_lock (*C function*), 1008
onoff_sync_service (*C struct*), 1011
onoff_sync_service.count (*C var*), 1011
onoff_sync_service.lock (*C var*), 1011
onoff_transition_fn (*C type*), 1004
onoff_transitions (*C struct*), 1009
ONOFF_TRANSITIONS_INITIALIZER (*C macro*), 1003
onoff_transitions.reset (*C var*), 1009
onoff_transitions.start (*C var*), 1009
onoff_transitions.stop (*C var*), 1009
openocd (*runners.core.RunnerConfig attribute*), 197
openocd_search (*runners.core.RunnerConfig attribute*), 197
openthread_api_mutex_lock (*C function*), 2710
openthread_api_mutex_try_lock (*C function*), 2710
openthread_api_mutex_unlock (*C function*), 2710
openthread_get_default_context (*C function*), 2709
openthread_get_default_instance (*C function*), 2709
openthread_start (*C function*), 2710
openthread_state_changed_cb (*C struct*), 2710
openthread_state_changed_cb_register (*C function*), 2709
openthread_state_changed_cb_unregister (*C function*), 2709
openthread_state_changed_cb.node (*C var*), 2711
openthread_state_changed_cb.state_changed_cb (*C var*), 2710
openthread_state_changed_cb.user_data (*C var*), 2711

openthread_thread_id_get (*C function*), 2709
os_mgmt_group_events (*C enum*), 771
os_mgmt_group_events.MGMT_EVT_OP_OS_MGMT_ALL (*C enumerator*), 771
os_mgmt_group_events.MGMT_EVT_OP_OS_MGMT_DATETIME_GET (*C enumerator*), 771
os_mgmt_group_events.MGMT_EVT_OP_OS_MGMT_DATETIME_SET (*C enumerator*), 771
os_mgmt_group_events.MGMT_EVT_OP_OS_MGMT_INFO_APPEND (*C enumerator*), 771
os_mgmt_group_events.MGMT_EVT_OP_OS_MGMT_INFO_CHECK (*C enumerator*), 771
os_mgmt_group_events.MGMT_EVT_OP_OS_MGMT_RESET (*C enumerator*), 771
OTS_OBJ_ID_DIR_LIST (*C macro*), 1939

P

PART_OF_ARRAY (*C macro*), 680
PATH, 7, 8, 16, 27, 96, 131, 136, 137, 284
pcie_alloc_irq (*C function*), 3513
pcie_bar (*C struct*), 3515
pcie_bdf_t (*C type*), 3511
PCIE_BUS_NUMBER (*C macro*), 3510
PCIE_BUS_NUMBER_VAL (*C macro*), 3510
PCIE_BUS_PRIMARY_NUMBER (*C macro*), 3510
PCIE_BUS_SECONDARY_NUMBER (*C macro*), 3510
PCIE_BUS_SUBORDINATE_NUMBER (*C macro*), 3510
PCIE_CONF_BAR0 (*C macro*), 3509
PCIE_CONF_BAR1 (*C macro*), 3509
PCIE_CONF_BAR2 (*C macro*), 3509
PCIE_CONF_BAR3 (*C macro*), 3509
PCIE_CONF_BAR4 (*C macro*), 3509
PCIE_CONF_BAR5 (*C macro*), 3509
PCIE_CONF_BAR_64 (*C macro*), 3509
PCIE_CONF_BAR_ADDR (*C macro*), 3509
PCIE_CONF_BAR_FLAGS (*C macro*), 3509
PCIE_CONF_BAR_INVAL (*C macro*), 3509
PCIE_CONF_BAR_INVAL64 (*C macro*), 3509
PCIE_CONF_BAR_INVAL_FLAGS (*C macro*), 3510
PCIE_CONF_BAR_IO (*C macro*), 3509
PCIE_CONF_BAR_IO_ADDR (*C macro*), 3509
PCIE_CONF_BAR_MEM (*C macro*), 3509
PCIE_CONF_BAR_NONE (*C macro*), 3509
PCIE_CONF_CAP_ID (*C macro*), 3508
PCIE_CONF_CAP_NEXT (*C macro*), 3508
PCIE_CONF_CAPPTR (*C macro*), 3508
PCIE_CONF_CAPPTR_FIRST (*C macro*), 3508
PCIE_CONF_CLASSREV (*C macro*), 3509
PCIE_CONF_CLASSREV_CLASS (*C macro*), 3509
PCIE_CONF_CLASSREV_PROGIF (*C macro*), 3509
PCIE_CONF_CLASSREV_REV (*C macro*), 3509
PCIE_CONF_CLASSREV_SUBCLASS (*C macro*), 3509
PCIE_CONF_CMDSTAT (*C macro*), 3508
PCIE_CONF_CMDSTAT_CAPS (*C macro*), 3508
PCIE_CONF_CMDSTAT_INTERRUPT (*C macro*), 3508
PCIE_CONF_CMDSTAT_IO (*C macro*), 3508
PCIE_CONF_CMDSTAT_MASTER (*C macro*), 3508
PCIE_CONF_CMDSTAT_MEM (*C macro*), 3508
PCIE_CONF_EXT_CAP_ID (*C macro*), 3508
PCIE_CONF_EXT_CAP_NEXT (*C macro*), 3508
PCIE_CONF_EXT_CAP_VER (*C macro*), 3508
PCIE_CONF_EXT_CAPPTR (*C macro*), 3508
PCIE_CONF_ID (*C macro*), 3508
PCIE_CONF_INTR (*C macro*), 3510

PCIE_CONF_INTR_IRQ (*C macro*), 3510
PCIE_CONF_INTR_IRQ_NONE (*C macro*), 3510
PCIE_CONF_MULTIFUNCTION (*C macro*), 3509
pcie_conf_read (*C function*), 3512
PCIE_CONF_TYPE (*C macro*), 3509
PCIE_CONF_TYPE_BRIDGE (*C macro*), 3509
PCIE_CONF_TYPE_CARDBUS_BRIDGE (*C macro*), 3509
PCIE_CONF_TYPE_GET (*C macro*), 3509
PCIE_CONF_TYPE_PCI_BRIDGE (*C macro*), 3509
PCIE_CONF_TYPE_STANDARD (*C macro*), 3509
pcie_conf_write (*C function*), 3512
pcie_connect_dynamic_irq (*C function*), 3514
pcie_dev (*C struct*), 3515
PCIE_DT_ID (*C macro*), 3507
PCIE_DT_INST_ID (*C macro*), 3507
pcie_get_cap (*C function*), 3514
pcie_get_ext_cap (*C function*), 3514
pcie_get_iobar (*C function*), 3513
pcie_get_irq (*C function*), 3514
pcie_get_mbar (*C function*), 3512
PCIE_HOST_CONTROLLER (*C macro*), 3508
PCIE_ID_IS_VALID (*C macro*), 3507
pcie_id_t (*C type*), 3511
PCIE_IO_BASE (*C macro*), 3510
PCIE_IO_BASE_LIMIT_UPPER (*C macro*), 3510
PCIE_IO_BASE_LIMIT_UPPER_VAL (*C macro*), 3510
PCIE_IO_BASE_UPPER (*C macro*), 3510
PCIE_IO_LIMIT (*C macro*), 3510
PCIE_IO_LIMIT_UPPER (*C macro*), 3510
PCIE_IO_SEC_STATUS (*C macro*), 3510
PCIE_IO_SEC_STATUS_VAL (*C macro*), 3510
PCIE_IRQ_CONNECT (*C macro*), 3511
pcie_irq_enable (*C function*), 3514
PCIE_MAX_BUS (*C macro*), 3511
PCIE_MAX_DEV (*C macro*), 3511
PCIE_MAX_FUNC (*C macro*), 3511
PCIE_MEM_BASE (*C macro*), 3510
PCIE_MEM_BASE_LIMIT (*C macro*), 3510
PCIE_MEM_BASE_LIMIT_VAL (*C macro*), 3510
PCIE_MEM_LIMIT (*C macro*), 3510
PCIE_PREFETCH_BASE (*C macro*), 3510
PCIE_PREFETCH_BASE_LIMIT (*C macro*), 3510
PCIE_PREFETCH_BASE_LIMIT_VAL (*C macro*), 3510
PCIE_PREFETCH_BASE_UPPER (*C macro*), 3510
PCIE_PREFETCH_LIMIT (*C macro*), 3510
PCIE_PREFETCH_LIMIT_UPPER (*C macro*), 3510
pcie_probe_iobar (*C function*), 3513
pcie_probe_mbar (*C function*), 3512
pcie_scan (*C function*), 3512
pcie_scan_cb_t (*C type*), 3511
pcie_scan_opt (*C struct*), 3515
pcie_scan_opt.bus (*C var*), 3515
pcie_scan_opt.cb (*C var*), 3515
pcie_scan_opt.cb_data (*C var*), 3515
pcie_scan_opt.flags (*C var*), 3515
PCIE_SEC_STATUS (*C macro*), 3510
PCIE_SECONDARY_LATENCY_TIMER (*C macro*), 3510
pcie_set_cmd (*C function*), 3513

pcm_stream_cfg (*C struct*), 3190
pcm_stream_cfg.block_size (*C var*), 3190
pcm_stream_cfg.mem_slab (*C var*), 3190
pcm_stream_cfg.pcm_rate (*C var*), 3190
pcm_stream_cfg.pcm_width (*C var*), 3190
pd_augmented_supply_pdo_sink (*C union*), 3689
pd_augmented_supply_pdo_sink.max_current (*C var*), 3689
pd_augmented_supply_pdo_sink.max_voltage (*C var*), 3689
pd_augmented_supply_pdo_sink.min_voltage (*C var*), 3689
pd_augmented_supply_pdo_sink.raw_value (*C var*), 3689
pd_augmented_supply_pdo_sink.reserved0 (*C var*), 3689
pd_augmented_supply_pdo_sink.reserved1 (*C var*), 3689
pd_augmented_supply_pdo_sink.reserved2 (*C var*), 3689
pd_augmented_supply_pdo_sink.reserved3 (*C var*), 3689
pd_augmented_supply_pdo_sink.type (*C var*), 3689
pd_augmented_supply_pdo_source (*C union*), 3688
pd_augmented_supply_pdo_source.max_current (*C var*), 3688
pd_augmented_supply_pdo_source.max_voltage (*C var*), 3688
pd_augmented_supply_pdo_source.min_voltage (*C var*), 3688
pd_augmented_supply_pdo_source.pps_power_limited (*C var*), 3688
pd_augmented_supply_pdo_source.raw_value (*C var*), 3689
pd_augmented_supply_pdo_source.reserved0 (*C var*), 3688
pd_augmented_supply_pdo_source.reserved1 (*C var*), 3688
pd_augmented_supply_pdo_source.reserved2 (*C var*), 3688
pd_augmented_supply_pdo_source.reserved3 (*C var*), 3688
pd_augmented_supply_pdo_source.type (*C var*), 3689
pd_battery_supply_pdo_sink (*C union*), 3687
pd_battery_supply_pdo_sink.max_voltage (*C var*), 3688
pd_battery_supply_pdo_sink.min_voltage (*C var*), 3688
pd_battery_supply_pdo_sink.operational_power (*C var*), 3688
pd_battery_supply_pdo_sink.raw_value (*C var*), 3688
pd_battery_supply_pdo_sink.type (*C var*), 3688
pd_battery_supply_pdo_source (*C union*), 3687
pd_battery_supply_pdo_source.max_power (*C var*), 3687
pd_battery_supply_pdo_source.max_voltage (*C var*), 3687
pd_battery_supply_pdo_source.min_voltage (*C var*), 3687
pd_battery_supply_pdo_source.raw_value (*C var*), 3687
pd_battery_supply_pdo_source.type (*C var*), 3687
PD_CONVERT_AUGMENTED_PDO_CURRENT_TO_MA (*C macro*), 3677
PD_CONVERT_AUGMENTED_PDO_VOLTAGE_TO_MV (*C macro*), 3678
PD_CONVERT_BATTERY_PDO_POWER_TO_MW (*C macro*), 3677
PD_CONVERT_BATTERY_PDO_VOLTAGE_TO_MV (*C macro*), 3677
PD_CONVERT_BYTES_TO_PD_HEADER_COUNT (*C macro*), 3675
PD_CONVERT_FIXED_PDO_CURRENT_TO_MA (*C macro*), 3676
PD_CONVERT_FIXED_PDO_VOLTAGE_TO_MV (*C macro*), 3676
PD_CONVERT_MA_TO_AUGMENTED_PDO_CURRENT (*C macro*), 3677
PD_CONVERT_MA_TO_FIXED_PDO_CURRENT (*C macro*), 3676
PD_CONVERT_MA_TO_VARIABLE_PDO_CURRENT (*C macro*), 3676
PD_CONVERT_MV_TO_AUGMENTED_PDO_VOLTAGE (*C macro*), 3677
PD_CONVERT_MV_TO_BATTERY_PDO_VOLTAGE (*C macro*), 3677
PD_CONVERT_MV_TO_FIXED_PDO_VOLTAGE (*C macro*), 3676
PD_CONVERT_MV_TO_VARIABLE_PDO_VOLTAGE (*C macro*), 3677
PD_CONVERT_MW_TO_BATTERY_PDO_POWER (*C macro*), 3677
PD_CONVERT_PD_HEADER_COUNT_TO_BYTES (*C macro*), 3676
PD_CONVERT_VARIABLE_PDO_CURRENT_TO_MA (*C macro*), 3677
PD_CONVERT_VARIABLE_PDO_VOLTAGE_TO_MV (*C macro*), 3677
pd_ctrl_msg_type (*C enum*), 3679
pd_ctrl_msg_type.PD_CTRL_ACCEPT (*C enumerator*), 3680

pd_ctrl_msg_type.PD_CTRL_DATA_RESET (*C enumerator*), 3680
pd_ctrl_msg_type.PD_CTRL_DATA_RESET_COMPLETE (*C enumerator*), 3680
pd_ctrl_msg_type.PD_CTRL_DR_SWAP (*C enumerator*), 3680
pd_ctrl_msg_type.PD_CTRL_FR_SWAP (*C enumerator*), 3681
pd_ctrl_msg_type.PD_CTRL_GET_COUNTRY_CODES (*C enumerator*), 3681
pd_ctrl_msg_type.PD_CTRL_GET_PPS_STATUS (*C enumerator*), 3681
pd_ctrl_msg_type.PD_CTRL_GET_SINK_CAP (*C enumerator*), 3680
pd_ctrl_msg_type.PD_CTRL_GET_SINK_CAP_EXT (*C enumerator*), 3681
pd_ctrl_msg_type.PD_CTRL_GET_SOURCE_CAP (*C enumerator*), 3680
pd_ctrl_msg_type.PD_CTRL_GET_SOURCE_CAP_EXT (*C enumerator*), 3681
pd_ctrl_msg_type.PD_CTRL_GET_STATUS (*C enumerator*), 3681
pd_ctrl_msg_type.PD_CTRL_GOOD_CRC (*C enumerator*), 3679
pd_ctrl_msg_type.PD_CTRL_GOTO_MIN (*C enumerator*), 3680
pd_ctrl_msg_type.PD_CTRL_NOT_SUPPORTED (*C enumerator*), 3680
pd_ctrl_msg_type.PD_CTRL_PING (*C enumerator*), 3680
pd_ctrl_msg_type.PD_CTRL_PR_SWAP (*C enumerator*), 3680
pd_ctrl_msg_type.PD_CTRL_PS_RDY (*C enumerator*), 3680
pd_ctrl_msg_type.PD_CTRL_REJECT (*C enumerator*), 3680
pd_ctrl_msg_type.PD_CTRL_SOFT_RESET (*C enumerator*), 3680
pd_ctrl_msg_type.PD_CTRL_VCONN_SWAP (*C enumerator*), 3680
pd_ctrl_msg_type.PD_CTRL_WAIT (*C enumerator*), 3680
pd_data_msg_type (*C enum*), 3681
pd_data_msg_type.PD_DATA_ALERT (*C enumerator*), 3681
pd_data_msg_type.PD_DATA_BATTERY_STATUS (*C enumerator*), 3681
pd_data_msg_type.PD_DATA_BIST (*C enumerator*), 3681
pd_data_msg_type.PD_DATA_ENTER_USB (*C enumerator*), 3681
pd_data_msg_type.PD_DATA_GET_COUNTRY_INFO (*C enumerator*), 3681
pd_data_msg_type.PD_DATA_REQUEST (*C enumerator*), 3681
pd_data_msg_type.PD_DATA_SINK_CAP (*C enumerator*), 3681
pd_data_msg_type.PD_DATA_SOURCE_CAP (*C enumerator*), 3681
pd_data_msg_type.PD_DATA_VENDOR_DEF (*C enumerator*), 3682
pd_ext_header (*C union*), 3683
pd_ext_header.chunk_number (*C var*), 3684
pd_ext_header.chunked (*C var*), 3684
pd_ext_header.data_size (*C var*), 3684
pd_ext_header.raw_value (*C var*), 3684
pd_ext_header.request_chunk (*C var*), 3684
pd_ext_header.reserved0 (*C var*), 3684
pd_ext_msg_type (*C enum*), 3682
pd_ext_msg_type.PD_EXT_BATTERY_CAP (*C enumerator*), 3682
pd_ext_msg_type.PD_EXT_COUNTRY_CODES (*C enumerator*), 3683
pd_ext_msg_type.PD_EXT_COUNTRY_INFO (*C enumerator*), 3682
pd_ext_msg_type.PD_EXT_FIRMWARE_UPDATE_REQUEST (*C enumerator*), 3682
pd_ext_msg_type.PD_EXT_FIRMWARE_UPDATE_RESPONSE (*C enumerator*), 3682
pd_ext_msg_type.PD_EXT_GET_BATTERY_CAP (*C enumerator*), 3682
pd_ext_msg_type.PD_EXT_GET_BATTERY_STATUS (*C enumerator*), 3682
pd_ext_msg_type.PD_EXT_GET_MANUFACTURER_INFO (*C enumerator*), 3682
pd_ext_msg_type.PD_EXT_MANUFACTURER_INFO (*C enumerator*), 3682
pd_ext_msg_type.PD_EXT_PPS_STATUS (*C enumerator*), 3682
pd_ext_msg_type.PD_EXT_SECURITY_REQUEST (*C enumerator*), 3682
pd_ext_msg_type.PD_EXT_SECURITY_RESPONSE (*C enumerator*), 3682
pd_ext_msg_type.PD_EXT_SOURCE_CAP (*C enumerator*), 3682
pd_ext_msg_type.PD_EXT_STATUS (*C enumerator*), 3682
pd_fixed_supply_pdo_sink (*C union*), 3685
pd_fixed_supply_pdo_sink.dual_role_data (*C var*), 3685
pd_fixed_supply_pdo_sink.dual_role_power (*C var*), 3686
pd_fixed_supply_pdo_sink.frs_required (*C var*), 3685
pd_fixed_supply_pdo_sink.higher_capability (*C var*), 3686

pd_fixed_supply_pdo_sink.operational_current (*C var*), 3685
pd_fixed_supply_pdo_sink.raw_value (*C var*), 3686
pd_fixed_supply_pdo_sink.reserved0 (*C var*), 3685
pd_fixed_supply_pdo_sink.type (*C var*), 3686
pd_fixed_supply_pdo_sink.unconstrained_power (*C var*), 3685
pd_fixed_supply_pdo_sink.usb_comms_capable (*C var*), 3685
pd_fixed_supply_pdo_sink.voltage (*C var*), 3685
pd_fixed_supply_pdo_source (*C union*), 3684
pd_fixed_supply_pdo_source.dual_role_data (*C var*), 3684
pd_fixed_supply_pdo_source.dual_role_power (*C var*), 3685
pd_fixed_supply_pdo_source.max_current (*C var*), 3684
pd_fixed_supply_pdo_source.peak_current (*C var*), 3684
pd_fixed_supply_pdo_source.raw_value (*C var*), 3685
pd_fixed_supply_pdo_source.reserved0 (*C var*), 3684
pd_fixed_supply_pdo_source.type (*C var*), 3685
pd_fixed_supply_pdo_source.unchunked_ext_msg_supported (*C var*), 3684
pd_fixed_supply_pdo_source.unconstrained_power (*C var*), 3685
pd_fixed_supply_pdo_source.usb_comms_capable (*C var*), 3685
pd_fixed_supply_pdo_source.usb_suspend_supported (*C var*), 3685
pd_fixed_supply_pdo_source.voltage (*C var*), 3684
pd_frs_type (*C enum*), 3678
pd_frs_type.FRS_1P5A_5V (*C enumerator*), 3678
pd_frs_type.FRS_3P0A_5V (*C enumerator*), 3678
pd_frs_type.FRS_DEFAULT_USB_POWER (*C enumerator*), 3678
pd_frs_type.FRS_NOT_SUPPORTED (*C enumerator*), 3678
PD_GET_EXT_HEADER (*C macro*), 3676
pd_header (*C union*), 3683
pd_header.extended (*C var*), 3683
pd_header.message_id (*C var*), 3683
pd_header.message_type (*C var*), 3683
pd_header.number_of_data_objects (*C var*), 3683
pd_header.port_data_role (*C var*), 3683
pd_header.port_power_role (*C var*), 3683
pd_header.raw_value (*C var*), 3683
pd_header.specification_revision (*C var*), 3683
PD_MAX_EXTENDED_MSG_CHUNK_LEN (*C macro*), 3674
PD_MAX_EXTENDED_MSG_LEGACY_LEN (*C macro*), 3673
PD_MAX_EXTENDED_MSG_LEN (*C macro*), 3673
pd_msg (*C struct*), 3691
pd_msg.data (*C var*), 3691
pd_msg.header (*C var*), 3691
pd_msg.len (*C var*), 3691
pd_msg.type (*C var*), 3691
PD_N_CAPS_COUNT (*C macro*), 3672
PD_N_HARD_RESET_COUNT (*C macro*), 3672
pd_packet_type (*C enum*), 3679
pd_packet_type.PD_PACKET_CABLE_RESET (*C enumerator*), 3679
pd_packet_type.PD_PACKET_DEBUG_PRIME (*C enumerator*), 3679
pd_packet_type.PD_PACKET_DEBUG_PRIME_PRIME (*C enumerator*), 3679
pd_packet_type.PD_PACKET_MSG_INVALID (*C enumerator*), 3679
pd_packet_type.PD_PACKET_PRIME_PRIME (*C enumerator*), 3679
pd_packet_type.PD_PACKET_SOP (*C enumerator*), 3679
pd_packet_type.PD_PACKET_SOP_PRIME (*C enumerator*), 3679
pd_packet_type.PD_PACKET_TX_BIST_MODE_2 (*C enumerator*), 3679
pd_packet_type.PD_PACKET_TX_HARD_RESET (*C enumerator*), 3679
pd_rdo (*C union*), 3690
pd_rdo.augmented (*C var*), 3691
pd_rdo.battery (*C var*), 3691

pd_rdo.cap_mismatch (*C var*), 3690
pd_rdo.fixed (*C var*), 3690
pd_rdo.giveback (*C var*), 3690
pd_rdo.min_operating_power (*C var*), 3691
pd_rdo.min_or_max_operating_current (*C var*), 3690
pd_rdo.no_usb_suspend (*C var*), 3690
pd_rdo.object_pos (*C var*), 3690
pd_rdo.operating_current (*C var*), 3690
pd_rdo.operating_power (*C var*), 3691
pd_rdo.output_voltage (*C var*), 3691
pd_rdo.raw_value (*C var*), 3691
pd_rdo.reserved0 (*C var*), 3690
pd_rdo.reserved1 (*C var*), 3690
pd_rdo.reserved2 (*C var*), 3691
pd_rdo.reserved3 (*C var*), 3691
pd_rdo.unchunked_ext_msg_supported (*C var*), 3690
pd_rdo.usb_comm_capable (*C var*), 3690
pd_rdo.variable (*C var*), 3690
pd_rev_type (*C enum*), 3678
pd_rev_type.PD_REV10 (*C enumerator*), 3679
pd_rev_type.PD_REV20 (*C enumerator*), 3679
pd_rev_type.PD_REV30 (*C enumerator*), 3679
PD_T_CHUNKING_NOT_SUPPORTED_MAX_MS (*C macro*), 3675
PD_T_CHUNKING_NOT_SUPPORTED_MIN_MS (*C macro*), 3675
PD_T_CHUNKING_NOT_SUPPORTED_NOM_MS (*C macro*), 3675
PD_T_EPR_PS_TRANSITION_MAX_MS (*C macro*), 3675
PD_T_EPR_PS_TRANSITION_MIN_MS (*C macro*), 3675
PD_T_EPR_PS_TRANSITION_NOM_MS (*C macro*), 3675
PD_T_HARD_RESET_COMPLETE_MAX_MS (*C macro*), 3674
PD_T_HARD_RESET_COMPLETE_MIN_MS (*C macro*), 3674
PD_T_NO_RESPONSE_MAX_MS (*C macro*), 3673
PD_T_NO_RESPONSE_MIN_MS (*C macro*), 3672
PD_T_PS_HARD_RESET_MAX_MS (*C macro*), 3673
PD_T_PS_HARD_RESET_MIN_MS (*C macro*), 3673
PD_T_SAFE_0V_MAX_MS (*C macro*), 3674
PD_T_SAFE_5V_MAX_MS (*C macro*), 3674
PD_T_SENDER_RESPONSE_MAX_MS (*C macro*), 3675
PD_T_SENDER_RESPONSE_MIN_MS (*C macro*), 3674
PD_T_SENDER_RESPONSE_NOM_MS (*C macro*), 3675
PD_T_SINK_REQUEST_MIN_MS (*C macro*), 3675
PD_T_SINK_TX_MAX_MS (*C macro*), 3673
PD_T_SINK_TX_MIN_MS (*C macro*), 3673
PD_T_SPR_PS_TRANSITION_MAX_MS (*C macro*), 3675
PD_T_SPR_PS_TRANSITION_MIN_MS (*C macro*), 3675
PD_T_SPR_PS_TRANSITION_NOM_MS (*C macro*), 3675
PD_T_TX_TIMEOUT_MS (*C macro*), 3674
PD_T_TYPEC_SEND_SOURCE_CAP_MAX_MS (*C macro*), 3673
PD_T_TYPEC_SEND_SOURCE_CAP_MIN_MS (*C macro*), 3673
PD_T_TYPEC_SINK_WAIT_CAP_MAX_MS (*C macro*), 3674
PD_T_TYPEC_SINK_WAIT_CAP_MIN_MS (*C macro*), 3674
PD_V_SAFE_0V_MAX_MV (*C macro*), 3674
PD_V_SAFE_5V_MIN_MV (*C macro*), 3674
pd_variable_supply_pdo_sink (*C union*), 3686
pd_variable_supply_pdo_sink.max_voltage (*C var*), 3687
pd_variable_supply_pdo_sink.min_voltage (*C var*), 3687
pd_variable_supply_pdo_sink.operational_current (*C var*), 3687
pd_variable_supply_pdo_sink.raw_value (*C var*), 3687
pd_variable_supply_pdo_sink.type (*C var*), 3687

pd_variable_supply_pdo_source (*C union*), 3686
pd_variable_supply_pdo_source.max_current (*C var*), 3686
pd_variable_supply_pdo_source.max_voltage (*C var*), 3686
pd_variable_supply_pdo_source.min_voltage (*C var*), 3686
pd_variable_supply_pdo_source.raw_value (*C var*), 3686
pd_variable_supply_pdo_source.type (*C var*), 3686
pdm_chan_cfg (*C struct*), 3190
pdm_chan_cfg.act_chan_map_hi (*C var*), 3191
pdm_chan_cfg.act_chan_map_lo (*C var*), 3191
pdm_chan_cfg.act_num_chan (*C var*), 3191
pdm_chan_cfg.act_num_streams (*C var*), 3191
pdm_chan_cfg.req_chan_map_hi (*C var*), 3191
pdm_chan_cfg.req_chan_map_lo (*C var*), 3191
pdm_chan_cfg.req_num_chan (*C var*), 3191
pdm_chan_cfg.req_num_streams (*C var*), 3191
pdm_io_cfg (*C struct*), 3189
pdm_io_cfg.max_pdm_clk_dc (*C var*), 3190
pdm_io_cfg.max_pdm_clk_freq (*C var*), 3189
pdm_io_cfg.min_pdm_clk_dc (*C var*), 3189
pdm_io_cfg.min_pdm_clk_freq (*C var*), 3189
pdm_io_cfg.pdm_clk_pol (*C var*), 3190
pdm_io_cfg.pdm_clk_skew (*C var*), 3190
pdm_io_cfg.pdm_data_pol (*C var*), 3190
pdm_lr (*C enum*), 3187
pdm_lr.PDM_CHAN_LEFT (*C enumerator*), 3187
pdm_lr.PDM_CHAN_RIGHT (*C enumerator*), 3188
PDO_MAX_DATA_OBJECTS (*C macro*), 3676
pdo_type (*C enum*), 3678
pdo_type.PDO_AUGMENTED (*C enumerator*), 3678
pdo_type.PDO_BATTERY (*C enumerator*), 3678
pdo_type.PDO_FIXED (*C enumerator*), 3678
pdo_type.PDO_VARIABLE (*C enumerator*), 3678
peci_buf (*C struct*), 3522
peci_buf.buf (*C var*), 3522
peci_buf.len (*C var*), 3522
PECI_CC_ILLEGAL_REQUEST (*C macro*), 3516
PECI_CC_OUT_OF_RESOURCES_TIMEOUT (*C macro*), 3516
PECI_CC_RESOURCES_LOWPWR_TIMEOUT (*C macro*), 3516
PECI_CC_RSP_SUCCESS (*C macro*), 3516
PECI_CC_RSP_TIMEOUT (*C macro*), 3516
peci_command_code (*C enum*), 3520
peci_command_code.PECI_CMD_GET_DIB (*C enumerator*), 3521
peci_command_code.PECI_CMD_GET_TEMP0 (*C enumerator*), 3520
peci_command_code.PECI_CMD_GET_TEMP1 (*C enumerator*), 3520
peci_command_code.PECI_CMD_PING (*C enumerator*), 3520
peci_command_code.PECI_CMD_RD_IAMSR0 (*C enumerator*), 3521
peci_command_code.PECI_CMD_RD_IAMSR1 (*C enumerator*), 3521
peci_command_code.PECI_CMD_RD_PCI_CFG0 (*C enumerator*), 3520
peci_command_code.PECI_CMD_RD_PCI_CFG1 (*C enumerator*), 3520
peci_command_code.PECI_CMD_RD_PCI_CFG_LOCAL0 (*C enumerator*), 3521
peci_command_code.PECI_CMD_RD_PCI_CFG_LOCAL1 (*C enumerator*), 3521
peci_command_code.PECI_CMD_RD_PKG_CFG0 (*C enumerator*), 3520
peci_command_code.PECI_CMD_RD_PKG_CFG1 (*C enumerator*), 3520
peci_command_code.PECI_CMD_WR_IAMSR0 (*C enumerator*), 3521
peci_command_code.PECI_CMD_WR_IAMSR1 (*C enumerator*), 3521
peci_command_code.PECI_CMD_WR_PCI_CFG0 (*C enumerator*), 3520
peci_command_code.PECI_CMD_WR_PCI_CFG1 (*C enumerator*), 3520
peci_command_code.PECI_CMD_WR_PCI_CFG_LOCAL0 (*C enumerator*), 3521

peci_command_code.PECI_CMD_WR_PCI_CFG_LOCAL1 (*C enumerator*), 3521
peci_command_code.PECI_CMD_WR_PKG_CFG0 (*C enumerator*), 3521
peci_command_code.PECI_CMD_WR_PKG_CFG1 (*C enumerator*), 3521
peci_config (*C function*), 3521
peci_disable (*C function*), 3521
peci_enable (*C function*), 3521
peci_error_code (*C enum*), 3520
peci_error_code.PECI_GENERAL_SENSOR_ERROR (*C enumerator*), 3520
peci_error_code.PECI_OVERFLOW_SENSOR_ERROR (*C enumerator*), 3520
peci_error_code.PECI_UNDERFLOW_SENSOR_ERROR (*C enumerator*), 3520
PECI_GET_DIB_CMD_LEN (*C macro*), 3517
PECI_GET_DIB_DEVINFO (*C macro*), 3517
PECI_GET_DIB_DOMAIN_BIT_MASK (*C macro*), 3517
PECI_GET_DIB_MAJOR_REV_MASK (*C macro*), 3517
PECI_GET_DIB_MINOR_REV_MASK (*C macro*), 3517
PECI_GET_DIB_RD_LEN (*C macro*), 3517
PECI_GET_DIB_REVNUM (*C macro*), 3517
PECI_GET_DIB_WR_LEN (*C macro*), 3517
PECI_GET_TEMP_CMD_LEN (*C macro*), 3517
PECI_GET_TEMP_ERR_LSB_GENERAL (*C macro*), 3517
PECI_GET_TEMP_ERR_LSB_RES (*C macro*), 3517
PECI_GET_TEMP_ERR_LSB_TEMP_HI (*C macro*), 3517
PECI_GET_TEMP_ERR_LSB_TEMP_LO (*C macro*), 3517
PECI_GET_TEMP_ERR_MSB (*C macro*), 3517
PECI_GET_TEMP_LSB (*C macro*), 3517
PECI_GET_TEMP_MSB (*C macro*), 3517
PECI_GET_TEMP_RD_LEN (*C macro*), 3517
PECI_GET_TEMP_WR_LEN (*C macro*), 3517
peci_msg (*C struct*), 3522
peci_msg.addr (*C var*), 3522
peci_msg.cmd_code (*C var*), 3522
peci_msg.flags (*C var*), 3523
peci_msg.rx_buffer (*C var*), 3523
peci_msg.tx_buffer (*C var*), 3522
PECI_PING_LEN (*C macro*), 3516
PECI_PING_RD_LEN (*C macro*), 3516
PECI_PING_WR_LEN (*C macro*), 3516
PECI_RD_IAMSR_CMD_LEN (*C macro*), 3518
PECI_RD_IAMSR_LEN_BYTE (*C macro*), 3518
PECI_RD_IAMSR_LEN_DWORD (*C macro*), 3518
PECI_RD_IAMSR_LEN_QWORD (*C macro*), 3518
PECI_RD_IAMSR_LEN_WORD (*C macro*), 3518
PECI_RD_IAMSR_WR_LEN (*C macro*), 3518
PECI_RD_PCICFG_CMD_LEN (*C macro*), 3519
PECI_RD_PCICFG_LEN_BYTE (*C macro*), 3519
PECI_RD_PCICFG_LEN_DWORD (*C macro*), 3519
PECI_RD_PCICFG_LEN_WORD (*C macro*), 3519
PECI_RD_PCICFG_WR_LEN (*C macro*), 3519
PECI_RD_PCICFGL_CMD_LEN (*C macro*), 3519
PECI_RD_PCICFGL_RD_LEN_BYTE (*C macro*), 3519
PECI_RD_PCICFGL_RD_LEN_DWORD (*C macro*), 3519
PECI_RD_PCICFGL_RD_LEN_WORD (*C macro*), 3519
PECI_RD_PCICFGL_WR_LEN (*C macro*), 3519
PECI_RD_PKG_CMD_LEN (*C macro*), 3518
PECI_RD_PKG_LEN_BYTE (*C macro*), 3517
PECI_RD_PKG_LEN_DWORD (*C macro*), 3518
PECI_RD_PKG_LEN_WORD (*C macro*), 3518
PECI_RD_PKG_WR_LEN (*C macro*), 3517

peci_transfer (*C function*), 3522
PECI_WR_IAMSR_CMD_LEN (*C macro*), 3519
PECI_WR_IAMSR_LEN_BYTE (*C macro*), 3518
PECI_WR_IAMSR_LEN_DWORD (*C macro*), 3518
PECI_WR_IAMSR_LEN_QWORD (*C macro*), 3519
PECI_WR_IAMSR_LEN_WORD (*C macro*), 3518
PECI_WR_IAMSR_RD_LEN (*C macro*), 3518
PECI_WR_PCICFG_CMD_LEN (*C macro*), 3519
PECI_WR_PCICFG_LEN_BYTE (*C macro*), 3519
PECI_WR_PCICFG_LEN_DWORD (*C macro*), 3519
PECI_WR_PCICFG_LEN_WORD (*C macro*), 3519
PECI_WR_PCICFG_RD_LEN (*C macro*), 3519
PECI_WR_PCICFGL_CMD_LEN (*C macro*), 3520
PECI_WR_PCICFGL_RD_LEN (*C macro*), 3520
PECI_WR_PCICFGL_WR_LEN_BYTE (*C macro*), 3520
PECI_WR_PCICFGL_WR_LEN_DWORD (*C macro*), 3520
PECI_WR_PCICFGL_WR_LEN_WORD (*C macro*), 3520
PECI_WR_PKG_CMD_LEN (*C macro*), 3518
PECI_WR_PKG_LEN_BYTE (*C macro*), 3518
PECI_WR_PKG_LEN_DWORD (*C macro*), 3518
PECI_WR_PKG_LEN_WORD (*C macro*), 3518
PECI_WR_PKG_RD_LEN (*C macro*), 3518
PF_CAN (*C macro*), 2507
PF_INET (*C macro*), 2507
PF_INET6 (*C macro*), 2507
PF_LOCAL (*C macro*), 2507
PF_NET_MGMT (*C macro*), 2507
PF_PACKET (*C macro*), 2507
PF_UNIX (*C macro*), 2507
PF_UNSPEC (*C macro*), 2507
PhonyNameDueToError.BT_AUDIO_CODEC_QOS_1M (*C enumerator*), 1742
PhonyNameDueToError.BT_AUDIO_CODEC_QOS_2M (*C enumerator*), 1742
PhonyNameDueToError.BT_AUDIO_CODEC_QOS_CODED (*C enumerator*), 1742
PhonyNameDueToError.BT_CONN_LE_OPT_CODED (*C enumerator*), 2315
PhonyNameDueToError.BT_CONN_LE_OPT_NO_1M (*C enumerator*), 2316
PhonyNameDueToError.BT_CONN_LE_OPT_NONE (*C enumerator*), 2315
PhonyNameDueToError.BT_CONN_LE_PHY_OPT_CODED_S2 (*C enumerator*), 2312
PhonyNameDueToError.BT_CONN_LE_PHY_OPT_CODED_S8 (*C enumerator*), 2312
PhonyNameDueToError.BT_CONN_LE_PHY_OPT_NONE (*C enumerator*), 2312
PhonyNameDueToError.BT_CONN_ROLE_CENTRAL (*C enumerator*), 2313
PhonyNameDueToError.BT_CONN_ROLE_PERIPHERAL (*C enumerator*), 2313
PhonyNameDueToError.BT_GAP_ADV_PROP_CONNECTABLE (*C enumerator*), 2032
PhonyNameDueToError.BT_GAP_ADV_PROP_DIRECTED (*C enumerator*), 2032
PhonyNameDueToError.BT_GAP_ADV_PROP_EXT_ADV (*C enumerator*), 2032
PhonyNameDueToError.BT_GAP_ADV_PROP_SCAN_RESPONSE (*C enumerator*), 2032
PhonyNameDueToError.BT_GAP_ADV_PROP_SCANNABLE (*C enumerator*), 2032
PhonyNameDueToError.BT_GAP_ADV_TYPE_ADV_DIRECT_IND (*C enumerator*), 2031
PhonyNameDueToError.BT_GAP_ADV_TYPE_ADV_IND (*C enumerator*), 2031
PhonyNameDueToError.BT_GAP_ADV_TYPE_ADV_NONCONN_IND (*C enumerator*), 2031
PhonyNameDueToError.BT_GAP_ADV_TYPE_ADV_SCAN_IND (*C enumerator*), 2031
PhonyNameDueToError.BT_GAP_ADV_TYPE_EXT_ADV (*C enumerator*), 2031
PhonyNameDueToError.BT_GAP_ADV_TYPE_SCAN_RSP (*C enumerator*), 2031
PhonyNameDueToError.BT_GAP_CTE_AOA (*C enumerator*), 2032
PhonyNameDueToError.BT_GAP_CTE_AOD_1US (*C enumerator*), 2032
PhonyNameDueToError.BT_GAP_CTE_AOD_2US (*C enumerator*), 2032
PhonyNameDueToError.BT_GAP_CTE_NONE (*C enumerator*), 2032
PhonyNameDueToError.BT_GAP_LE_PHY_1M (*C enumerator*), 2031
PhonyNameDueToError.BT_GAP_LE_PHY_2M (*C enumerator*), 2031

PhonyNameDueToError.BT_GAP_LE_PHY_CODED (*C enumerator*), 2031
PhonyNameDueToError.BT_GAP_LE_PHY_NONE (*C enumerator*), 2031
PhonyNameDueToError.BT_GAP_SCA_0_20 (*C enumerator*), 2033
PhonyNameDueToError.BT_GAP_SCA_21_30 (*C enumerator*), 2033
PhonyNameDueToError.BT_GAP_SCA_31_50 (*C enumerator*), 2033
PhonyNameDueToError.BT_GAP_SCA_51_75 (*C enumerator*), 2033
PhonyNameDueToError.BT_GAP_SCA_76_100 (*C enumerator*), 2033
PhonyNameDueToError.BT_GAP_SCA_101_150 (*C enumerator*), 2032
PhonyNameDueToError.BT_GAP_SCA_151_250 (*C enumerator*), 2032
PhonyNameDueToError.BT_GAP_SCA_251_500 (*C enumerator*), 2032
PhonyNameDueToError.BT_GAP_SCA_UNKNOWN (*C enumerator*), 2032
PhonyNameDueToError.BT_GATT_DISCOVER_ATTRIBUTE (*C enumerator*), 2061
PhonyNameDueToError.BT_GATT_DISCOVER_CHARACTERISTIC (*C enumerator*), 2061
PhonyNameDueToError.BT_GATT_DISCOVER_DESCRIPTOR (*C enumerator*), 2061
PhonyNameDueToError.BT_GATT_DISCOVER_INCLUDE (*C enumerator*), 2060
PhonyNameDueToError.BT_GATT_DISCOVER_PRIMARY (*C enumerator*), 2060
PhonyNameDueToError.BT_GATT_DISCOVER_SECONDARY (*C enumerator*), 2060
PhonyNameDueToError.BT_GATT_DISCOVER_STD_CHAR_DESC (*C enumerator*), 2061
PhonyNameDueToError.BT_GATT_ITER_CONTINUE (*C enumerator*), 2047
PhonyNameDueToError.BT_GATT_ITER_STOP (*C enumerator*), 2047
PhonyNameDueToError.BT_GATT_SUBSCRIBE_FLAG_NO_RESUB (*C enumerator*), 2061
PhonyNameDueToError.BT_GATT_SUBSCRIBE_FLAG_SENT (*C enumerator*), 2062
PhonyNameDueToError.BT_GATT_SUBSCRIBE_FLAG_VOLATILE (*C enumerator*), 2061
PhonyNameDueToError.BT_GATT_SUBSCRIBE_FLAG_WRITE_PENDING (*C enumerator*), 2062
PhonyNameDueToError.BT_GATT_SUBSCRIBE_NUM_FLAGS (*C enumerator*), 2062
PhonyNameDueToError.BT_GATT_WRITE_FLAG_CMD (*C enumerator*), 2039
PhonyNameDueToError.BT_GATT_WRITE_FLAG_EXECUTE (*C enumerator*), 2039
PhonyNameDueToError.BT_GATT_WRITE_FLAG_PREPARE (*C enumerator*), 2039
PhonyNameDueToError.BT_HCI_RAW_MODE_H4 (*C enumerator*), 2351
PhonyNameDueToError.BT_HCI_RAW_MODE_PASSTHROUGH (*C enumerator*), 2351
PhonyNameDueToError.BT_LE_ADV_OPT_ANONYMOUS (*C enumerator*), 1965
PhonyNameDueToError.BT_LE_ADV_OPT_CODED (*C enumerator*), 1965
PhonyNameDueToError.BT_LE_ADV_OPT_CONNECTABLE (*C enumerator*), 1962
PhonyNameDueToError.BT_LE_ADV_OPT_DIR_ADDR_RPA (*C enumerator*), 1964
PhonyNameDueToError.BT_LE_ADV_OPT_DIR_MODE_LOW_DUTY (*C enumerator*), 1963
PhonyNameDueToError.BT_LE_ADV_OPT_DISABLE_CHAN_37 (*C enumerator*), 1965
PhonyNameDueToError.BT_LE_ADV_OPT_DISABLE_CHAN_38 (*C enumerator*), 1965
PhonyNameDueToError.BT_LE_ADV_OPT_DISABLE_CHAN_39 (*C enumerator*), 1965
PhonyNameDueToError.BT_LE_ADV_OPT_EXT_ADV (*C enumerator*), 1964
PhonyNameDueToError.BT_LE_ADV_OPT_FILTER_CONN (*C enumerator*), 1964
PhonyNameDueToError.BT_LE_ADV_OPT_FILTER_SCAN_REQ (*C enumerator*), 1964
PhonyNameDueToError.BT_LE_ADV_OPT_FORCE_NAME_IN_AD (*C enumerator*), 1966
PhonyNameDueToError.BT_LE_ADV_OPT_NO_2M (*C enumerator*), 1964
PhonyNameDueToError.BT_LE_ADV_OPT_NONE (*C enumerator*), 1962
PhonyNameDueToError.BT_LE_ADV_OPT_NOTIFY_SCAN_REQ (*C enumerator*), 1964
PhonyNameDueToError.BT_LE_ADV_OPT_ONE_TIME (*C enumerator*), 1963
PhonyNameDueToError.BT_LE_ADV_OPT_SCANNABLE (*C enumerator*), 1964
PhonyNameDueToError.BT_LE_ADV_OPT_USE_IDENTITY (*C enumerator*), 1963
PhonyNameDueToError.BT_LE_ADV_OPT_USE_NAME (*C enumerator*), 1963
PhonyNameDueToError.BT_LE_ADV_OPT_USE_NRPA (*C enumerator*), 1966
PhonyNameDueToError.BT_LE_ADV_OPT_USE_TX_POWER (*C enumerator*), 1965
PhonyNameDueToError.BT_LE_PER_ADV_OPT_INCLUDE_ADI (*C enumerator*), 1966
PhonyNameDueToError.BT_LE_PER_ADV_OPT_NONE (*C enumerator*), 1966
PhonyNameDueToError.BT_LE_PER_ADV_OPT_USE_TX_POWER (*C enumerator*), 1966
PhonyNameDueToError.BT_LE_PER_ADV_SYNC_OPT_DONT_SYNC_AOA (*C enumerator*), 1967
PhonyNameDueToError.BT_LE_PER_ADV_SYNC_OPT_DONT_SYNC_AOD_1US (*C enumerator*), 1967
PhonyNameDueToError.BT_LE_PER_ADV_SYNC_OPT_DONT_SYNC_AOD_2US (*C enumerator*), 1967
PhonyNameDueToError.BT_LE_PER_ADV_SYNC_OPT_FILTER_DUPLICATE (*C enumerator*), 1967

PhonyNameDueToError.BT_LE_PER_ADV_SYNC_OPT_NONE (*C enumerator*), 1967
PhonyNameDueToError.BT_LE_PER_ADV_SYNC_OPT_REPORTING_INITIALLY_DISABLED (*C enumerator*), 1967
PhonyNameDueToError.BT_LE_PER_ADV_SYNC_OPT_SYNC_ONLY_CONST_TONE_EXT (*C enumerator*), 1967
PhonyNameDueToError.BT_LE_PER_ADV_SYNC_OPT_USE_PER_ADV_LIST (*C enumerator*), 1967
PhonyNameDueToError.BT_LE_PER_ADV_SYNC_TRANSFER_OPT_FILTER_DUPLICATES (*C enumerator*), 1968
PhonyNameDueToError.BT_LE_PER_ADV_SYNC_TRANSFER_OPT_NONE (*C enumerator*), 1967
PhonyNameDueToError.BT_LE_PER_ADV_SYNC_TRANSFER_OPT_REPORTING_INITIALLY_DISABLED (*C enumerator*), 1968
PhonyNameDueToError.BT_LE_PER_ADV_SYNC_TRANSFER_OPT_SYNC_NO_AOA (*C enumerator*), 1967
PhonyNameDueToError.BT_LE_PER_ADV_SYNC_TRANSFER_OPT_SYNC_NO_AOD_1US (*C enumerator*), 1967
PhonyNameDueToError.BT_LE_PER_ADV_SYNC_TRANSFER_OPT_SYNC_NO_AOD_2US (*C enumerator*), 1968
PhonyNameDueToError.BT_LE_PER_ADV_SYNC_TRANSFER_OPT_SYNC_ONLY_CTE (*C enumerator*), 1968
PhonyNameDueToError.BT_LE_SCAN_OPT_CODED (*C enumerator*), 1968
PhonyNameDueToError.BT_LE_SCAN_OPT_FILTER_ACCEPT_LIST (*C enumerator*), 1968
PhonyNameDueToError.BT_LE_SCAN_OPT_FILTER_DUPLICATE (*C enumerator*), 1968
PhonyNameDueToError.BT_LE_SCAN_OPT_NO_1M (*C enumerator*), 1968
PhonyNameDueToError.BT_LE_SCAN_OPT_NONE (*C enumerator*), 1968
PhonyNameDueToError.BT_LE_SCAN_TYPE_ACTIVE (*C enumerator*), 1969
PhonyNameDueToError.BT_LE_SCAN_TYPE_PASSIVE (*C enumerator*), 1968
PhonyNameDueToError.BT_MESH_OOB_AUTH_REQUIRED (*C enumerator*), 2242
PhonyNameDueToError.BT_MESH_PROV_AUTH_CMAC_AES128_AES_CCM (*C enumerator*), 2242
PhonyNameDueToError.BT_MESH_PROV_AUTH_HMAC_SHA256_AES_CCM (*C enumerator*), 2242
PhonyNameDueToError.BT_MESH_STATIC_OOB_AVAILABLE (*C enumerator*), 2242
PhonyNameDueToError.BT_OTS_METADATA_REQ_ALL (*C enumerator*), 1946
PhonyNameDueToError.BT_OTS_METADATA_REQ_CREATED (*C enumerator*), 1946
PhonyNameDueToError.BT_OTS_METADATA_REQ_ID (*C enumerator*), 1946
PhonyNameDueToError.BT_OTS_METADATA_REQ_MODIFIED (*C enumerator*), 1946
PhonyNameDueToError.BT_OTS_METADATA_REQ_NAME (*C enumerator*), 1946
PhonyNameDueToError.BT_OTS_METADATA_REQ_PROPS (*C enumerator*), 1946
PhonyNameDueToError.BT_OTS_METADATA_REQ_SIZE (*C enumerator*), 1946
PhonyNameDueToError.BT_OTS_METADATA_REQ_TYPE (*C enumerator*), 1946
PhonyNameDueToError.BT_OTS_OACP_FEAT_ABORT (*C enumerator*), 1945
PhonyNameDueToError.BT_OTS_OACP_FEAT_APPEND (*C enumerator*), 1945
PhonyNameDueToError.BT_OTS_OACP_FEAT_CHECKSUM (*C enumerator*), 1945
PhonyNameDueToError.BT_OTS_OACP_FEAT_CREATE (*C enumerator*), 1945
PhonyNameDueToError.BT_OTS_OACP_FEAT_DELETE (*C enumerator*), 1945
PhonyNameDueToError.BT_OTS_OACP_FEAT_EXECUTE (*C enumerator*), 1945
PhonyNameDueToError.BT_OTS_OACP_FEAT_PATCH (*C enumerator*), 1945
PhonyNameDueToError.BT_OTS_OACP_FEAT_READ (*C enumerator*), 1945
PhonyNameDueToError.BT_OTS_OACP_FEAT_TRUNCATE (*C enumerator*), 1945
PhonyNameDueToError.BT_OTS_OACP_FEAT_WRITE (*C enumerator*), 1945
PhonyNameDueToError.BT_OTS_OBJ_PROP_APPEND (*C enumerator*), 1944
PhonyNameDueToError.BT_OTS_OBJ_PROP_DELETE (*C enumerator*), 1944
PhonyNameDueToError.BT_OTS_OBJ_PROP_EXECUTE (*C enumerator*), 1944
PhonyNameDueToError.BT_OTS_OBJ_PROP_MARKED (*C enumerator*), 1944
PhonyNameDueToError.BT_OTS_OBJ_PROP_PATCH (*C enumerator*), 1944
PhonyNameDueToError.BT_OTS_OBJ_PROP_READ (*C enumerator*), 1944
PhonyNameDueToError.BT_OTS_OBJ_PROP_TRUNCATE (*C enumerator*), 1944
PhonyNameDueToError.BT_OTS_OBJ_PROP_WRITE (*C enumerator*), 1944
PhonyNameDueToError.BT_OTS_OLCP_FEAT_CLEAR (*C enumerator*), 1946
PhonyNameDueToError.BT_OTS_OLCP_FEAT_GO_TO (*C enumerator*), 1945
PhonyNameDueToError.BT_OTS_OLCP_FEAT_NUM_REQ (*C enumerator*), 1946
PhonyNameDueToError.BT_OTS_OLCP_FEAT_ORDER (*C enumerator*), 1945

PhonyNameDueToError.BT_QUIRK_NO_AUTO_DLE (*C enumerator*), 2346
PhonyNameDueToError.BT_QUIRK_NO_RESET (*C enumerator*), 2346
PhonyNameDueToError.BT_RFCOMM_CHAN_HFP_AG (*C enumerator*), 1708
PhonyNameDueToError.BT_RFCOMM_CHAN_HFP_HF (*C enumerator*), 1708
PhonyNameDueToError.BT_RFCOMM_CHAN_HSP_AG (*C enumerator*), 1708
PhonyNameDueToError.BT_RFCOMM_CHAN_HSP_HS (*C enumerator*), 1708
PhonyNameDueToError.BT_RFCOMM_CHAN_SPP (*C enumerator*), 1708
PhonyNameDueToError.BT_SDP_DISCOVER_UUID_CONTINUE (*C enumerator*), 1725
PhonyNameDueToError.BT_SDP_DISCOVER_UUID_STOP (*C enumerator*), 1725
PhonyNameDueToError.BT_UUID_TYPE_16 (*C enumerator*), 2428
PhonyNameDueToError.BT_UUID_TYPE_32 (*C enumerator*), 2428
PhonyNameDueToError.BT_UUID_TYPE_128 (*C enumerator*), 2428
PhonyNameDueToError.FS_EXT2 (*C enumerator*), 845
PhonyNameDueToError.FS_FATFS (*C enumerator*), 845
PhonyNameDueToError.FS_LITTLEFS (*C enumerator*), 845
PhonyNameDueToError.FS_TYPE_EXTERNAL_BASE (*C enumerator*), 845
PhonyNameDueToError.HID_REPORT_TYPE_FEATURE (*C enumerator*), 3063
PhonyNameDueToError.HID_REPORT_TYPE_INPUT (*C enumerator*), 3063
PhonyNameDueToError.HID_REPORT_TYPE_OUTPUT (*C enumerator*), 3063
PhonyNameDueToError.K_WORK_CANCELING (*C enumerator*), 356
PhonyNameDueToError.K_WORK_DELAYED (*C enumerator*), 357
PhonyNameDueToError.K_WORK_FLUSHING (*C enumerator*), 357
PhonyNameDueToError.K_WORK_QUEUED (*C enumerator*), 356
PhonyNameDueToError.K_WORK_RUNNING (*C enumerator*), 356
PhonyNameDueToError.PCIE_SCAN_CB_ALL (*C enumerator*), 3512
PhonyNameDueToError.PCIE_SCAN_RECURSIVE (*C enumerator*), 3512
pinctrl_apply_state (*C function*), 3717
pinctrl_apply_state_direct (*C function*), 3716
pinctrl_configure_pins (*C function*), 3716
pinctrl_dev_config (*C struct*), 3717
pinctrl_dev_config.reg (*C var*), 3717
pinctrl_dev_config.state_cnt (*C var*), 3717
pinctrl_dev_config.states (*C var*), 3717
PINCTRL_DT_DEFINE (*C macro*), 3715
PINCTRL_DT_DEV_CONFIG_DECLARE (*C macro*), 3715
PINCTRL_DT_DEV_CONFIG_GET (*C macro*), 3715
PINCTRL_DT_INST_DEFINE (*C macro*), 3715
PINCTRL_DT_INST_DEV_CONFIG_GET (*C macro*), 3715
PINCTRL_DT_STATE_INIT (*C macro*), 3718
PINCTRL_DT_STATE_PINS_DEFINE (*C macro*), 3718
pinctrl_lookup_state (*C function*), 3716
PINCTRL_REG_NONE (*C macro*), 3715
pinctrl_state (*C struct*), 3717
PINCTRL_STATE_DEFAULT (*C macro*), 3714
PINCTRL_STATE_PRIV_START (*C macro*), 3715
PINCTRL_STATE_SLEEP (*C macro*), 3714
pinctrl_state.id (*C var*), 3717
pinctrl_state.pin_cnt (*C var*), 3717
pinctrl_state.pins (*C var*), 3717
pinctrl_update_states (*C function*), 3719
PM_ALL_SUBSTATES (*C macro*), 1065
pm_device (*C struct*), 1078
pm_device_action (*C enum*), 1073
pm_device_action_cb_t (*C type*), 1072
pm_device_action_failed_cb_t (*C type*), 1072
pm_device_action_run (*C function*), 1074
pm_device_action.PM_DEVICE_ACTION_RESUME (*C enumerator*), 1073
pm_device_action.PM_DEVICE_ACTION_SUSPEND (*C enumerator*), 1073

pm_device_action.PM_DEVICE_ACTION_TURN_OFF (*C enumerator*), 1073
pm_device_action.PM_DEVICE_ACTION_TURN_ON (*C enumerator*), 1073
pm_device_base (*C struct*), 1078
pm_device_base.action_cb (*C var*), 1078
pm_device_base.flags (*C var*), 1078
pm_device_base.state (*C var*), 1078
pm_device_base.usage (*C var*), 1078
pm_device_busy_clear (*C function*), 1075
pm_device_busy_set (*C function*), 1075
pm_device_children_action_run (*C function*), 1074
PM_DEVICE_DEFINE (*C macro*), 1070
pm_device_driver_init (*C function*), 1077
PM_DEVICE_DT_DEFINE (*C macro*), 1071
PM_DEVICE_DT_GET (*C macro*), 1072
PM_DEVICE_DT_INST_DEFINE (*C macro*), 1071
PM_DEVICE_DT_INST_GET (*C macro*), 1072
PM_DEVICE_GET (*C macro*), 1071
pm_device_init_off (*C function*), 1075
pm_device_init_suspended (*C function*), 1075
pm_device_is_any_busy (*C function*), 1075
pm_device_is_busy (*C function*), 1076
pm_device_is_powered (*C function*), 1077
pm_device_isr (*C struct*), 1078
PM_DEVICE_ISR_SAFE (*C macro*), 1070
pm_device_isr.base (*C var*), 1079
pm_device_isr.lock (*C var*), 1079
pm_device_on_power_domain (*C function*), 1076
pm_device_power_domain_add (*C function*), 1076
pm_device_power_domain_remove (*C function*), 1077
pm_device_runtime_auto_enable (*C function*), 1079
pm_device_runtime_disable (*C function*), 1080
pm_device_runtime_enable (*C function*), 1079
pm_device_runtime_get (*C function*), 1080
pm_device_runtime_is_enabled (*C function*), 1082
pm_device_runtime_put (*C function*), 1080
pm_device_runtime_put_async (*C function*), 1081
pm_device_runtime_usage (*C function*), 1082
pm_device_state (*C enum*), 1072
pm_device_state_get (*C function*), 1074
pm_device_state_str (*C function*), 1074
pm_device_state.PM_DEVICE_STATE_ACTIVE (*C enumerator*), 1073
pm_device_state.PM_DEVICE_STATE_OFF (*C enumerator*), 1073
pm_device_state.PM_DEVICE_STATE_SUSPENDED (*C enumerator*), 1073
pm_device_state.PM_DEVICE_STATE_SUSPENDING (*C enumerator*), 1073
pm_device_wakeup_enable (*C function*), 1076
pm_device_wakeup_is_capable (*C function*), 1076
pm_device_wakeup_is_enabled (*C function*), 1076
pm_device.base (*C var*), 1078
pm_device.dev (*C var*), 1078
pm_device.event (*C var*), 1078
pm_device.lock (*C var*), 1078
pm_device.work (*C var*), 1078
pm_notifier (*C struct*), 1059
pm_notifier_register (*C function*), 1058
pm_notifier_unregister (*C function*), 1059
pm_notifier.state_entry (*C var*), 1060
pm_notifier.state_exit (*C var*), 1060
pm_policy_device_power_lock_get (*C function*), 1068

pm_policy_device_power_lock_put (*C function*), 1069
pm_policy_event (*C struct*), 1069
pm_policy_event_register (*C function*), 1067
pm_policy_event_unregister (*C function*), 1068
pm_policy_event_update (*C function*), 1068
pm_policy_latency_changed_cb_t (*C type*), 1065
pm_policy_latency_changed_subscribe (*C function*), 1067
pm_policy_latency_changed_unsubscribe (*C function*), 1067
pm_policy_latency_request (*C struct*), 1069
pm_policy_latency_request_add (*C function*), 1067
pm_policy_latency_request_remove (*C function*), 1067
pm_policy_latency_request_update (*C function*), 1067
pm_policy_latency_subscription (*C struct*), 1069
pm_policy_state_lock_get (*C function*), 1066
pm_policy_state_lock_is_active (*C function*), 1066
pm_policy_state_lock_put (*C function*), 1066
pm_state (*C enum*), 1062
pm_state_constraint (*C struct*), 1065
pm_state_constraint.state (*C var*), 1065
pm_state_constraint.substate_id (*C var*), 1065
pm_state_cpu_get_all (*C function*), 1064
PM_STATE_DT_INIT (*C macro*), 1060
pm_state_exit_post_ops (*C function*), 1070
pm_state_force (*C function*), 1058
pm_state_info (*C struct*), 1064
PM_STATE_INFO_DT_INIT (*C macro*), 1060
PM_STATE_INFO_LIST_FROM_DT_CPU (*C macro*), 1060
pm_state_info.exit_latency_us (*C var*), 1064
pm_state_info.min_residency_us (*C var*), 1064
pm_state_info.pm_device_disabled (*C var*), 1064
pm_state_info.substate_id (*C var*), 1064
PM_STATE_LIST_FROM_DT_CPU (*C macro*), 1061
pm_state_next_get (*C function*), 1059
pm_state_set (*C function*), 1070
pm_state.PM_STATE_ACTIVE (*C enumerator*), 1062
pm_state.PM_STATE_COUNT (*C enumerator*), 1063
pm_state.PM_STATE_RUNTIME_IDLE (*C enumerator*), 1062
pm_state.PM_STATE_SOFT_OFF (*C enumerator*), 1063
pm_state.PM_STATE_STANDBY (*C enumerator*), 1062
pm_state.PM_STATE_SUSPEND_TO_DISK (*C enumerator*), 1063
pm_state.PM_STATE_SUSPEND_TO_IDLE (*C enumerator*), 1062
pm_state.PM_STATE_SUSPEND_TO_RAM (*C enumerator*), 1063
pm_system_resume (*C function*), 1059
POINTER_TO_INT (*C macro*), 679
POINTER_TO_UINT (*C macro*), 679
policy_cb_change_src_caps_t (*C type*), 3096
policy_cb_check_sink_request_t (*C type*), 3095
policy_cb_check_t (*C type*), 3094
policy_cb_get_rdo_t (*C type*), 3094
policy_cb_get_snk_cap_t (*C type*), 3093
policy_cb_get_src_caps_t (*C type*), 3095
policy_cb_get_src_rp_t (*C type*), 3096
policy_cb_is_ps_ready_t (*C type*), 3095
policy_cb_is_snk_at_default_t (*C type*), 3094
policy_cb_notify_t (*C type*), 3094
policy_cb_present_contract_is_valid_t (*C type*), 3095
policy_cb_set_port_partner_snk_cap_t (*C type*), 3096
policy_cb_set_src_cap_t (*C type*), 3094

policy_cb_src_en_t (*C type*), 3096
policy_cb_wait_notify_t (*C type*), 3094
poll (*C function*), 2489
POLLERR (*C macro*), 2489
pollfd (*C macro*), 2489
POLLHUP (*C macro*), 2490
POLLIN (*C macro*), 2489
POLLNVAL (*C macro*), 2490
POLLOUT (*C macro*), 2489
popen_ignore_int() (*runners.core.ZephyrBinaryRunner method*), 200
power domain, **3936**
power gating, **3936**
PRIkbdrow (*C macro*), 878
printfcb (*C function*), 870
ps2_callback_t (*C type*), 3523
ps2_config (*C function*), 3524
ps2_disable_callback (*C function*), 3524
ps2_enable_callback (*C function*), 3524
ps2_read (*C function*), 3524
ps2_write (*C function*), 3524
PTP_MAJOR_VERSION (*C macro*), 2882
PTP_MINOR_VERSION (*C macro*), 2882
PTP_VERSION (*C macro*), 2882
pwm_capture_callback_handler_t (*C type*), 3532
pwm_capture_cycles (*C function*), 3537
PWM_CAPTURE_MODE_CONTINUOUS (*C macro*), 3525
PWM_CAPTURE_MODE_SINGLE (*C macro*), 3525
pwm_capture_nsec (*C function*), 3538
PWM_CAPTURE_TYPE_BOTH (*C macro*), 3525
PWM_CAPTURE_TYPE_PERIOD (*C macro*), 3525
PWM_CAPTURE_TYPE_PULSE (*C macro*), 3525
pwm_capture_usec (*C function*), 3537
pwm_configure_capture (*C function*), 3535
pwm_cycles_to_nsec (*C function*), 3535
pwm_cycles_to_usec (*C function*), 3535
pwm_disable_capture (*C function*), 3536
pwm_dt_spec (*C struct*), 3539
PWM_DT_SPEC_GET (*C macro*), 3530
PWM_DT_SPEC_GET_BY_IDX (*C macro*), 3528
PWM_DT_SPEC_GET_BY_IDX_OR (*C macro*), 3529
PWM_DT_SPEC_GET_BY_NAME (*C macro*), 3526
PWM_DT_SPEC_GET_BY_NAME_OR (*C macro*), 3527
PWM_DT_SPEC_GET_OR (*C macro*), 3531
PWM_DT_SPEC_INST_GET (*C macro*), 3530
PWM_DT_SPEC_INST_GET_BY_IDX (*C macro*), 3529
PWM_DT_SPEC_INST_GET_BY_IDX_OR (*C macro*), 3529
PWM_DT_SPEC_INST_GET_BY_NAME (*C macro*), 3527
PWM_DT_SPEC_INST_GET_BY_NAME_OR (*C macro*), 3528
PWM_DT_SPEC_INST_GET_OR (*C macro*), 3531
pwm_dt_spec.channel (*C var*), 3539
pwm_dt_spec.dev (*C var*), 3539
pwm_dt_spec.flags (*C var*), 3540
pwm_dt_spec.period (*C var*), 3540
pwm_enable_capture (*C function*), 3536
pwm_flags_t (*C type*), 3531
pwm_get_cycles_per_sec (*C function*), 3533
PWM_HZ (*C macro*), 3526
pwm_is_ready_dt (*C function*), 3538

PWM_KHZ (*C macro*), 3526
PWM_MSEC (*C macro*), 3526
PWM_NSEC (*C macro*), 3526
PWM_POLARITY_INVERTED (*C macro*), 3526
PWM_POLARITY_NORMAL (*C macro*), 3526
PWM_SEC (*C macro*), 3526
pwm_set (*C function*), 3533
pwm_set_cycles (*C function*), 3532
pwm_set_dt (*C function*), 3534
pwm_set_pulse_dt (*C function*), 3534
PWM_USEC (*C macro*), 3526

Q

q7_t (*C type*), 841
q15_t (*C type*), 841
q31_t (*C type*), 841
q63_t (*C type*), 841
QEMU_EXTRA_FLAGS, 48

R

rb_contains (*C function*), 633
RB_FOR_EACH (*C macro*), 632
RB_FOR_EACH_CONTAINER (*C macro*), 633
rb_get_max (*C function*), 633
rb_get_min (*C function*), 633
rb_insert (*C function*), 633
rb_lessthan_t (*C type*), 633
rb_remove (*C function*), 633
rb_visit_t (*C type*), 633
rb_walk (*C function*), 633
rbnode (*C struct*), 634
rbtree (*C struct*), 634
rbtree.lessthan_fn (*C var*), 634
rbtree.root (*C var*), 634
readline() (*twister_harness.DeviceAdapter method*), 268
readlines() (*twister_harness.DeviceAdapter method*), 268
readlines_until() (*twister_harness.DeviceAdapter method*), 268
recv (*C function*), 2488
recvfrom (*C function*), 2488
recvmsg (*C function*), 2489
regulator_count_current_limits (*C function*), 3552
regulator_count_voltages (*C function*), 3550
regulator_disable (*C function*), 3550
regulator_dvs_state_t (*C type*), 3549
regulator_enable (*C function*), 3550
regulator_error_flags_t (*C type*), 3549
REGULATOR_ERROR_OVER_CURRENT (*C macro*), 3549
REGULATOR_ERROR_OVER_TEMP (*C macro*), 3549
REGULATOR_ERROR_OVER_VOLTAGE (*C macro*), 3549
regulator_get_active_discharge (*C function*), 3554
regulator_get_current_limit (*C function*), 3553
regulator_get_error_flags (*C function*), 3554
regulator_get_mode (*C function*), 3553
regulator_get_voltage (*C function*), 3551
regulator_is_enabled (*C function*), 3550
regulator_is_supported_voltage (*C function*), 3551
regulator_list_current_limit (*C function*), 3552
regulator_list_voltage (*C function*), 3550

regulator_mode_t (*C type*), 3549
regulator_set_active_discharge (*C function*), 3553
regulator_set_current_limit (*C function*), 3552
regulator_set_mode (*C function*), 3553
regulator_set_voltage (*C function*), 3551
require() (*runners.core.ZephyrBinaryRunner static method*), 200
RESET_BROWNOUT (*C macro*), 3384
RESET_CLOCK (*C macro*), 3385
RESET_CPU_LOCKUP (*C macro*), 3385
RESET_DEBUG (*C macro*), 3384
reset_dt_spec (*C struct*), 3559
RESET_DT_SPEC_GET (*C macro*), 3556
RESET_DT_SPEC_GET_BY_IDX (*C macro*), 3555
RESET_DT_SPEC_GET_BY_IDX_OR (*C macro*), 3555
RESET_DT_SPEC_GET_OR (*C macro*), 3556
RESET_DT_SPEC_INST_GET (*C macro*), 3557
RESET_DT_SPEC_INST_GET_BY_IDX (*C macro*), 3556
RESET_DT_SPEC_INST_GET_BY_IDX_OR (*C macro*), 3556
RESET_DT_SPEC_INST_GET_OR (*C macro*), 3557
reset_dt_spec.dev (*C var*), 3559
reset_dt_spec.id (*C var*), 3559
RESET_HARDWARE (*C macro*), 3385
reset_line_assert (*C function*), 3558
reset_line_assert_dt (*C function*), 3558
reset_line_deassert (*C function*), 3558
reset_line_deassert_dt (*C function*), 3558
reset_line_toggle (*C function*), 3559
reset_line_toggle_dt (*C function*), 3559
RESET_LOW_POWER_WAKE (*C macro*), 3385
RESET_PARITY (*C macro*), 3385
RESET_PIN (*C macro*), 3384
RESET_PLL (*C macro*), 3385
RESET_POR (*C macro*), 3384
RESET_SECURITY (*C macro*), 3384
RESET_SOFTWARE (*C macro*), 3384
reset_status (*C function*), 3557
reset_status_dt (*C function*), 3557
RESET_TEMPERATURE (*C macro*), 3385
RESET_USER (*C macro*), 3385
RESET_WATCHDOG (*C macro*), 3384
RESULT_CONNECTION_LOST (*C macro*), 2825
RESULT_DEFAULT (*C macro*), 2825
RESULT_INTEGRITY_FAILED (*C macro*), 2825
RESULT_INVALID_URI (*C macro*), 2825
RESULT_NO_STORAGE (*C macro*), 2825
RESULT_OUT_OF_MEM (*C macro*), 2825
RESULT_SUCCESS (*C macro*), 2825
RESULT_UNSUP_FW (*C macro*), 2825
RESULT_UNSUP_PROTO (*C macro*), 2825
RESULT_UPDATE_FAILED (*C macro*), 2825
retained_mem_clear (*C function*), 3561
retained_mem_clear_api (*C type*), 3560
retained_mem_driver_api (*C struct*), 3561
retained_mem_read (*C function*), 3561
retained_mem_read_api (*C type*), 3560
retained_mem_size (*C function*), 3561
retained_mem_size_api (*C type*), 3560
retained_mem_write (*C function*), 3561

retained_mem_write_api (*C type*), 3560
retention_api (*C struct*), 1241
retention_clear (*C function*), 1241
retention_clear_api (*C type*), 1240
retention_is_valid (*C function*), 1240
retention_is_valid_api (*C type*), 1240
retention_read (*C function*), 1241
retention_read_api (*C type*), 1240
retention_size (*C function*), 1240
retention_size_api (*C type*), 1240
retention_write (*C function*), 1241
retention_write_api (*C type*), 1240
RETURN_HANDLED_CONTEXT (*C macro*), 239
REVERSE_ARGS (*C macro*), 694
ring_buf (*C struct*), 646
ring_buf_capacity_get (*C function*), 641
RING_BUF_DECLARE (*C macro*), 639
ring_buf_get (*C function*), 644
ring_buf_get_claim (*C function*), 643
ring_buf_get_finish (*C function*), 644
ring_buf_init (*C function*), 640
ring_buf_internal_reset (*C function*), 640
ring_buf_is_empty (*C function*), 641
RING_BUF_ITEM_DECLARE (*C macro*), 640
RING_BUF_ITEM_DECLARE_POW2 (*C macro*), 640
RING_BUF_ITEM_DECLARE_SIZE (*C macro*), 640
ring_buf_item_get (*C function*), 646
ring_buf_item_init (*C function*), 641
ring_buf_item_put (*C function*), 645
RING_BUF_ITEM_SIZEOF (*C macro*), 640
ring_buf_item_space_get (*C function*), 641
ring_buf_peek (*C function*), 645
ring_buf_put (*C function*), 643
ring_buf_put_claim (*C function*), 642
ring_buf_put_finish (*C function*), 642
ring_buf_reset (*C function*), 641
ring_buf_size_get (*C function*), 642
ring_buf_space_get (*C function*), 641
ROUND_DOWN (*C macro*), 682
ROUND_UP (*C macro*), 682
rtc_alarm_callback (*C type*), 3546
rtc_alarm_get_supported_fields (*C function*), 3541
rtc_alarm_get_time (*C function*), 3542
rtc_alarm_is_pending (*C function*), 3542
rtc_alarm_set_callback (*C function*), 3543
rtc_alarm_set_time (*C function*), 3541
RTC_ALARM_TIME_MASK_HOUR (*C macro*), 3545
RTC_ALARM_TIME_MASK_MINUTE (*C macro*), 3545
RTC_ALARM_TIME_MASK_MONTH (*C macro*), 3545
RTC_ALARM_TIME_MASK_MONTHDAY (*C macro*), 3545
RTC_ALARM_TIME_MASK_NSEC (*C macro*), 3545
RTC_ALARM_TIME_MASK_SECOND (*C macro*), 3545
RTC_ALARM_TIME_MASK_WEEKDAY (*C macro*), 3545
RTC_ALARM_TIME_MASK_YEAR (*C macro*), 3545
RTC_ALARM_TIME_MASK_YEARDAY (*C macro*), 3545
rtc_calibration_from_frequency (*C function*), 3545
rtc_get_calibration (*C function*), 3544
rtc_get_time (*C function*), 3546

rtc_set_calibration (*C function*), 3544
rtc_set_time (*C function*), 3546
rtc_time (*C struct*), 3546
rtc_time_to_tm (*C function*), 3545
rtc_time.tm_hour (*C var*), 3547
rtc_time.tm_isdst (*C var*), 3547
rtc_time.tm_mday (*C var*), 3547
rtc_time.tm_min (*C var*), 3547
rtc_time.tm_mon (*C var*), 3547
rtc_time.tm_nsec (*C var*), 3547
rtc_time.tm_sec (*C var*), 3547
rtc_time.tm_wday (*C var*), 3547
rtc_time.tm_yday (*C var*), 3547
rtc_time.tm_year (*C var*), 3547
rtc_update_callback (*C type*), 3546
rtc_update_set_callback (*C function*), 3543
rtio (*C struct*), 1256
rtio_access_grant (*C function*), 1253
rtio_block_pool_alloc (*C function*), 1250
rtio_block_pool_free (*C function*), 1250
RTIO_BMEM (*C macro*), 1248
rtio_callback_t (*C type*), 1248
rtio_chain_next (*C function*), 1250
rtio_cqe (*C struct*), 1256
rtio_cqe_acquire (*C function*), 1251
rtio_cqe_compute_flags (*C function*), 1252
rtio_cqe_consume (*C function*), 1251
rtio_cqe_consume_block (*C function*), 1251
rtio_cqe_copy_out (*C function*), 1254
rtio_cqe_get_mempool_buffer (*C function*), 1252
rtio_cqe_pool (*C struct*), 1256
rtio_cqe_pool_alloc (*C function*), 1250
rtio_cqe_pool_free (*C function*), 1250
rtio_cqe_produce (*C function*), 1251
rtio_cqe_release (*C function*), 1251
rtio_cqe_submit (*C function*), 1252
rtio_cqe.flags (*C var*), 1256
rtio_cqe.result (*C var*), 1256
rtio_cqe.userdata (*C var*), 1256
RTIO_DEFINE (*C macro*), 1248
RTIO_DEFINE_WITH_MEMPOOL (*C macro*), 1248
RTIO_DMEM (*C macro*), 1248
rtio_executor_err (*C function*), 1252
rtio_executor_ok (*C function*), 1252
rtio_executor_submit (*C function*), 1252
rtio_iodev (*C struct*), 1257
rtio_iodev_api (*C struct*), 1257
rtio_iodev_api.submit (*C var*), 1257
RTIO_IODEV_DEFINE (*C macro*), 1247
RTIO_IODEV_I2C_10_BITS (*C macro*), 1247
RTIO_IODEV_I2C_RESTART (*C macro*), 1247
RTIO_IODEV_I2C_STOP (*C macro*), 1247
rtio_iodev_sqe (*C struct*), 1256
rtio_iodev_sqe_err (*C function*), 1252
rtio_iodev_sqe_next (*C function*), 1250
rtio_iodev_sqe_ok (*C function*), 1252
rtio_mempool_block_size (*C function*), 1249
RTIO_OP_CALLBACK (*C macro*), 1247

RTIO_OP_I2C_CONFIGURE (*C macro*), 1247
RTIO_OP_I2C_RECOVER (*C macro*), 1247
RTIO_OP_NOP (*C macro*), 1247
RTIO_OP_RX (*C macro*), 1247
RTIO_OP_TINY_TX (*C macro*), 1247
RTIO_OP_TX (*C macro*), 1247
RTIO_OP_TXRX (*C macro*), 1247
rtio_partition (*C var*), 1255
rtio_release_buffer (*C function*), 1253
rtio_sqe (*C struct*), 1255
rtio_sqe_acquirable (*C function*), 1250
rtio_sqe_acquire (*C function*), 1251
rtio_sqe_cancel (*C function*), 1253
rtio_sqe_copy_in (*C function*), 1254
rtio_sqe_copy_in_get_handles (*C function*), 1254
rtio_sqe_drop_all (*C function*), 1251
rtio_sqe_pool (*C struct*), 1256
rtio_sqe_pool_alloc (*C function*), 1250
rtio_sqe_pool_free (*C function*), 1250
rtio_sqe_prep_callback (*C function*), 1249
rtio_sqe_prep_nop (*C function*), 1249
rtio_sqe_prep_read (*C function*), 1249
rtio_sqe_prep_read_multishot (*C function*), 1249
rtio_sqe_prep_read_with_pool (*C function*), 1249
rtio_sqe_prep_tiny_write (*C function*), 1249
rtio_sqe_prep_transceive (*C function*), 1250
rtio_sqe_prep_write (*C function*), 1249
rtio_sqe_rx_buf (*C function*), 1253
rtio_sqe.arg0 (*C var*), 1256
rtio_sqe.buf (*C var*), 1256
rtio_sqe.buf_len (*C var*), 1255
rtio_sqe.flags (*C var*), 1255
rtio_sqe.i2c_config (*C var*), 1256
rtio_sqe.iodev (*C var*), 1255
rtio_sqe.iodev_flags (*C var*), 1255
rtio_sqe.op (*C var*), 1255
rtio_sqe.prio (*C var*), 1255
rtio_sqe.tiny_buf (*C var*), 1256
rtio_sqe.tiny_buf_len (*C var*), 1256
rtio_sqe.userdata (*C var*), 1255
rtio_submit (*C function*), 1255
rtio_txn_next (*C function*), 1250
run() (*runners.core.ZephyrBinaryRunner method*), 201
run_client() (*runners.core.ZephyrBinaryRunner method*), 201
run_server_and_client() (*runners.core.ZephyrBinaryRunner method*), 201
RunnerCaps (*class in runners.core*), 196
RunnerConfig (*class in runners.core*), 197
runners.core
    module, 195

S

sa_family_t (*C type*), 2510
SAME_TYPE (*C macro*), 681
sbs_gauge_device_chemistry (*C struct*), 3349
SBS_GAUGE_DEVICE_CHEMISTRY_MAX_SIZE (*C macro*), 3343
sbs_gauge_device_name (*C struct*), 3349
SBS_GAUGE_DEVICE_NAME_MAX_SIZE (*C macro*), 3343
sbs_gauge_manufacturer_name (*C struct*), 3349

SBS_GAUGE_MANUFACTURER_NAME_MAX_SIZE (*C macro*), 3343
scan_result_cb_t (*C type*), 2717
SCM_TXTIME (*C macro*), 2495
sd_voltage (*C enum*), 3564
sd_voltage.SD_VOL_1_2_V (*C enumerator*), 3565
sd_voltage.SD_VOL_1_8_V (*C enumerator*), 3565
sd_voltage.SD_VOL_3_0_V (*C enumerator*), 3565
sd_voltage.SD_VOL_3_3_V (*C enumerator*), 3565
sdhc_bus_mode (*C enum*), 3563
sdhc_bus_mode.SDHC_BUSMODE_OPENDRAIN (*C enumerator*), 3563
sdhc_bus_mode.SDHC_BUSMODE_PUSHPULL (*C enumerator*), 3563
sdhc_bus_width (*C enum*), 3563
sdhc_bus_width.SDHC_BUS_WIDTH1BIT (*C enumerator*), 3564
sdhc_bus_width.SDHC_BUS_WIDTH4BIT (*C enumerator*), 3564
sdhc_bus_width.SDHC_BUS_WIDTH8BIT (*C enumerator*), 3564
sdhc_card_busy (*C function*), 3567
sdhc_card_present (*C function*), 3566
sdhc_command (*C struct*), 3568
sdhc_command.arg (*C var*), 3568
sdhc_command.opcode (*C var*), 3568
sdhc_command.response (*C var*), 3568
sdhc_command.response_type (*C var*), 3568
sdhc_command.retries (*C var*), 3568
sdhc_command.timeout_ms (*C var*), 3568
sdhc_data (*C struct*), 3568
sdhc_data.block_addr (*C var*), 3569
sdhc_data.block_size (*C var*), 3569
sdhc_data.blocks (*C var*), 3569
sdhc_data.bytes_xfered (*C var*), 3569
sdhc_data.data (*C var*), 3569
sdhc_data.timeout_ms (*C var*), 3569
sdhc_disable_interrupt (*C function*), 3568
sdhc_driver_api (*C struct*), 3572
sdhc_enable_interrupt (*C function*), 3567
sdhc_execute_tuning (*C function*), 3566
sdhc_get_host_props (*C function*), 3567
sdhc_host_caps (*C struct*), 3569
sdhc_host_caps.address_64_bit_support_v3 (*C var*), 3570
sdhc_host_caps.address_64_bit_support_v4 (*C var*), 3570
sdhc_host_caps.adma3_support (*C var*), 3571
sdhc_host_caps.adma_2_support (*C var*), 3569
sdhc_host_caps.bus_4_bit_support (*C var*), 3569
sdhc_host_caps.bus_8_bit_support (*C var*), 3569
sdhc_host_caps.clk_multiplier (*C var*), 3571
sdhc_host_caps.ddr50_support (*C var*), 3570
sdhc_host_caps.drv_type_a_support (*C var*), 3570
sdhc_host_caps.drv_type_c_support (*C var*), 3570
sdhc_host_caps.drv_type_d_support (*C var*), 3570
sdhc_host_caps.high_spd_support (*C var*), 3569
sdhc_host_caps.hs200_support (*C var*), 3571
sdhc_host_caps.hs400_support (*C var*), 3571
sdhc_host_caps.max_blk_len (*C var*), 3569
sdhc_host_caps.retune_timer_count (*C var*), 3571
sdhc_host_caps.retuning_mode (*C var*), 3571
sdhc_host_caps.sd_base_clk (*C var*), 3569
sdhc_host_caps.sdio_async_interrupt_support (*C var*), 3570
sdhc_host_caps.sdma_support (*C var*), 3570
sdhc_host_caps.sdr50_needs_tuning (*C var*), 3571

sdhc_host_caps.sdr50_support (*C var*), 3570
sdhc_host_caps.sdr104_support (*C var*), 3570
sdhc_host_caps.slot_type (*C var*), 3570
sdhc_host_caps.suspend_res_support (*C var*), 3570
sdhc_host_caps.timeout_clk_freq (*C var*), 3569
sdhc_host_caps.timeout_clk_unit (*C var*), 3569
sdhc_host_caps.uhs_2_support (*C var*), 3570
sdhc_host_caps.vdd2_180_support (*C var*), 3571
sdhc_host_caps.vol_180_support (*C var*), 3570
sdhc_host_caps.vol_300_support (*C var*), 3570
sdhc_host_caps.vol_330_support (*C var*), 3570
sdhc_host_props (*C struct*), 3572
sdhc_host_props.f_max (*C var*), 3572
sdhc_host_props.f_min (*C var*), 3572
sdhc_host_props.host_caps (*C var*), 3572
sdhc_host_props.is_spi (*C var*), 3572
sdhc_host_props.max_current_180 (*C var*), 3572
sdhc_host_props.max_current_300 (*C var*), 3572
sdhc_host_props.max_current_330 (*C var*), 3572
sdhc_host_props.power_delay (*C var*), 3572
sdhc_hw_reset (*C function*), 3565
sdhc_interrupt_cb_t (*C type*), 3563
sdhc_interrupt_source (*C enum*), 3565
sdhc_interrupt_source.SDHC_INT_INSERTED (*C enumerator*), 3565
sdhc_interrupt_source.SDHC_INT_REMOVED (*C enumerator*), 3565
sdhc_interrupt_source.SDHC_INT_SDIO (*C enumerator*), 3565
sdhc_io (*C struct*), 3571
sdhc_io.bus_mode (*C var*), 3571
sdhc_io.bus_width (*C var*), 3571
sdhc_io.clock (*C var*), 3571
sdhc_io.driver_type (*C var*), 3571
sdhc_io.power_mode (*C var*), 3571
sdhc_io.signal_voltage (*C var*), 3572
sdhc_io.timing (*C var*), 3571
SDHC_NATIVE_RESPONSE_MASK (*C macro*), 3563
sdhc_power (*C enum*), 3563
sdhc_power.SDHC_POWER_OFF (*C enumerator*), 3563
sdhc_power.SDHC_POWER_ON (*C enumerator*), 3563
sdhc_request (*C function*), 3565
sdhc_set_io (*C function*), 3566
SDHC_SPI_RESPONSE_TYPE_MASK (*C macro*), 3563
SDHC_TIMEOUT_FOREVER (*C macro*), 3562
sdhc_timing_mode (*C enum*), 3564
sdhc_timing_mode.SDHC_TIMING_DDR50 (*C enumerator*), 3564
sdhc_timing_mode.SDHC_TIMING_DDR52 (*C enumerator*), 3564
sdhc_timing_mode.SDHC_TIMING_HS (*C enumerator*), 3564
sdhc_timing_mode.SDHC_TIMING_HS200 (*C enumerator*), 3564
sdhc_timing_mode.SDHC_TIMING_HS400 (*C enumerator*), 3564
sdhc_timing_mode.SDHC_TIMING_LEGACY (*C enumerator*), 3564
sdhc_timing_mode.SDHC_TIMING_SDR12 (*C enumerator*), 3564
sdhc_timing_mode.SDHC_TIMING_SDR25 (*C enumerator*), 3564
sdhc_timing_mode.SDHC_TIMING_SDR50 (*C enumerator*), 3564
sdhc_timing_mode.SDHC_TIMING_SDR104 (*C enumerator*), 3564
SEARCH_LEN_MAX (*C macro*), 1842
SEARCH_LEN_MIN (*C macro*), 1842
SEARCH_PARAM_MAX (*C macro*), 1842
SEARCH_SCI_LEN_MIN (*C macro*), 1842
SEC_PER_MIN (*C macro*), 476

sec_tag_t (*C type*), 2504
semihost_close (*C function*), 3126
semihost_exec (*C function*), 3126
semihost_flen (*C function*), 3127
semihost_instr (*C enum*), 3124
semihost_instr.SEMIHOST_CLOCK (*C enumerator*), 3125
semihost_instr.SEMIHOST_CLOSE (*C enumerator*), 3124
semihost_instr.SEMIHOST_ELAPSED (*C enumerator*), 3125
semihost_instr.SEMIHOST_ERRNO (*C enumerator*), 3125
semihost_instr.SEMIHOST_FLEN (*C enumerator*), 3124
semihost_instr.SEMIHOST_GET_CMDLINE (*C enumerator*), 3125
semihost_instr.SEMIHOST_HEAPINFO (*C enumerator*), 3125
semihost_instr.SEMIHOST_ISERROR (*C enumerator*), 3125
semihost_instr.SEMIHOST_ISTTY (*C enumerator*), 3124
semihost_instr.SEMIHOST_OPEN (*C enumerator*), 3124
semihost_instr.SEMIHOST_READ (*C enumerator*), 3124
semihost_instr.SEMIHOST_READC (*C enumerator*), 3125
semihost_instr.SEMIHOST_REMOVE (*C enumerator*), 3124
semihost_instr.SEMIHOST_RENAME (*C enumerator*), 3124
semihost_instr.SEMIHOST_SEEK (*C enumerator*), 3124
semihost_instr.SEMIHOST_SYSTEM (*C enumerator*), 3125
semihost_instr.SEMIHOST_TICKFREQ (*C enumerator*), 3125
semihost_instr.SEMIHOST_TIME (*C enumerator*), 3125
semihost_instr.SEMIHOST_TMPNAM (*C enumerator*), 3124
semihost_instr.SEMIHOST_WRITE (*C enumerator*), 3124
semihost_instr.SEMIHOST_WRITE0 (*C enumerator*), 3125
semihost_instr.SEMIHOST_WRITEC (*C enumerator*), 3124
semihost_open (*C function*), 3126
semihost_open_mode (*C enum*), 3125
semihost_open_mode.SEMIHOST_OPEN_A (*C enumerator*), 3126
semihost_open_mode.SEMIHOST_OPEN_A_PLUS (*C enumerator*), 3126
semihost_open_mode.SEMIHOST_OPEN_AB (*C enumerator*), 3126
semihost_open_mode.SEMIHOST_OPEN_AB_PLUS (*C enumerator*), 3126
semihost_open_mode.SEMIHOST_OPEN_R (*C enumerator*), 3125
semihost_open_mode.SEMIHOST_OPEN_R_PLUS (*C enumerator*), 3125
semihost_open_mode.SEMIHOST_OPEN_RB (*C enumerator*), 3125
semihost_open_mode.SEMIHOST_OPEN_RB_PLUS (*C enumerator*), 3125
semihost_open_mode.SEMIHOST_OPEN_W (*C enumerator*), 3125
semihost_open_mode.SEMIHOST_OPEN_W_PLUS (*C enumerator*), 3125
semihost_open_mode.SEMIHOST_OPEN_WB (*C enumerator*), 3125
semihost_open_mode.SEMIHOST_OPEN_WB_PLUS (*C enumerator*), 3126
semihost_poll_in (*C function*), 3126
semihost_poll_out (*C function*), 3126
semihost_read (*C function*), 3127
semihost_seek (*C function*), 3127
semihost_write (*C function*), 3127
send (*C function*), 2488
sendmsg (*C function*), 2488
sendto (*C function*), 2488
sensing_callback_list (*C struct*), 1216
sensing_callback_list.context (*C var*), 1216
sensing_callback_list.on_data_event (*C var*), 1216
sensing_close_sensor (*C function*), 1215
sensing_data_event_t (*C type*), 1213
sensing_get_config (*C function*), 1215
sensing_get_sensor_info (*C function*), 1215
sensing_get_sensors (*C function*), 1214
sensing_open_sensor (*C function*), 1214

sensing_open_sensor_by_dt (*C function*), 1214
SENSING_SENSITIVITY_INDEX_ALL (*C macro*), 1213
sensing_sensor_attribute (*C enum*), 1213
sensing_sensor_attribute.SENSING_SENSOR_ATTRIBUTE_INTERVAL (*C enumerator*), 1214
sensing_sensor_attribute.SENSING_SENSOR_ATTRIBUTE_LATENCY (*C enumerator*), 1214
sensing_sensor_attribute.SENSING_SENSOR_ATTRIBUTE_MAX (*C enumerator*), 1214
sensing_sensor_attribute.SENSING_SENSOR_ATTRIBUTE_SENSITIVITY (*C enumerator*), 1214
sensing_sensor_config (*C struct*), 1217
sensing_sensor_config.attri (*C var*), 1217
sensing_sensor_config.data_field (*C var*), 1217
sensing_sensor_config.interval (*C var*), 1217
sensing_sensor_config.latency (*C var*), 1217
sensing_sensor_config.sensitivity (*C var*), 1217
SENSING_SENSOR_FLAG_REPORT_ON_CHANGE (*C macro*), 1213
SENSING_SENSOR_FLAG_REPORT_ON_EVENT (*C macro*), 1213
sensing_sensor_get_reporters (*C function*), 1218
sensing_sensor_get_reporters_count (*C function*), 1218
sensing_sensor_get_state (*C function*), 1218
sensing_sensor_handle_t (*C type*), 1213
sensing_sensor_info (*C struct*), 1216
sensing_sensor_info.friendly_name (*C var*), 1216
sensing_sensor_info.minimal_interval (*C var*), 1216
sensing_sensor_info.model (*C var*), 1216
sensing_sensor_info.name (*C var*), 1216
sensing_sensor_info.type (*C var*), 1216
sensing_sensor_info.vendor (*C var*), 1216
sensing_sensor_register_info (*C struct*), 1219
sensing_sensor_register_info.flags (*C var*), 1219
sensing_sensor_register_info.sample_size (*C var*), 1219
sensing_sensor_register_info.sensitivity_count (*C var*), 1219
sensing_sensor_register_info.version (*C var*), 1219
sensing_sensor_state (*C enum*), 1213
sensing_sensor_state.SENSING_SENSOR_STATE_OFFLINE (*C enumerator*), 1213
sensing_sensor_state.SENSING_SENSOR_STATE_READY (*C enumerator*), 1213
SENSING_SENSOR_TYPE_ALL (*C macro*), 1210
SENSING_SENSOR_TYPE_LIGHT_AMBIENTLIGHT (*C macro*), 1209
SENSING_SENSOR_TYPE_MOTION_ACCELEROMETER_3D (*C macro*), 1209
SENSING_SENSOR_TYPE_MOTION_GYROMETER_3D (*C macro*), 1209
SENSING_SENSOR_TYPE_MOTION_HINGE_ANGLE (*C macro*), 1210
SENSING_SENSOR_TYPE_MOTION_MOTION_DETECTOR (*C macro*), 1210
SENSING_SENSOR_TYPE_MOTION_UNCALIB_ACCELEROMETER_3D (*C macro*), 1210
SENSING_SENSOR_TYPE_OTHER_CUSTOM (*C macro*), 1210
sensing_sensor_value_3d_q31 (*C struct*), 1210
sensing_sensor_value_3d_q31.header (*C var*), 1211
sensing_sensor_value_3d_q31.readings (*C var*), 1211
sensing_sensor_value_3d_q31.shift (*C var*), 1211
sensing_sensor_value_3d_q31.timestamp_delta (*C var*), 1211
sensing_sensor_value_3d_q31.v (*C var*), 1211
sensing_sensor_value_3d_q31.x (*C var*), 1211
sensing_sensor_value_3d_q31.y (*C var*), 1211
sensing_sensor_value_3d_q31.z (*C var*), 1211
sensing_sensor_value_header (*C struct*), 1210
sensing_sensor_value_header.base_timestamp (*C var*), 1210
sensing_sensor_value_header.reading_count (*C var*), 1210
sensing_sensor_value_q31 (*C struct*), 1212
sensing_sensor_value_q31.header (*C var*), 1212
sensing_sensor_value_q31.readings (*C var*), 1212
sensing_sensor_value_q31.shift (*C var*), 1212

sensing_sensor_value_q31.timestamp_delta (*C var*), 1212
sensing_sensor_value_q31.v (*C var*), 1212
sensing_sensor_value_uint32 (*C struct*), 1211
sensing_sensor_value_uint32.header (*C var*), 1211
sensing_sensor_value_uint32.readings (*C var*), 1212
sensing_sensor_value_uint32.timestamp_delta (*C var*), 1211
sensing_sensor_value_uint32.v (*C var*), 1212
SENSING_SENSOR_VERSION (*C macro*), 1212
sensing_sensor_version (*C struct*), 1215
sensing_sensor_version.build (*C var*), 1216
sensing_sensor_version.hotfix (*C var*), 1216
sensing_sensor_version.major (*C var*), 1216
sensing_sensor_version.minor (*C var*), 1216
sensing_sensor_version.value (*C var*), 1216
SENSING_SENSORS_DT_DEFINE (*C macro*), 1217
SENSING_SENSORS_DT_INST_DEFINE (*C macro*), 1218
sensing_set_config (*C function*), 1215
sensor_10udegrees_to_rad (*C function*), 3599
sensor_attr_get (*C function*), 3595
sensor_attr_get_t (*C type*), 3586
sensor_attr_set (*C function*), 3595
sensor_attr_set_t (*C type*), 3586
sensor_attribute (*C enum*), 3593
sensor_attribute.SENSOR_ATTR_ALERT (*C enumerator*), 3594
sensor_attribute.SENSOR_ATTR_BATCH_DURATION (*C enumerator*), 3594
sensor_attribute.SENSOR_ATTR_CALIB_TARGET (*C enumerator*), 3593
sensor_attribute.SENSOR_ATTR_CALIBRATION (*C enumerator*), 3593
sensor_attribute.SENSOR_ATTR_COMMON_COUNT (*C enumerator*), 3594
sensor_attribute.SENSOR_ATTR_CONFIGURATION (*C enumerator*), 3593
sensor_attribute.SENSOR_ATTR_FEATURE_MASK (*C enumerator*), 3593
sensor_attribute.SENSOR_ATTR_FF_DUR (*C enumerator*), 3594
sensor_attribute.SENSOR_ATTR_FULL_SCALE (*C enumerator*), 3593
sensor_attribute.SENSOR_ATTR_HYSTERESIS (*C enumerator*), 3593
sensor_attribute.SENSOR_ATTR_LOWER_THRESH (*C enumerator*), 3593
sensor_attribute.SENSOR_ATTR_MAX (*C enumerator*), 3594
sensor_attribute.SENSOR_ATTR_OFFSET (*C enumerator*), 3593
sensor_attribute.SENSOR_ATTR_OVERSAMPLING (*C enumerator*), 3593
sensor_attribute.SENSOR_ATTR_PRIV_START (*C enumerator*), 3594
sensor_attribute.SENSOR_ATTR_SAMPLING_FREQUENCY (*C enumerator*), 3593
sensor_attribute.SENSOR_ATTR_SLOPE_DUR (*C enumerator*), 3593
sensor_attribute.SENSOR_ATTR_SLOPE_TH (*C enumerator*), 3593
sensor_attribute.SENSOR_ATTR_UPPER_THRESH (*C enumerator*), 3593
sensor_chan_spec (*C struct*), 3601
sensor_chan_spec_eq (*C function*), 3594
sensor_chan_spec.chan_idx (*C var*), 3602
sensor_chan_spec.chan_type (*C var*), 3602
sensor_channel (*C enum*), 3587
SENSOR_CHANNEL_3_AXIS (*C macro*), 3585
sensor_channel_get (*C function*), 3596
sensor_channel_get_t (*C type*), 3586
sensor_channel.SENSOR_CHAN_ACCEL_X (*C enumerator*), 3587
sensor_channel.SENSOR_CHAN_ACCEL_XYZ (*C enumerator*), 3588
sensor_channel.SENSOR_CHAN_ACCEL_Y (*C enumerator*), 3587
sensor_channel.SENSOR_CHAN_ACCEL_Z (*C enumerator*), 3587
sensor_channel.SENSOR_CHAN_ALL (*C enumerator*), 3591
sensor_channel.SENSOR_CHAN_ALTITUDE (*C enumerator*), 3589
sensor_channel.SENSOR_CHAN_AMBIENT_TEMP (*C enumerator*), 3588
sensor_channel.SENSOR_CHAN_BLUE (*C enumerator*), 3589

sensor_channel.SENSOR_CHAN_CO2 (*C enumerator*), 3589
sensor_channel.SENSOR_CHAN_COMMON_COUNT (*C enumerator*), 3591
sensor_channel.SENSOR_CHAN_CURRENT (*C enumerator*), 3589
sensor_channel.SENSOR_CHAN_DIE_TEMP (*C enumerator*), 3588
sensor_channel.SENSOR_CHAN_DISTANCE (*C enumerator*), 3589
sensor_channel.SENSOR_CHAN_GAS_RES (*C enumerator*), 3589
sensor_channel.SENSOR_CHAN_GAUGE_AVG_CURRENT (*C enumerator*), 3590
sensor_channel.SENSOR_CHAN_GAUGE_AVG_POWER (*C enumerator*), 3591
sensor_channel.SENSOR_CHAN_GAUGE_CYCLE_COUNT (*C enumerator*), 3591
sensor_channel.SENSOR_CHAN_GAUGE_DESIGN_VOLTAGE (*C enumerator*), 3591
sensor_channel.SENSOR_CHAN_GAUGE_DESIRED_CHARGING_CURRENT (*C enumerator*), 3591
sensor_channel.SENSOR_CHAN_GAUGE_DESIRED_VOLTAGE (*C enumerator*), 3591
sensor_channel.SENSOR_CHAN_GAUGE_FULL_AVAIL_CAPACITY (*C enumerator*), 3591
sensor_channel.SENSOR_CHAN_GAUGE_FULL_CHARGE_CAPACITY (*C enumerator*), 3590
sensor_channel.SENSOR_CHAN_GAUGE_MAX_LOAD_CURRENT (*C enumerator*), 3590
sensor_channel.SENSOR_CHAN_GAUGE_NOM_AVAIL_CAPACITY (*C enumerator*), 3590
sensor_channel.SENSOR_CHAN_GAUGE_REMAINING_CHARGE_CAPACITY (*C enumerator*), 3590
sensor_channel.SENSOR_CHAN_GAUGE_STATE_OF_CHARGE (*C enumerator*), 3590
sensor_channel.SENSOR_CHAN_GAUGE_STATE_OF_HEALTH (*C enumerator*), 3591
sensor_channel.SENSOR_CHAN_GAUGE_STDBY_CURRENT (*C enumerator*), 3590
sensor_channel.SENSOR_CHAN_GAUGE_TEMP (*C enumerator*), 3590
sensor_channel.SENSOR_CHAN_GAUGE_TIME_TO_EMPTY (*C enumerator*), 3591
sensor_channel.SENSOR_CHAN_GAUGE_TIME_TO_FULL (*C enumerator*), 3591
sensor_channel.SENSOR_CHAN_GAUGE_VOLTAGE (*C enumerator*), 3590
sensor_channel.SENSOR_CHAN_GREEN (*C enumerator*), 3589
sensor_channel.SENSOR_CHAN_GYRO_X (*C enumerator*), 3588
sensor_channel.SENSOR_CHAN_GYRO_XYZ (*C enumerator*), 3588
sensor_channel.SENSOR_CHAN_GYRO_Y (*C enumerator*), 3588
sensor_channel.SENSOR_CHAN_GYRO_Z (*C enumerator*), 3588
sensor_channel.SENSOR_CHAN_HUMIDITY (*C enumerator*), 3588
sensor_channel.SENSOR_CHAN_IR (*C enumerator*), 3588
sensor_channel.SENSOR_CHAN_LIGHT (*C enumerator*), 3588
sensor_channel.SENSOR_CHAN_MAGN_X (*C enumerator*), 3588
sensor_channel.SENSOR_CHAN_MAGN_XYZ (*C enumerator*), 3588
sensor_channel.SENSOR_CHAN_MAGN_Y (*C enumerator*), 3588
sensor_channel.SENSOR_CHAN_MAGN_Z (*C enumerator*), 3588
sensor_channel.SENSOR_CHAN_MAX (*C enumerator*), 3591
sensor_channel.SENSOR_CHAN_O2 (*C enumerator*), 3589
sensor_channel.SENSOR_CHAN_PM_1_0 (*C enumerator*), 3589
sensor_channel.SENSOR_CHAN_PM_2_5 (*C enumerator*), 3589
sensor_channel.SENSOR_CHAN_PM_10 (*C enumerator*), 3589
sensor_channel.SENSOR_CHAN_POS_DX (*C enumerator*), 3590
sensor_channel.SENSOR_CHAN_POS_DXYZ (*C enumerator*), 3590
sensor_channel.SENSOR_CHAN_POS_DY (*C enumerator*), 3590
sensor_channel.SENSOR_CHAN_POS_DZ (*C enumerator*), 3590
sensor_channel.SENSOR_CHAN_POWER (*C enumerator*), 3589
sensor_channel.SENSOR_CHAN_PRESS (*C enumerator*), 3588
sensor_channel.SENSOR_CHAN_PRIV_START (*C enumerator*), 3591
sensor_channel.SENSOR_CHAN_PROX (*C enumerator*), 3588
sensor_channel.SENSOR_CHAN_RED (*C enumerator*), 3589
sensor_channel.SENSOR_CHAN_RESISTANCE (*C enumerator*), 3590
sensor_channel.SENSOR_CHAN_ROTATION (*C enumerator*), 3590
sensor_channel.SENSOR_CHAN_RPM (*C enumerator*), 3590
sensor_channel.SENSOR_CHAN_VOC (*C enumerator*), 3589
sensor_channel.SENSOR_CHAN_VOLTAGE (*C enumerator*), 3589
sensor_channel.SENSOR_CHAN_VSHUNT (*C enumerator*), 3589
sensor_data_generic_header (*C struct*), 3604
sensor_decode (*C function*), 3595

sensor_decode_context (*C struct*), 3603
SENSOR_DECODE_CONTEXT_INIT (*C macro*), 3584
sensor_decoder_api (*C struct*), 3602
sensor_decoder_api.decode (*C var*), 3602
sensor_decoder_api.get_frame_count (*C var*), 3602
sensor_decoder_api.get_size_info (*C var*), 3602
sensor_decoder_api.has_trigger (*C var*), 3603
sensor_degrees_to_rad (*C function*), 3599
SENSOR_DEVICE_DT_DEFINE (*C macro*), 3585
SENSOR_DEVICE_DT_INST_DEFINE (*C macro*), 3586
sensor_driver_api (*C struct*), 3604
SENSOR_DT_READ_IODEV (*C macro*), 3584
SENSOR_DT_STREAM_IODEV (*C macro*), 3585
SENSOR_G (*C macro*), 3585
sensor_g_to_ms2 (*C function*), 3598
sensor_get_decoder (*C function*), 3597
sensor_get_decoder_t (*C type*), 3587
SENSOR_INFO_DEFINE (*C macro*), 3585
SENSOR_INFO_DT_DEFINE (*C macro*), 3585
sensor_ms2_to_g (*C function*), 3598
sensor_ms2_to_ug (*C function*), 3599
sensor_natively_supported_channel_size_info (*C function*), 3595
SENSOR_PI (*C macro*), 3585
sensor_processing_callback_t (*C type*), 3587
sensor_processing_with_callback (*C function*), 3598
sensor_rad_to_10udegrees (*C function*), 3599
sensor_rad_to_degrees (*C function*), 3599
sensor_read (*C function*), 3597
sensor_read_async_mempool (*C function*), 3598
sensor_read_config (*C struct*), 3604
sensor_reconfigure_read_iodev (*C function*), 3597
sensor_sample_fetch (*C function*), 3596
sensor_sample_fetch_chan (*C function*), 3596
sensor_sample_fetch_t (*C type*), 3586
sensor_stream (*C function*), 3597
sensor_stream_data_opt (*C enum*), 3594
sensor_stream_data_opt.SENSOR_STREAM_DATA_DROP (*C enumerator*), 3594
sensor_stream_data_opt.SENSOR_STREAM_DATA_INCLUDE (*C enumerator*), 3594
sensor_stream_data_opt.SENSOR_STREAM_DATA_NOP (*C enumerator*), 3594
sensor_stream_trigger (*C struct*), 3603
SENSOR_STREAM_TRIGGER_PREP (*C macro*), 3584
sensor_submit_t (*C type*), 3587
sensor_trigger (*C struct*), 3601
sensor_trigger_handler_t (*C type*), 3586
sensor_trigger_set (*C function*), 3595
sensor_trigger_set_t (*C type*), 3586
sensor_trigger_type (*C enum*), 3591
sensor_trigger_type.SENSOR_TRIG_COMMON_COUNT (*C enumerator*), 3592
sensor_trigger_type.SENSOR_TRIG_DATA_READY (*C enumerator*), 3592
sensor_trigger_type.SENSOR_TRIG_DELTA (*C enumerator*), 3592
sensor_trigger_type.SENSOR_TRIG_DOUBLE_TAP (*C enumerator*), 3592
sensor_trigger_type.SENSOR_TRIG_FIFO_FULL (*C enumerator*), 3592
sensor_trigger_type.SENSOR_TRIG_FIFO_WATERMARK (*C enumerator*), 3592
sensor_trigger_type.SENSOR_TRIG_FREEFALL (*C enumerator*), 3592
sensor_trigger_type.SENSOR_TRIG_MAX (*C enumerator*), 3592
sensor_trigger_type.SENSOR_TRIG_MOTION (*C enumerator*), 3592
sensor_trigger_type.SENSOR_TRIG_NEAR_FAR (*C enumerator*), 3592
sensor_trigger_type.SENSOR_TRIG_PRIV_START (*C enumerator*), 3592

sensor_trigger_type.SENSOR_TRIG_STATIONARY (*C enumerator*), 3592
sensor_trigger_type.SENSOR_TRIG_TAP (*C enumerator*), 3592
sensor_trigger_type.SENSOR_TRIG_THRESHOLD (*C enumerator*), 3592
sensor_trigger_type.SENSOR_TRIG_TIMER (*C enumerator*), 3591
sensor_trigger.chan (*C var*), 3601
sensor_trigger.type (*C var*), 3601
sensor_ug_to_ms2 (*C function*), 3599
sensor_value (*C struct*), 3601
sensor_value_from_double (*C function*), 3600
sensor_value_from_float (*C function*), 3600
sensor_value_from_micro (*C function*), 3600
sensor_value_from_milli (*C function*), 3600
sensor_value_to_double (*C function*), 3599
sensor_value_to_float (*C function*), 3599
sensor_value_to_micro (*C function*), 3600
sensor_value_to_milli (*C function*), 3600
sensor_value.val1 (*C var*), 3601
sensor_value.val2 (*C var*), 3601
setsockopt (*C function*), 2489
settings_call_set_handler (*C function*), 1161
settings_commit (*C function*), 1156
settings_commit_subtree (*C function*), 1156
settings_delete (*C function*), 1156
settings_dst_register (*C function*), 1160
SETTINGS_EXTRA_LEN (*C macro*), 1153
settings_handler (*C struct*), 1156
settings_handler_static (*C struct*), 1157
settings_handler_static.h_commit (*C var*), 1158
settings_handler_static.h_export (*C var*), 1158
settings_handler_static.h_get (*C var*), 1158
settings_handler_static.h_set (*C var*), 1158
settings_handler_static.name (*C var*), 1158
settings_handler.h_commit (*C var*), 1157
settings_handler.h_export (*C var*), 1157
settings_handler.h_get (*C var*), 1157
settings_handler.h_set (*C var*), 1157
settings_handler.name (*C var*), 1157
settings_handler.node (*C var*), 1157
settings_load (*C function*), 1155
settings_load_arg (*C struct*), 1161
settings_load_arg.cb (*C var*), 1161
settings_load_arg.param (*C var*), 1162
settings_load_arg.subtree (*C var*), 1161
settings_load_direct_cb (*C type*), 1154
settings_load_subtree (*C function*), 1155
settings_load_subtree_direct (*C function*), 1155
SETTINGS_MAX_DIR_DEPTH (*C macro*), 1153
SETTINGS_MAX_NAME_LEN (*C macro*), 1153
SETTINGS_MAX_VAL_LEN (*C macro*), 1153
settings_mgmt_group_events (*C enum*), 771
settings_mgmt_group_events.MGMT_EVT_OP_SETTINGS_MGMT_ACCESS (*C enumerator*), 771
settings_mgmt_group_events.MGMT_EVT_OP_SETTINGS_MGMT_ALL (*C enumerator*), 771
SETTINGS_NAME_END (*C macro*), 1153
settings_name_next (*C function*), 1159
SETTINGS_NAME_SEPARATOR (*C macro*), 1153
settings_name_steq (*C function*), 1159
settings_parse_and_lookup (*C function*), 1160
settings_read_cb (*C type*), 1154

settings_register (*C function*), 1154
settings_runtime_commit (*C function*), 1160
settings_runtime_get (*C function*), 1160
settings_runtime_set (*C function*), 1160
settings_save (*C function*), 1155
settings_save_one (*C function*), 1156
settings_save_subtree (*C function*), 1155
settings_src_register (*C function*), 1160
SETTINGS_STATIC_HANDLER_DEFINE (*C macro*), 1153
settings_store (*C struct*), 1161
settings_store_itf (*C struct*), 1162
settings_store_itf.csi_load (*C var*), 1162
settings_store_itf.csi_save (*C var*), 1162
settings_store_itf.csi_save_end (*C var*), 1162
settings_store_itf.csi_save_start (*C var*), 1162
settings_store.cs_itf (*C var*), 1161
settings_store.cs_next (*C var*), 1161
settings_subsys_init (*C function*), 1154
shared_multi_heap_add (*C function*), 579
shared_multi_heap_aligned_alloc (*C function*), 579
shared_multi_heap_alloc (*C function*), 578
shared_multi_heap_attr (*C enum*), 578
shared_multi_heap_attr.SMH_REG_ATTR_CACHEABLE (*C enumerator*), 578
shared_multi_heap_attr.SMH_REG_ATTR_NON_CACHEABLE (*C enumerator*), 578
shared_multi_heap_attr.SMH_REG_ATTR_NUM (*C enumerator*), 578
shared_multi_heap_free (*C function*), 579
shared_multi_heap_pool_init (*C function*), 578
shared_multi_heap_region (*C struct*), 579
shared_multi_heap_region.addr (*C var*), 580
shared_multi_heap_region.attr (*C var*), 580
shared_multi_heap_region.size (*C var*), 580
shell (*C struct*), 1146
Shell (*class in twister_harness*), 268
shell_backend_cfg (*C union*), 1145
shell_backend_cfg.flags (*C var*), 1145
shell_backend_cfg.value (*C var*), 1145
shell_backend_config_flags (*C struct*), 1144
shell_backend_config_flags.echo (*C var*), 1144
shell_backend_config_flags.insert_mode (*C var*), 1144
shell_backend_config_flags.mode_delete (*C var*), 1144
shell_backend_config_flags.obscure (*C var*), 1144
shell_backend_config_flags.use_colors (*C var*), 1144
shell_backend_config_flags.use_vt100 (*C var*), 1144
shell_backend_ctx (*C union*), 1145
shell_backend_ctx_flags (*C struct*), 1144
shell_backend_ctx_flags.cmd_ctx (*C var*), 1145
shell_backend_ctx_flags.handle_log (*C var*), 1145
shell_backend_ctx_flags.history_exit (*C var*), 1145
shell_backend_ctx_flags.last_nl (*C var*), 1145
shell_backend_ctx_flags.print_noinit (*C var*), 1145
shell_backend_ctx_flags.processing (*C var*), 1145
shell_backend_ctx_flags.sync_mode (*C var*), 1145
shell_backend_ctx.flags (*C var*), 1145
shell_backend_ctx.value (*C var*), 1145
shell_bypass_cb_t (*C type*), 1135
SHELL_CMD (*C macro*), 1130
SHELL_CMD_ARG (*C macro*), 1129
SHELL_CMD_ARG_REGISTER (*C macro*), 1126

SHELL_CMD_DICT_CREATE (*C macro*), 1131
shell_cmd_entry (*C union*), 1141
shell_cmd_entry.dynamic_get (*C var*), 1142
shell_cmd_entry.entry (*C var*), 1142
shell_cmd_handler (*C type*), 1134
SHELL_CMD_HELP_PRINTED (*C macro*), 1133
SHELL_CMD_REGISTER (*C macro*), 1127
SHELL_COND_CMD (*C macro*), 1131
SHELL_COND_CMD_ARG (*C macro*), 1129
SHELL_COND_CMD_ARG_REGISTER (*C macro*), 1126
SHELL_COND_CMD_REGISTER (*C macro*), 1127
shell_ctx (*C struct*), 1145
shell_ctx.active_cmd (*C var*), 1146
shell_ctx.bypass (*C var*), 1146
shell_ctx.cmd_buff (*C var*), 1146
shell_ctx.cmd_buff_len (*C var*), 1146
shell_ctx.cmd_buff_pos (*C var*), 1146
shell_ctx.cmd_tmp_buff_len (*C var*), 1146
shell_ctx.events (*C var*), 1146
shell_ctx.printf_buff (*C var*), 1146
shell_ctx.receive_state (*C var*), 1146
shell_ctx.selected_cmd (*C var*), 1146
shell_ctx.state (*C var*), 1146
shell_ctx.temp_buff (*C var*), 1146
shell_ctx.uninit_cb (*C var*), 1146
shell_ctx.vt100_ctx (*C var*), 1146
SHELL_DEFAULT_BACKEND_CONFIG_FLAGS (*C macro*), 1132
SHELL_DEFINE (*C macro*), 1132
shell_device_filter (*C function*), 1136
shell_device_filter_t (*C type*), 1134
shell_device_lookup (*C function*), 1136
shell_dict_cmd_handler (*C type*), 1134
SHELL_DYNAMIC_CMD_CREATE (*C macro*), 1129
shell_dynamic_get (*C type*), 1134
shell_echo_set (*C function*), 1140
SHELL_ERROR (*C macro*), 1133
shell_error (*C macro*), 1133
shell_execute_cmd (*C function*), 1139
SHELL_EXPR_CMD (*C macro*), 1131
SHELL_EXPR_CMD_ARG (*C macro*), 1130
shell_flag (*C enum*), 1136
shell_flag.SHELL_FLAG_CRLF_DEFAULT (*C enumerator*), 1136
shell_flag.SHELL_FLAG_OLF_CRLF (*C enumerator*), 1136
shell_fprintf (*C macro*), 1133
shell_fprintf_impl (*C function*), 1137
shell_get_return_value (*C function*), 1141
shell_help (*C function*), 1139
shell_hexdump (*C function*), 1138
shell_hexdump_line (*C function*), 1138
SHELL_INFO (*C macro*), 1132
shell_info (*C macro*), 1133
shell_init (*C function*), 1137
shell_insert_mode_set (*C function*), 1140
shell_mode_delete_set (*C function*), 1141
SHELL_NORMAL (*C macro*), 1132
shell_obscure_set (*C function*), 1141
SHELL_OPTION (*C macro*), 1132
shell_print (*C macro*), 1133

shell_process (*C function*), 1138
shell_prompt_change (*C function*), 1139
shell_ready (*C function*), 1140
shell_receive_state (*C enum*), 1135
shell_receive_state.SHELL_RECEIVE_DEFAULT (*C enumerator*), 1135
shell_receive_state.SHELL_RECEIVE_ESC (*C enumerator*), 1135
shell_receive_state.SHELL_RECEIVE_ESC_SEQ (*C enumerator*), 1135
shell_receive_state.SHELL_RECEIVE_TILDE_EXP (*C enumerator*), 1135
shell_set_bypass (*C function*), 1139
shell_set_root_cmd (*C function*), 1139
shell_signal (*C enum*), 1136
shell_signal.SHELL_SIGNAL_KILL (*C enumerator*), 1136
shell_signal.SHELL_SIGNAL_LOG_MSG (*C enumerator*), 1136
shell_signal.SHELL_SIGNAL_RXRDY (*C enumerator*), 1136
shell_signal.SHELL_SIGNAL_TXDONE (*C enumerator*), 1136
shell_signal.SHELL_SIGNALS (*C enumerator*), 1136
shell_start (*C function*), 1137
shell_state (*C enum*), 1135
shell_state.SHELL_STATE_ACTIVE (*C enumerator*), 1135
shell_state.SHELL_STATE_INITIALIZED (*C enumerator*), 1135
shell_state.SHELL_STATE_PANIC_MODE_ACTIVE (*C enumerator*), 1135
shell_state.SHELL_STATE_PANIC_MODE_INACTIVE (*C enumerator*), 1135
shell_state.SHELL_STATE_UNINITIALIZED (*C enumerator*), 1135
shell_static_args (*C struct*), 1142
shell_static_args.mandatory (*C var*), 1142
shell_static_args.optional (*C var*), 1142
shell_static_entry (*C struct*), 1142
shell_static_entry.args (*C var*), 1142
shell_static_entry.handler (*C var*), 1142
shell_static_entry.help (*C var*), 1142
shell_static_entry.subcmd (*C var*), 1142
shell_static_entry.syntax (*C var*), 1142
SHELL_STATIC_SUBCMD_SET_CREATE (*C macro*), 1127
shell_stats (*C struct*), 1144
shell_stats.log_lost_cnt (*C var*), 1144
shell_stop (*C function*), 1137
SHELL_SUBCMD_ADD (*C macro*), 1129
SHELL_SUBCMD_COND_ADD (*C macro*), 1128
SHELL_SUBCMD_DICT_SET_CREATE (*C macro*), 1131
SHELL_SUBCMD_SET_CREATE (*C macro*), 1128
SHELL_SUBCMD_SET_END (*C macro*), 1129
shell_transport (*C struct*), 1144
shell_transport_api (*C struct*), 1142
shell_transport_api.enable (*C var*), 1143
shell_transport_api.init (*C var*), 1142
shell_transport_api.read (*C var*), 1143
shell_transport_api.uninit (*C var*), 1143
shell_transport_api.update (*C var*), 1144
shell_transport_api.write (*C var*), 1143
shell_transport_evt (*C enum*), 1136
shell_transport_evt.SHELL_TRANSPORT_EVT_RX_RDY (*C enumerator*), 1136
shell_transport_evt.SHELL_TRANSPORT_EVT_TX_RDY (*C enumerator*), 1136
shell_transport_handler_t (*C type*), 1135
shell_uninit (*C function*), 1137
shell_uninit_cb_t (*C type*), 1135
shell_use_colors_set (*C function*), 1140
shell_use_vt100_set (*C function*), 1140
shell_vfprintf (*C function*), 1138

shell_warn (*C macro*), 1133
SHELL_WARNING (*C macro*), 1132
shell.ctx (*C var*), 1147
shell.default_prompt (*C var*), 1147
shell.iface (*C var*), 1147
SHUT_RD (*C macro*), 2490
SHUT_RDWR (*C macro*), 2490
SHUT_WR (*C macro*), 2490
shutdown (*C function*), 2488
sign_extend (*C function*), 698
sign_extend_64 (*C function*), 698
SINK_TX_NG (*C macro*), 3676
SINK_TX_OK (*C macro*), 3676
SIZEOF_FIELD (*C macro*), 682
SMBUS_ADDRESS_ARA (*C macro*), 3625
SMBUS_BLOCK_BYTES_MAX (*C macro*), 3625
smbus_block_pcall (*C function*), 3632
smbus_block_read (*C function*), 3631
smbus_block_write (*C function*), 3631
smbus_byte_data_read (*C function*), 3629
smbus_byte_data_write (*C function*), 3629
smbus_byte_read (*C function*), 3629
smbus_byte_write (*C function*), 3629
smbus_callback (*C struct*), 3632
smbus_callback_handler_t (*C type*), 3626
smbus_callback.addr (*C var*), 3632
smbus_callback.handler (*C var*), 3632
smbus_callback.node (*C var*), 3632
SMBUS_CMD_BLOCK (*C macro*), 3624
SMBUS_CMD_BLOCK_PROC (*C macro*), 3624
SMBUS_CMD_BYTE (*C macro*), 3622
SMBUS_CMD_BYTE_DATA (*C macro*), 3623
SMBUS_CMD_PROC_CALL (*C macro*), 3624
SMBUS_CMD_QUICK (*C macro*), 3622
SMBUS_CMD_WORD_DATA (*C macro*), 3623
smbus_configure (*C function*), 3627
SMBUS_DEVICE_DT_DEFINE (*C macro*), 3626
SMBUS_DEVICE_DT_INST_DEFINE (*C macro*), 3626
smbus_direction (*C enum*), 3622
smbus_direction.SMBUS_MSG_READ (*C enumerator*), 3622
smbus_direction.SMBUS_MSG_WRITE (*C enumerator*), 3622
smbus_dt_spec (*C struct*), 3632
SMBUS_DT_SPEC_GET (*C macro*), 3625
SMBUS_DT_SPEC_INST_GET (*C macro*), 3625
smbus_dt_spec.addr (*C var*), 3633
smbus_dt_spec.bus (*C var*), 3633
smbus_get_config (*C function*), 3627
smbus_host_notify_remove_cb (*C function*), 3628
smbus_host_notify_set_cb (*C function*), 3628
SMBUS_MODE_CONTROLLER (*C macro*), 3625
SMBUS_MODE_HOST_NOTIFY (*C macro*), 3625
SMBUS_MODE_PEC (*C macro*), 3625
SMBUS_MODE_SMBALERT (*C macro*), 3625
smbus_pcall (*C function*), 3631
smbus_quick (*C function*), 3628
smbus_smbalert_remove_cb (*C function*), 3627
smbus_smbalert_set_cb (*C function*), 3627
smbus_word_data_read (*C function*), 3630

smbus_word_data_write (*C function*), 3630
smbus_xfer_stats (*C function*), 3626
SMF_CREATE_STATE (*C macro*), 1173
SMF_CTX (*C macro*), 1173
smf_ctx (*C struct*), 1174
smf_ctx.current (*C var*), 1174
smf_ctx.internal (*C var*), 1175
smf_ctx.previous (*C var*), 1174
smf_ctx.terminate_val (*C var*), 1175
smf_run_state (*C function*), 1174
smf_set_handled (*C function*), 1174
smf_set_initial (*C function*), 1173
smf_set_state (*C function*), 1173
smf_set_terminate (*C function*), 1174
smf_state (*C struct*), 1174
smf_state.entry (*C var*), 1174
smf_state.exit (*C var*), 1174
smf_state.run (*C var*), 1174
smp_all_events (*C enum*), 769
smp_all_events.MGMT_EVT_OP_ALL (*C enumerator*), 769
smp_client_transport_entry (*C struct*), 819
smp_client_transport_entry.smpt (*C var*), 819
smp_client_transport_entry.smpt_type (*C var*), 819
smp_client_transport_get (*C function*), 818
smp_client_transport_register (*C function*), 818
smp_group_events (*C enum*), 769
smp_group_events.MGMT_EVT_OP_CMD_ALL (*C enumerator*), 770
smp_group_events.MGMT_EVT_OP_CMD_DONE (*C enumerator*), 770
smp_group_events.MGMT_EVT_OP_CMD_RECV (*C enumerator*), 769
smp_group_events.MGMT_EVT_OP_CMD_STATUS (*C enumerator*), 770
smp_rx_clear (*C function*), 818
smp_rx_remove_invalid (*C function*), 818
smp_transport (*C struct*), 819
smp_transport_api_t (*C struct*), 818
smp_transport_api_t.get_mtu (*C var*), 818
smp_transport_api_t.output (*C var*), 818
smp_transport_api_t.query_valid_check (*C var*), 819
smp_transport_api_t.ud_copy (*C var*), 818
smp_transport_api_t.ud_free (*C var*), 819
smp_transport_get_mtu_fn (*C type*), 816
smp_transport_init (*C function*), 817
smp_transport_out_fn (*C type*), 816
smp_transport_query_valid_check_fn (*C type*), 817
smp_transport_type (*C enum*), 817
smp_transport_type.SMP_BLUETOOTH_TRANSPORT (*C enumerator*), 817
smp_transport_type.SMP_SERIAL_TRANSPORT (*C enumerator*), 817
smp_transport_type.SMP_SHELL_TRANSPORT (*C enumerator*), 817
smp_transport_type.SMP_UDP_IPV4_TRANSPORT (*C enumerator*), 817
smp_transport_type.SMP_UDP_IPV6_TRANSPORT (*C enumerator*), 817
smp_transport_type.SMP_USER_DEFINED_TRANSPORT (*C enumerator*), 817
smp_transport_ud_copy_fn (*C type*), 816
smp_transport_ud_free_fn (*C type*), 816
snprintfcb (*C function*), 871
sntp_close (*C function*), 2566
sntp_ctx (*C struct*), 2567
sntp_ctx.expected_orig_ts (*C var*), 2567
sntp_init (*C function*), 2566
sntp_query (*C function*), 2566

sntp_simple (*C function*), 2566
sntp_simple_addr (*C function*), 2567
sntp_time (*C struct*), 2567
sntp_time.fraction (*C var*), 2567
sntp_time.seconds (*C var*), 2567
SO_ACCEPTCONN (*C macro*), 2494
SO_BINDTODEVICE (*C macro*), 2494
SO_BROADCAST (*C macro*), 2493
SO_DEBUG (*C macro*), 2493
SO_DOMAIN (*C macro*), 2494
SO_DONTROUTE (*C macro*), 2493
SO_ERROR (*C macro*), 2493
SO_KEEPALIVE (*C macro*), 2494
SO_LINGER (*C macro*), 2494
SO_OOBINLINE (*C macro*), 2494
SO_PRIORITY (*C macro*), 2494
SO_PROTOCOL (*C macro*), 2494
SO_RCVBUF (*C macro*), 2494
SO_RCVLOWAT (*C macro*), 2494
SO_RCVTIMEO (*C macro*), 2494
SO_REUSEADDR (*C macro*), 2493
SO_REUSEPORT (*C macro*), 2494
SO_SNDBUF (*C macro*), 2493
SO_SNDLOWAT (*C macro*), 2494
SO_SNDTIMEO (*C macro*), 2494
SO_SOCKS5 (*C macro*), 2494
SO_TIMESTAMPING (*C macro*), 2494
SO_TXTIME (*C macro*), 2495
SO_TYPE (*C macro*), 2493
SoC, **3936**
SoC family, **3936**
SoC series, **3936**
SOC_FLASH_0_ID (*C macro*), 1189
soc_timing_counter_get (*C function*), 657
soc_timing_cycles_get (*C function*), 657
soc_timing_cycles_to_ns (*C function*), 658
soc_timing_cycles_to_ns_avg (*C function*), 658
soc_timing_freq_get (*C function*), 657
soc_timing_freq_get_mhz (*C function*), 658
soc_timing_init (*C function*), 656
soc_timing_start (*C function*), 656
soc_timing_stop (*C function*), 656
sockaddr (*C struct*), 2528
sockaddr_in (*C struct*), 2526
sockaddr_in6 (*C struct*), 2526
sockaddr_in6.sin6_addr (*C var*), 2526
sockaddr_in6.sin6_family (*C var*), 2526
sockaddr_in6.sin6_port (*C var*), 2526
sockaddr_in6.sin6_scope_id (*C var*), 2526
sockaddr_in.sin_addr (*C var*), 2526
sockaddr_in.sin_family (*C var*), 2526
sockaddr_in.sin_port (*C var*), 2526
sockaddr_ll (*C struct*), 2526
sockaddr_ll.sll_addr (*C var*), 2527
sockaddr_ll.sll_family (*C var*), 2526
sockaddr_ll.sll_halen (*C var*), 2527
sockaddr_ll.sll_hatype (*C var*), 2527
sockaddr_ll.sll_ifindex (*C var*), 2527

sockaddr_ll.sll_pkttype (*C var*), 2527
sockaddr_ll.sll_protocol (*C var*), 2527
sockaddr.sa_family (*C var*), 2528
socket (*C function*), 2488
socketpair (*C function*), 2488
socklen_t (*C type*), 2510
SOF_TIMESTAMPING_RX_HARDWARE (*C macro*), 2495
SOF_TIMESTAMPING_TX_HARDWARE (*C macro*), 2495
SOL_SOCKET (*C macro*), 2493
SOL_TLS (*C macro*), 2481
SOMAXCONN (*C macro*), 2497
spi_api_io (*C type*), 3611
spi_api_io_async (*C type*), 3612
spi_api_release (*C type*), 3612
spi_buf (*C struct*), 3620
spi_buf_set (*C struct*), 3620
spi_buf_set.buffers (*C var*), 3620
spi_buf_set.count (*C var*), 3620
spi_buf.buf (*C var*), 3620
spi_buf.len (*C var*), 3620
spi_callback_t (*C type*), 3612
spi_config (*C struct*), 3619
SPI_CONFIG_DT (*C macro*), 3610
SPI_CONFIG_DT_INST (*C macro*), 3610
spi_config.cs (*C var*), 3620
spi_config.frequency (*C var*), 3619
spi_config.operation (*C var*), 3619
spi_config.slave (*C var*), 3620
SPI_CS_ACTIVE_HIGH (*C macro*), 3607
spi_cs_control (*C struct*), 3619
SPI_CS_CONTROL_INIT (*C macro*), 3609
SPI_CS_CONTROL_INIT_INST (*C macro*), 3610
spi_cs_control.delay (*C var*), 3619
spi_cs_control.gpio (*C var*), 3619
SPI_CS_GPIOS_DT_SPEC_GET (*C macro*), 3609
SPI_CS_GPIOS_DT_SPEC_INST_GET (*C macro*), 3609
spi_cs_is_gpio (*C function*), 3612
spi_cs_is_gpio_dt (*C function*), 3612
SPI_DEVICE_DT_DEFINE (*C macro*), 3611
spi_driver_api (*C struct*), 3620
SPI_DT_IODEV_DEFINE (*C macro*), 3611
spi_dt_spec (*C struct*), 3620
SPI_DT_SPEC_GET (*C macro*), 3610
SPI_DT_SPEC_INST_GET (*C macro*), 3611
spi_dt_spec.bus (*C var*), 3620
spi_dt_spec.config (*C var*), 3620
SPI_FLASH_0_ID (*C macro*), 1189
SPI_FRAME_FORMAT_MOTOROLA (*C macro*), 3608
SPI_FRAME_FORMAT_TI (*C macro*), 3608
SPI_FULL_DUPLEX (*C macro*), 3608
SPI_HALF_DUPLEX (*C macro*), 3608
SPI_HOLD_ON_CS (*C macro*), 3607
spi_iodev_api (*C var*), 3619
spi_iodev_submit (*C function*), 3617
spi_is_ready_dt (*C function*), 3612
spi_is_ready_iodev (*C function*), 3618
SPI_LINES_DUAL (*C macro*), 3608
SPI_LINES_MASK (*C macro*), 3608

SPI_LINES_OCTAL (*C macro*), 3608
SPI_LINES_QUAD (*C macro*), 3608
SPI_LINES_SINGLE (*C macro*), 3608
SPI_LOCK_ON (*C macro*), 3607
SPI_MODE_CPHA (*C macro*), 3607
SPI_MODE_CPOL (*C macro*), 3606
SPI_MODE_GET (*C macro*), 3607
SPI_MODE_LOOP (*C macro*), 3607
SPI_OP_MODE_GET (*C macro*), 3606
SPI_OP_MODE_MASTER (*C macro*), 3606
SPI_OP_MODE_SLAVE (*C macro*), 3606
spi_operation_t (*C type*), 3611
spi_read (*C function*), 3613
spi_read_dt (*C function*), 3614
spi_read_signal (*C function*), 3616
spi_release (*C function*), 3618
spi_release_dt (*C function*), 3618
spi_rtio_copy (*C function*), 3618
SPI_STATS_RX_BYTES_INC (*C macro*), 3611
SPI_STATS_TRANSFER_ERROR_INC (*C macro*), 3611
SPI_STATS_TX_BYTES_INC (*C macro*), 3611
spi_transceive (*C function*), 3613
spi_transceive_cb (*C function*), 3615
spi_transceive_dt (*C function*), 3613
spi_transceive_signal (*C function*), 3615
spi_transceive_stats (*C macro*), 3611
SPI_TRANSFER_LSB (*C macro*), 3607
SPI_TRANSFER_MSB (*C macro*), 3607
SPI_WORD_SET (*C macro*), 3607
SPI_WORD_SIZE_GET (*C macro*), 3607
spi_write (*C function*), 3614
spi_write_dt (*C function*), 3615
spi_write_signal (*C function*), 3617
spsc (*C struct*), 650
spsc_acquirable (*C macro*), 649
spsc_acquire (*C macro*), 649
spsc_consumable (*C macro*), 649
spsc_consume (*C macro*), 649
SPSC_DECLARE (*C macro*), 648
SPSC_DEFINE (*C macro*), 648
spsc_drop_all (*C macro*), 649
SPSC_INITIALIZER (*C macro*), 648
spsc_next (*C macro*), 650
spsc_peek (*C macro*), 650
spsc_prev (*C macro*), 650
spsc_produce (*C macro*), 649
spsc_produce_all (*C macro*), 649
spsc_release (*C macro*), 649
spsc_release_all (*C macro*), 649
spsc_reset (*C macro*), 648
spsc_size (*C macro*), 648
STATE_DOWNLOADED (*C macro*), 2825
STATE_DOWNLOADING (*C macro*), 2824
state_execution (*C type*), 1173
STATE_IDLE (*C macro*), 2824
STATE_UPDATING (*C macro*), 2825
STM32_GPIO_WKUP (*C macro*), 3364
STP_DECODER_TYPE2STR (*C macro*), 743

stream_flash_buffered_write (*C function*), 1202
stream_flash_bytes_written (*C function*), 1201
stream_flash_callback_t (*C type*), 1201
stream_flash_ctx (*C struct*), 1203
stream_flash_erase_page (*C function*), 1202
stream_flash_init (*C function*), 1201
stream_flash_progress_clear (*C function*), 1203
stream_flash_progress_load (*C function*), 1202
stream_flash_progress_save (*C function*), 1203
STRUCT_SECTION_COUNT (*C macro*), 705
STRUCT_SECTION_END (*C macro*), 703
STRUCT_SECTION_END_EXTERN (*C macro*), 703
STRUCT_SECTION_FOREACH (*C macro*), 705
STRUCT_SECTION_FOREACH_ALTERNATE (*C macro*), 705
STRUCT_SECTION_GET (*C macro*), 705
STRUCT_SECTION_ITERABLE (*C macro*), 704
STRUCT_SECTION_ITERABLE_ALTERNATE (*C macro*), 704
STRUCT_SECTION_ITERABLE_ARRAY (*C macro*), 704
STRUCT_SECTION_ITERABLE_ARRAY_ALTERNATE (*C macro*), 704
STRUCT_SECTION_ITERABLE_NAMED (*C macro*), 704
STRUCT_SECTION_ITERABLE_NAMED_ALTERNATE (*C macro*), 704
STRUCT_SECTION_START (*C macro*), 703
STRUCT_SECTION_START_EXTERN (*C macro*), 703
subsystem, **3937**
symtab_find_symbol_name (*C function*), 747
symtab_get (*C function*), 747
symtab_info (*C struct*), 747
sys_cache_cached_ptr_get (*C function*), 3137
sys_cache_data_disable (*C function*), 3132
sys_cache_data_enable (*C function*), 3132
sys_cache_data_flush_all (*C function*), 3132
sys_cache_data_flush_and_invd_all (*C function*), 3133
sys_cache_data_flush_and_invd_range (*C function*), 3135
sys_cache_data_flush_range (*C function*), 3133
sys_cache_data_invd_all (*C function*), 3132
sys_cache_data_invd_range (*C function*), 3134
sys_cache_data_line_size_get (*C function*), 3136
sys_cache_instr_disable (*C function*), 3132
sys_cache_instr_enable (*C function*), 3132
sys_cache_instr_flush_all (*C function*), 3132
sys_cache_instr_flush_and_invd_all (*C function*), 3133
sys_cache_instr_flush_and_invd_range (*C function*), 3135
sys_cache_instr_flush_range (*C function*), 3134
sys_cache_instr_invd_all (*C function*), 3133
sys_cache_instr_invd_range (*C function*), 3134
sys_cache_instr_line_size_get (*C function*), 3136
sys_cache_is_ptr_cached (*C function*), 3136
sys_cache_is_ptr_uncached (*C function*), 3137
sys_cache_uncached_ptr_get (*C function*), 3137
sys_clock_announce (*C function*), 478
sys_clock_cycle_get_32 (*C function*), 478
sys_clock_cycle_get_64 (*C function*), 479
sys_clock_disable (*C function*), 478
sys_clock_elapsed (*C function*), 478
SYS_CLOCK_HW_CYCLES_TO_NS_AVG (*C macro*), 477
sys_clock_idle_exit (*C function*), 478
sys_clock_set_timeout (*C function*), 477
sys_clock_tick_get (*C function*), 481

sys_clock_tick_get_32 (*C function*), 481
sys_clock_timeout_end_calc (*C function*), 482
sys_csrand_get (*C function*), 718
sys_dlist_append (*C function*), 626
SYS_DLIST_CONTAINER (*C macro*), 622
SYS_DLIST_FOR_EACH_CONTAINER (*C macro*), 623
SYS_DLIST_FOR_EACH_CONTAINER_SAFE (*C macro*), 623
SYS_DLIST_FOR_EACH_NODE (*C macro*), 622
SYS_DLIST_FOR_EACH_NODE_SAFE (*C macro*), 622
sys_dlist_get (*C function*), 627
sys_dlist_has_multiple_nodes (*C function*), 625
sys_dlist_init (*C function*), 624
sys_dlist_insert (*C function*), 626
sys_dlist_insert_at (*C function*), 627
sys_dlist_is_empty (*C function*), 625
sys_dlist_is_head (*C function*), 624
sys_dlist_is_tail (*C function*), 624
SYS_DLIST_ITERATE_FROM_NODE (*C macro*), 622
sys_dlist_len (*C function*), 627
sys_dlist_peek_head (*C function*), 625
SYS_DLIST_PEEK_HEAD_CONTAINER (*C macro*), 623
sys_dlist_peek_head_not_empty (*C function*), 625
sys_dlist_peek_next (*C function*), 625
SYS_DLIST_PEEK_NEXT_CONTAINER (*C macro*), 623
sys_dlist_peek_next_no_check (*C function*), 625
sys_dlist_peek_prev (*C function*), 626
sys_dlist_peek_prev_no_check (*C function*), 626
sys_dlist_peek_tail (*C function*), 626
sys_dlist_prepend (*C function*), 626
sys_dlist_remove (*C function*), 627
SYS_DLIST_STATIC_INIT (*C macro*), 624
sys_dlist_t (*C type*), 624
sys_dnode_init (*C function*), 624
sys_dnode_is_linked (*C function*), 624
sys_dnode_t (*C type*), 624
sys_heap_aligned_alloc (*C function*), 567
sys_heap_aligned_realloc (*C function*), 568
sys_heap_alloc (*C function*), 567
sys_heap_free (*C function*), 567
sys_heap_init (*C function*), 566
sys_heap_print_info (*C function*), 569
sys_heap_realloc (*C macro*), 566
sys_heap_stress (*C function*), 569
sys_heap_usable_size (*C function*), 568
sys_heap_validate (*C function*), 568
SYS_KERNEL_VER_MAJOR (*C macro*), 506
SYS_KERNEL_VER_MINOR (*C macro*), 506
SYS_KERNEL_VER_PATCHLEVEL (*C macro*), 506
sys_kernel_version_get (*C function*), 506
sys_mem_blocks_alloc (*C function*), 589
sys_mem_blocks_alloc_contiguous (*C function*), 590
SYS_MEM_BLOCKS_DEFINE (*C macro*), 588
SYS_MEM_BLOCKS_DEFINE_STATIC (*C macro*), 588
SYS_MEM_BLOCKS_DEFINE_STATIC_WITH_EXT_BUF (*C macro*), 589
SYS_MEM_BLOCKS_DEFINE_WITH_EXT_BUF (*C macro*), 588
sys_mem_blocks_free (*C function*), 591
sys_mem_blocks_free_contiguous (*C function*), 591
sys_mem_blocks_get (*C function*), 590

sys_mem_blocks_is_region_free (*C function*), 590
sys_mem_blocks_t (*C type*), 589
sys_multi_heap (*C struct*), 572
sys_multi_heap_add_heap (*C function*), 570
sys_multi_heap_aligned_alloc (*C function*), 571
sys_multi_heap_alloc (*C function*), 571
sys_multi_heap_fn_t (*C type*), 569
sys_multi_heap_free (*C function*), 571
sys_multi_heap_get_heap (*C function*), 571
sys_multi_heap_init (*C function*), 570
sys_multi_heap_rec (*C struct*), 572
sys_multi_mem_blocks_add_allocator (*C function*), 591
sys_multi_mem_blocks_alloc (*C function*), 592
sys_multi_mem_blocks_choice_fn_t (*C type*), 589
sys_multi_mem_blocks_free (*C function*), 592
sys_multi_mem_blocks_init (*C function*), 591
sys_multi_mem_blocks_t (*C type*), 589
SYS_MUTEX_DEFINE (*C macro*), 409
sys_mutex_init (*C function*), 409
sys_mutex_lock (*C function*), 409
sys_mutex_unlock (*C function*), 410
sys_notify (*C struct*), 1040
sys_notify_fetch_result (*C function*), 1039
sys_notify_finalize (*C function*), 1039
sys_notify_generic_callback (*C type*), 1038
sys_notify_get_method (*C function*), 1038
sys_notify_init_callback (*C function*), 1040
sys_notify_init_signal (*C function*), 1039
sys_notify_init_spinwait (*C function*), 1039
sys_notify_uses_callback (*C function*), 1040
sys_notify_validate (*C function*), 1038
sys_notify.method (*C union*), 1041
sys_notify.method.callback (*C var*), 1041
sys_notify.method.signal (*C var*), 1041
sys_port_trace_k_condvar_broadcast_enter (*C macro*), 981
sys_port_trace_k_condvar_broadcast_exit (*C macro*), 981
sys_port_trace_k_condvar_init (*C macro*), 980
sys_port_trace_k_condvar_signal_blocking (*C macro*), 981
sys_port_trace_k_condvar_signal_enter (*C macro*), 980
sys_port_trace_k_condvar_signal_exit (*C macro*), 981
sys_port_trace_k_condvar_wait_enter (*C macro*), 981
sys_port_trace_k_condvar_wait_exit (*C macro*), 981
sys_port_trace_k_fifo_alloc_put_enter (*C macro*), 986
sys_port_trace_k_fifo_alloc_put_exit (*C macro*), 986
sys_port_trace_k_fifo_alloc_put_slist_enter (*C macro*), 986
sys_port_trace_k_fifo_alloc_put_slist_exit (*C macro*), 987
sys_port_trace_k_fifo_cancel_wait_enter (*C macro*), 985
sys_port_trace_k_fifo_cancel_wait_exit (*C macro*), 985
sys_port_trace_k_fifo_get_enter (*C macro*), 987
sys_port_trace_k_fifo_get_exit (*C macro*), 987
sys_port_trace_k_fifo_init_enter (*C macro*), 985
sys_port_trace_k_fifo_init_exit (*C macro*), 985
sys_port_trace_k_fifo_peek_head_enter (*C macro*), 987
sys_port_trace_k_fifo_peek_head_exit (*C macro*), 987
sys_port_trace_k_fifo_peek_tail_enter (*C macro*), 987
sys_port_trace_k_fifo_peek_tail_exit (*C macro*), 987
sys_port_trace_k_fifo_put_enter (*C macro*), 986
sys_port_trace_k_fifo_put_exit (*C macro*), 986

sys_port_trace_k_fifo_put_list_enter (*C macro*), 986
sys_port_trace_k_fifo_put_list_exit (*C macro*), 986
sys_port_trace_k_heap_aligned_alloc_blocking (*C macro*), 996
sys_port_trace_k_heap_aligned_alloc_enter (*C macro*), 996
sys_port_trace_k_heap_aligned_alloc_exit (*C macro*), 996
sys_port_trace_k_heap_alloc_enter (*C macro*), 996
sys_port_trace_k_heap_alloc_exit (*C macro*), 996
sys_port_trace_k_heap_free (*C macro*), 996
sys_port_trace_k_heap_init (*C macro*), 996
sys_port_trace_k_heap_realloc_enter (*C macro*), 996
sys_port_trace_k_heap_realloc_exit (*C macro*), 997
sys_port_trace_k_heap_sys_k_aligned_alloc_enter (*C macro*), 997
sys_port_trace_k_heap_sys_k_aligned_alloc_exit (*C macro*), 997
sys_port_trace_k_heap_sys_k_calloc_enter (*C macro*), 997
sys_port_trace_k_heap_sys_k_calloc_exit (*C macro*), 998
sys_port_trace_k_heap_sys_k_free_enter (*C macro*), 997
sys_port_trace_k_heap_sys_k_free_exit (*C macro*), 997
sys_port_trace_k_heap_sys_k_malloc_enter (*C macro*), 997
sys_port_trace_k_heap_sys_k_malloc_exit (*C macro*), 997
sys_port_trace_k_heap_sys_k_realloc_enter (*C macro*), 998
sys_port_trace_k_heap_sys_k_realloc_exit (*C macro*), 998
sys_port_trace_k_lifo_alloc_put_enter (*C macro*), 988
sys_port_trace_k_lifo_alloc_put_exit (*C macro*), 988
sys_port_trace_k_lifo_get_enter (*C macro*), 988
sys_port_trace_k_lifo_get_exit (*C macro*), 988
sys_port_trace_k_lifo_init_enter (*C macro*), 988
sys_port_trace_k_lifo_init_exit (*C macro*), 988
sys_port_trace_k_lifo_put_enter (*C macro*), 988
sys_port_trace_k_lifo_put_exit (*C macro*), 988
sys_port_trace_k_mbox_async_put_enter (*C macro*), 993
sys_port_trace_k_mbox_async_put_exit (*C macro*), 993
sys_port_trace_k_mbox_data_get (*C macro*), 993
sys_port_trace_k_mbox_get_blocking (*C macro*), 993
sys_port_trace_k_mbox_get_enter (*C macro*), 993
sys_port_trace_k_mbox_get_exit (*C macro*), 993
sys_port_trace_k_mbox_init (*C macro*), 992
sys_port_trace_k_mbox_message_put_blocking (*C macro*), 992
sys_port_trace_k_mbox_message_put_enter (*C macro*), 992
sys_port_trace_k_mbox_message_put_exit (*C macro*), 992
sys_port_trace_k_mbox_put_enter (*C macro*), 992
sys_port_trace_k_mbox_put_exit (*C macro*), 992
sys_port_trace_k_mem_slab_alloc_blocking (*C macro*), 998
sys_port_trace_k_mem_slab_alloc_enter (*C macro*), 998
sys_port_trace_k_mem_slab_alloc_exit (*C macro*), 999
sys_port_trace_k_mem_slab_free_enter (*C macro*), 999
sys_port_trace_k_mem_slab_free_exit (*C macro*), 999
sys_port_trace_k_mem_slab_init (*C macro*), 998
sys_port_trace_k_msgq_alloc_init_enter (*C macro*), 990
sys_port_trace_k_msgq_alloc_init_exit (*C macro*), 990
sys_port_trace_k_msgq_cleanup_enter (*C macro*), 990
sys_port_trace_k_msgq_cleanup_exit (*C macro*), 990
sys_port_trace_k_msgq_get_blocking (*C macro*), 991
sys_port_trace_k_msgq_get_enter (*C macro*), 991
sys_port_trace_k_msgq_get_exit (*C macro*), 991
sys_port_trace_k_msgq_init (*C macro*), 990
sys_port_trace_k_msgq_peek (*C macro*), 991
sys_port_trace_k_msgq_purge (*C macro*), 992
sys_port_trace_k_msgq_put_blocking (*C macro*), 991

sys_port_trace_k_msgq_put_enter (*C macro*), 991
sys_port_trace_k_msgq_put_exit (*C macro*), 991
sys_port_trace_k_mutex_init (*C macro*), 980
sys_port_trace_k_mutex_lock_blocking (*C macro*), 980
sys_port_trace_k_mutex_lock_enter (*C macro*), 980
sys_port_trace_k_mutex_lock_exit (*C macro*), 980
sys_port_trace_k_mutex_unlock_enter (*C macro*), 980
sys_port_trace_k_mutex_unlock_exit (*C macro*), 980
sys_port_trace_k_pipe_alloc_init_enter (*C macro*), 994
sys_port_trace_k_pipe_alloc_init_exit (*C macro*), 994
sys_port_trace_k_pipe_buffer_flush_enter (*C macro*), 994
sys_port_trace_k_pipe_buffer_flush_exit (*C macro*), 994
sys_port_trace_k_pipe_cleanup_enter (*C macro*), 994
sys_port_trace_k_pipe_cleanup_exit (*C macro*), 994
sys_port_trace_k_pipe_flush_enter (*C macro*), 994
sys_port_trace_k_pipe_flush_exit (*C macro*), 994
sys_port_trace_k_pipe_get_blocking (*C macro*), 995
sys_port_trace_k_pipe_get_enter (*C macro*), 995
sys_port_trace_k_pipe_get_exit (*C macro*), 995
sys_port_trace_k_pipe_init (*C macro*), 994
sys_port_trace_k_pipe_put_blocking (*C macro*), 995
sys_port_trace_k_pipe_put_enter (*C macro*), 994
sys_port_trace_k_pipe_put_exit (*C macro*), 995
sys_port_trace_k_poll_api_event_init (*C macro*), 978
sys_port_trace_k_poll_api_poll_enter (*C macro*), 978
sys_port_trace_k_poll_api_poll_exit (*C macro*), 978
sys_port_trace_k_poll_api_signal_check (*C macro*), 978
sys_port_trace_k_poll_api_signal_init (*C macro*), 978
sys_port_trace_k_poll_api_signal_raise (*C macro*), 978
sys_port_trace_k_poll_api_signal_reset (*C macro*), 978
sys_port_trace_k_queue_alloc_append_enter (*C macro*), 982
sys_port_trace_k_queue_alloc_append_exit (*C macro*), 982
sys_port_trace_k_queue_alloc_prepend_enter (*C macro*), 983
sys_port_trace_k_queue_alloc_prepend_exit (*C macro*), 983
sys_port_trace_k_queue_append_enter (*C macro*), 982
sys_port_trace_k_queue_append_exit (*C macro*), 982
sys_port_trace_k_queue_append_list_enter (*C macro*), 983
sys_port_trace_k_queue_append_list_exit (*C macro*), 983
sys_port_trace_k_queue_cancel_wait (*C macro*), 982
sys_port_trace_k_queue_get_blocking (*C macro*), 984
sys_port_trace_k_queue_get_enter (*C macro*), 984
sys_port_trace_k_queue_get_exit (*C macro*), 984
sys_port_trace_k_queue_init (*C macro*), 982
sys_port_trace_k_queue_insert_blocking (*C macro*), 983
sys_port_trace_k_queue_insert_enter (*C macro*), 983
sys_port_trace_k_queue_insert_exit (*C macro*), 983
sys_port_trace_k_queue_merge_slist_enter (*C macro*), 984
sys_port_trace_k_queue_merge_slist_exit (*C macro*), 984
sys_port_trace_k_queue_peek_head (*C macro*), 985
sys_port_trace_k_queue_peek_tail (*C macro*), 985
sys_port_trace_k_queue_prepend_enter (*C macro*), 983
sys_port_trace_k_queue_prepend_exit (*C macro*), 983
sys_port_trace_k_queue_queue_insert_blocking (*C macro*), 982
sys_port_trace_k_queue_queue_insert_enter (*C macro*), 982
sys_port_trace_k_queue_queue_insert_exit (*C macro*), 982
sys_port_trace_k_queue_remove_enter (*C macro*), 984
sys_port_trace_k_queue_remove_exit (*C macro*), 984
sys_port_trace_k_queue_unique_append_enter (*C macro*), 984

sys_port_trace_k_queue_unique_append_exit (*C macro*), 985
sys_port_trace_k_sem_give_enter (*C macro*), 979
sys_port_trace_k_sem_give_exit (*C macro*), 979
sys_port_trace_k_sem_init (*C macro*), 979
sys_port_trace_k_sem_reset (*C macro*), 979
sys_port_trace_k_sem_take_blocking (*C macro*), 979
sys_port_trace_k_sem_take_enter (*C macro*), 979
sys_port_trace_k_sem_take_exit (*C macro*), 979
sys_port_trace_k_stack_alloc_init_enter (*C macro*), 989
sys_port_trace_k_stack_alloc_init_exit (*C macro*), 989
sys_port_trace_k_stack_cleanup_enter (*C macro*), 989
sys_port_trace_k_stack_cleanup_exit (*C macro*), 989
sys_port_trace_k_stack_init (*C macro*), 989
sys_port_trace_k_stack_pop_blocking (*C macro*), 990
sys_port_trace_k_stack_pop_enter (*C macro*), 989
sys_port_trace_k_stack_pop_exit (*C macro*), 990
sys_port_trace_k_stack_push_enter (*C macro*), 989
sys_port_trace_k_stack_push_exit (*C macro*), 989
sys_port_trace_k_thread_abort (*C macro*), 973
sys_port_trace_k_thread_abort_enter (*C macro*), 974
sys_port_trace_k_thread_abort_exit (*C macro*), 974
sys_port_trace_k_thread_busy_wait_enter (*C macro*), 973
sys_port_trace_k_thread_busy_wait_exit (*C macro*), 973
sys_port_trace_k_thread_create (*C macro*), 972
sys_port_trace_k_thread_foreach_enter (*C macro*), 972
sys_port_trace_k_thread_foreach_exit (*C macro*), 972
sys_port_trace_k_thread_foreach_unlocked_enter (*C macro*), 972
sys_port_trace_k_thread_foreach_unlocked_exit (*C macro*), 972
sys_port_trace_k_thread_info (*C macro*), 975
sys_port_trace_k_thread_join_blocking (*C macro*), 972
sys_port_trace_k_thread_join_enter (*C macro*), 972
sys_port_trace_k_thread_join_exit (*C macro*), 972
sys_port_trace_k_thread_msleep_enter (*C macro*), 973
sys_port_trace_k_thread_msleep_exit (*C macro*), 973
sys_port_trace_k_thread_name_set (*C macro*), 974
sys_port_trace_k_thread_pend (*C macro*), 975
sys_port_trace_k_thread_priority_set (*C macro*), 974
sys_port_trace_k_thread_ready (*C macro*), 975
sys_port_trace_k_thread_resume_enter (*C macro*), 974
sys_port_trace_k_thread_resume_exit (*C macro*), 974
sys_port_trace_k_thread_sched_abort (*C macro*), 975
sys_port_trace_k_thread_sched_lock (*C macro*), 974
sys_port_trace_k_thread_sched_pend (*C macro*), 975
sys_port_trace_k_thread_sched_priority_set (*C macro*), 975
sys_port_trace_k_thread_sched_ready (*C macro*), 975
sys_port_trace_k_thread_sched_resume (*C macro*), 975
sys_port_trace_k_thread_sched_suspend (*C macro*), 975
sys_port_trace_k_thread_sched_unlock (*C macro*), 974
sys_port_trace_k_thread_sched_wakeup (*C macro*), 975
sys_port_trace_k_thread_sleep_enter (*C macro*), 972
sys_port_trace_k_thread_sleep_exit (*C macro*), 972
sys_port_trace_k_thread_start (*C macro*), 973
sys_port_trace_k_thread_suspend_enter (*C macro*), 974
sys_port_trace_k_thread_suspend_exit (*C macro*), 974
sys_port_trace_k_thread_switched_in (*C macro*), 975
sys_port_trace_k_thread_switched_out (*C macro*), 974
sys_port_trace_k_thread_user_mode_enter (*C macro*), 972
sys_port_trace_k_thread_usleep_enter (*C macro*), 973

sys_port_trace_k_thread_usleep_exit (*C macro*), 973
sys_port_trace_k_thread_wakeup (*C macro*), 973
sys_port_trace_k_thread_yield (*C macro*), 973
sys_port_trace_k_timer_init (*C macro*), 999
sys_port_trace_k_timer_start (*C macro*), 999
sys_port_trace_k_timer_status_sync_blocking (*C macro*), 1000
sys_port_trace_k_timer_status_sync_enter (*C macro*), 999
sys_port_trace_k_timer_status_sync_exit (*C macro*), 1000
sys_port_trace_k_timer_stop (*C macro*), 999
sys_port_trace_k_work_cancel_enter (*C macro*), 977
sys_port_trace_k_work_cancel_exit (*C macro*), 977
sys_port_trace_k_work_cancel_sync_blocking (*C macro*), 977
sys_port_trace_k_work_cancel_sync_enter (*C macro*), 977
sys_port_trace_k_work_cancel_sync_exit (*C macro*), 977
sys_port_trace_k_work_flush_blocking (*C macro*), 976
sys_port_trace_k_work_flush_enter (*C macro*), 976
sys_port_trace_k_work_flush_exit (*C macro*), 977
sys_port_trace_k_work_init (*C macro*), 976
sys_port_trace_k_work_submit_enter (*C macro*), 976
sys_port_trace_k_work_submit_exit (*C macro*), 976
sys_port_trace_k_work_submit_to_queue_enter (*C macro*), 976
sys_port_trace_k_work_submit_to_queue_exit (*C macro*), 976
sys_port_trace_syscall_enter (*C macro*), 1001
sys_port_trace_syscall_exit (*C macro*), 1001
SYS_PORT_TRACK_NEXT (*C macro*), 1000
sys_poweroff (*C function*), 1111
sys_rand8_get (*C function*), 719
sys_rand16_get (*C function*), 719
sys_rand32_get (*C function*), 719
sys_rand64_get (*C function*), 719
sys_rand_get (*C function*), 718
sys_sem_count_get (*C function*), 404
SYS_SEM_DEFINE (*C macro*), 403
sys_sem_give (*C function*), 403
sys_sem_init (*C function*), 403
sys_sem_take (*C function*), 403
sys_sflist_append (*C function*), 618
sys_sflist_append_list (*C function*), 618
SYS_SFLIST_CONTAINER (*C macro*), 614
sys_sflist_find_and_remove (*C function*), 619
SYS_SFLIST_FLAGS_MASK (*C macro*), 616
SYS_SFLIST_FOR_EACH_CONTAINER (*C macro*), 615
SYS_SFLIST_FOR_EACH_CONTAINER_SAFE (*C macro*), 615
SYS_SFLIST_FOR_EACH_NODE (*C macro*), 613
SYS_SFLIST_FOR_EACH_NODE_SAFE (*C macro*), 614
sys_sflist_get (*C function*), 618
sys_sflist_get_not_empty (*C function*), 618
sys_sflist_init (*C function*), 616
sys_sflist_insert (*C function*), 618
sys_sflist_is_empty (*C function*), 617
SYS_SFLIST_ITERATE_FROM_NODE (*C macro*), 614
sys_sflist_len (*C function*), 619
sys_sflist_merge_sflist (*C function*), 618
sys_sflist_peek_head (*C function*), 616
SYS_SFLIST_PEEK_HEAD_CONTAINER (*C macro*), 614
sys_sflist_peek_next (*C function*), 617
SYS_SFLIST_PEEK_NEXT_CONTAINER (*C macro*), 615
sys_sflist_peek_next_no_check (*C function*), 617

sys_sflist_peek_tail (*C function*), 616
SYS_SFLIST_PEEK_TAIL_CONTAINER (*C macro*), 615
sys_sflist_prepend (*C function*), 617
sys_sflist_remove (*C function*), 619
SYS_SFLIST_STATIC_INIT (*C macro*), 615
sys_sflist_t (*C type*), 616
sys_sfnode_flags_get (*C function*), 616
sys_sfnode_flags_set (*C function*), 617
sys_sfnode_init (*C function*), 616
sys_sfnode_t (*C type*), 616
sys_slist_append (*C function*), 611
sys_slist_append_list (*C function*), 611
SYS_SLIST_CONTAINER (*C macro*), 609
sys_slist_find (*C function*), 613
sys_slist_find_and_remove (*C function*), 613
SYS_SLIST_FOR_EACH_CONTAINER (*C macro*), 609
SYS_SLIST_FOR_EACH_CONTAINER_SAFE (*C macro*), 610
SYS_SLIST_FOR_EACH_NODE (*C macro*), 608
SYS_SLIST_FOR_EACH_NODE_SAFE (*C macro*), 608
sys_slist_get (*C function*), 612
sys_slist_get_not_empty (*C function*), 612
sys_slist_init (*C function*), 610
sys_slist_insert (*C function*), 612
sys_slist_is_empty (*C function*), 611
SYS_SLIST_ITERATE_FROM_NODE (*C macro*), 608
sys_slist_len (*C function*), 613
sys_slist_merge_slist (*C function*), 612
sys_slist_peek_head (*C function*), 610
SYS_SLIST_PEEK_HEAD_CONTAINER (*C macro*), 609
sys_slist_peek_next (*C function*), 611
SYS_SLIST_PEEK_NEXT_CONTAINER (*C macro*), 609
sys_slist_peek_next_no_check (*C function*), 611
sys_slist_peek_tail (*C function*), 610
SYS_SLIST_PEEK_TAIL_CONTAINER (*C macro*), 609
sys_slist_prepend (*C function*), 611
sys_slist_remove (*C function*), 612
SYS_SLIST_STATIC_INIT (*C macro*), 610
sys_slist_t (*C type*), 610
sys_snode_t (*C type*), 610
sys_timepoint_calc (*C function*), 481
sys_timepoint_cmp (*C function*), 482
sys_timepoint_expired (*C function*), 482
sys_timepoint_timeout (*C function*), 481
sys_trace_idle (*C function*), 971
sys_trace_isr_enter (*C function*), 971
sys_trace_isr_exit (*C function*), 971
sys_trace_isr_exit_to_scheduler (*C function*), 971
sys_trace_sys_init_enter (*C macro*), 971
sys_trace_sys_init_exit (*C macro*), 971
sysbuild_conf (*runners.core.ZephyrBinaryRunner property*), 201
SysbuildConfiguration (*class in runners.core*), 197
system power state, **3937**

T

T32_DIR, 98
task_wdt_add (*C function*), 1221
task_wdt_callback_t (*C type*), 1220
task_wdt_delete (*C function*), 1221

task_wdt_feed (*C function*), 1221
task_wdt_init (*C function*), 1220
tc_cable_plug (*C enum*), 3660
tc_cable_plug.PD_PLUG_FROM_CABLE_VPD (*C enumerator*), 3660
tc_cable_plug.PD_PLUG_FROM_DFP_UFP (*C enumerator*), 3660
tc_cc_polarity (*C enum*), 3661
tc_cc_polarity.TC_POLARITY_CC1 (*C enumerator*), 3661
tc_cc_polarity.TC_POLARITY_CC2 (*C enumerator*), 3661
tc_cc_pull (*C enum*), 3660
tc_cc_pull.TC_CC_OPEN (*C enumerator*), 3660
tc_cc_pull.TC_CC_RA (*C enumerator*), 3660
tc_cc_pull.TC_CC_RD (*C enumerator*), 3660
tc_cc_pull.TC_CC_RP (*C enumerator*), 3660
tc_cc_pull.TC_RA_RD (*C enumerator*), 3660
tc_cc_states (*C enum*), 3661
tc_cc_states.TC_CC_DFP_ATTACHED (*C enumerator*), 3662
tc_cc_states.TC_CC_DFP_DEBUG_ACC (*C enumerator*), 3662
tc_cc_states.TC_CC_NONE (*C enumerator*), 3661
tc_cc_states.TC_CC_UFP_ATTACHED (*C enumerator*), 3662
tc_cc_states.TC_CC_UFP_AUDIO_ACC (*C enumerator*), 3661
tc_cc_states.TC_CC_UFP_DEBUG_ACC (*C enumerator*), 3661
tc_cc_states.TC_CC_UFP_NONE (*C enumerator*), 3661
tc_cc_voltage_state (*C enum*), 3659
tc_cc_voltage_state.TC_CC_VOLT_OPEN (*C enumerator*), 3659
tc_cc_voltage_state.TC_CC_VOLT_RA (*C enumerator*), 3659
tc_cc_voltage_state.TC_CC_VOLT_RD (*C enumerator*), 3659
tc_cc_voltage_state.TC_CC_VOLT_RP_1A5 (*C enumerator*), 3659
tc_cc_voltage_state.TC_CC_VOLT_RP_3A0 (*C enumerator*), 3659
tc_cc_voltage_state.TC_CC_VOLT_RP_DEF (*C enumerator*), 3659
tc_data_role (*C enum*), 3661
tc_data_role.TC_ROLE_DFP (*C enumerator*), 3661
tc_data_role.TC_ROLE_DISCONNECTED (*C enumerator*), 3661
tc_data_role.TC_ROLE_UFP (*C enumerator*), 3661
tc_power_role (*C enum*), 3660
tc_power_role.TC_ROLE_SINK (*C enumerator*), 3661
tc_power_role.TC_ROLE_SOURCE (*C enumerator*), 3661
tc_rp_value (*C enum*), 3660
tc_rp_value.TC_RP_1A5 (*C enumerator*), 3660
tc_rp_value.TC_RP_3A0 (*C enumerator*), 3660
tc_rp_value.TC_RP_RESERVED (*C enumerator*), 3660
tc_rp_value.TC_RP_USB (*C enumerator*), 3660
TC_T_CC_DEBOUNCE_MAX_MS (*C macro*), 3657
TC_T_CC_DEBOUNCE_MIN_MS (*C macro*), 3657
TC_T_DRP_MAX_MS (*C macro*), 3656
TC_T_DRP_MIN_MS (*C macro*), 3656
TC_T_DRP_TRANSITION_MAX_MS (*C macro*), 3656
TC_T_DRP_TRANSITION_MIN_MS (*C macro*), 3656
TC_T_DRP_TRY_MAX_MS (*C macro*), 3656
TC_T_DRP_TRY_MIN_MS (*C macro*), 3656
TC_T_DRP_TRY_WAIT_MAX_MS (*C macro*), 3656
TC_T_DRP_TRY_WAIT_MIN_MS (*C macro*), 3656
TC_T_ERROR_RECOVERY_SELF_POWERED_MIN_MS (*C macro*), 3657
TC_T_ERROR_RECOVERY_SOURCE_MIN_MS (*C macro*), 3657
TC_T_NO_TOGGLE_CONNECT_MAX_MS (*C macro*), 3658
TC_T_NO_TOGGLE_CONNECT_MIN_MS (*C macro*), 3658
TC_T_ONE_PORT_TOGGLE_CONNECT_MAX_MS (*C macro*), 3658
TC_T_ONE_PORT_TOGGLE_CONNECT_MIN_MS (*C macro*), 3658
TC_T_PD_DEBOUNCE_MAX_MS (*C macro*), 3657

TC_T_PD_DEBOUNCE_MIN_MS (*C macro*), 3657
TC_T_RP_VALUE_CHANGE_MAX_MS (*C macro*), 3657
TC_T_RP_VALUE_CHANGE_MIN_MS (*C macro*), 3657
TC_T_SINK_ADJ_MAX_MS (*C macro*), 3655
TC_T_SRC_DISCONNECT_MAX_MS (*C macro*), 3658
TC_T_SRC_DISCONNECT_MIN_MS (*C macro*), 3658
TC_T_TRY_CC_DEBOUNCE_MAX_MS (*C macro*), 3657
TC_T_TRY_CC_DEBOUNCE_MIN_MS (*C macro*), 3657
TC_T_TRY_TIMEOUT_MAX_MS (*C macro*), 3656
TC_T_TRY_TIMEOUT_MIN_MS (*C macro*), 3656
TC_T_TWO_PORT_TOGGLE_CONNECT_MAX_MS (*C macro*), 3658
TC_T_TWO_PORT_TOGGLE_CONNECT_MIN_MS (*C macro*), 3658
TC_T_VBUS_OFF_MAX_MS (*C macro*), 3655
TC_T_VBUS_ON_MAX_MS (*C macro*), 3655
TC_T_VCONN_OFF_MAX_MS (*C macro*), 3655
TC_T_VCONN_ON_MAX_MS (*C macro*), 3655
TC_T_VCONN_ON_PA_MAX_MS (*C macro*), 3655
TC_T_VPD_DETACH_MAX_MS (*C macro*), 3657
TC_T_VPD_DETACH_MIN_MS (*C macro*), 3656
TC_T_VPDCTDD_MAX_MS (*C macro*), 3659
TC_T_VPDCTDD_MIN_US (*C macro*), 3658
TC_T_VPDDISABLE_MIN_MS (*C macro*), 3659
TC_V_SINK_DISCONNECT_MAX_MV (*C macro*), 3655
TC_V_SINK_DISCONNECT_MIN_MV (*C macro*), 3655
tc_vbus_level (*C enum*), 3659
tc_vbus_level.TC_VBUS_PRESENT (*C enumerator*), 3659
tc_vbus_level.TC_VBUS_REMOVED (*C enumerator*), 3659
tc_vbus_level.TC_VBUS_SAFE0V (*C enumerator*), 3659
TCP_KEEPCNT (*C macro*), 2495
TCP_KEEPIDLE (*C macro*), 2495
TCP_KEEPINTVL (*C macro*), 2495
TCP_NODELAY (*C macro*), 2495
tcpc_alert (*C enum*), 3662
tcpc_alert_handler_cb_t (*C type*), 3662
tcpc_alert.TCPC_ALERT_BEGINNING_MSG_STATUS (*C enumerator*), 3663
tcpc_alert.TCPC_ALERT_CC_STATUS (*C enumerator*), 3662
tcpc_alert.TCPC_ALERT_EXTENDED (*C enumerator*), 3663
tcpc_alert.TCPC_ALERT_EXTENDED_STATUS (*C enumerator*), 3663
tcpc_alert.TCPC_ALERT_FAULT_STATUS (*C enumerator*), 3663
tcpc_alert.TCPC_ALERT_HARD_RESET_RECEIVED (*C enumerator*), 3662
tcpc_alert.TCPC_ALERT_MSG_STATUS (*C enumerator*), 3662
tcpc_alert.TCPC_ALERT_POWER_STATUS (*C enumerator*), 3662
tcpc_alert.TCPC_ALERT_RX_BUFFER_OVERFLOW (*C enumerator*), 3663
tcpc_alert.TCPC_ALERT_TRANSMIT_MSG_DISCARDED (*C enumerator*), 3663
tcpc_alert.TCPC_ALERT_TRANSMIT_MSG_FAILED (*C enumerator*), 3663
tcpc_alert.TCPC_ALERT_TRANSMIT_MSG_SUCCESS (*C enumerator*), 3663
tcpc_alert.TCPC_ALERT_VBUS_ALARM_HI (*C enumerator*), 3663
tcpc_alert.TCPC_ALERT_VBUS_ALARM_LO (*C enumerator*), 3663
tcpc_alert.TCPC_ALERT_VBUS_SNK_DISCONNECT (*C enumerator*), 3663
tcpc_alert.TCPC_ALERT_VENDOR_DEFINED (*C enumerator*), 3663
tcpc_chip_info (*C struct*), 3672
tcpc_chip_info.device_id (*C var*), 3672
tcpc_chip_info.fw_version_number (*C var*), 3672
tcpc_chip_info.min_req_fw_version_number (*C var*), 3672
tcpc_chip_info.min_req_fw_version_string (*C var*), 3672
tcpc_chip_info.product_id (*C var*), 3672
tcpc_chip_info.vendor_id (*C var*), 3672
tcpc_clear_status_register (*C function*), 3669

tcpc_driver_api (*C struct*), 3672
tcpc_dump_std_reg (*C function*), 3668
tcpc_get_cc (*C function*), 3664
tcpc_get_chip_info (*C function*), 3671
tcpc_get_rp_value (*C function*), 3665
tcpc_get_rx_pending_msg (*C function*), 3667
tcpc_get_snk_ctrl (*C function*), 3670
tcpc_get_src_ctrl (*C function*), 3670
tcpc_get_status_register (*C function*), 3668
tcpc_init (*C function*), 3664
tcpc_is_cc_at_least_one_rd (*C function*), 3664
tcpc_is_cc_audio_acc (*C function*), 3664
tcpc_is_cc_only_one_rd (*C function*), 3664
tcpc_is_cc_open (*C function*), 3664
tcpc_is_cc_rp (*C function*), 3664
tcpc_is_cc_snk_dbg_acc (*C function*), 3664
tcpc_is_cc_src_dbg_acc (*C function*), 3664
tcpc_mask_status_register (*C function*), 3669
tcpc_select_rp_value (*C function*), 3665
tcpc_set_alert_handler_cb (*C function*), 3668
tcpc_set_bist_test_mode (*C function*), 3671
tcpc_set_cc (*C function*), 3665
tcpc_set_cc_polarity (*C function*), 3667
tcpc_set_debug_accessory (*C function*), 3669
tcpc_set_debug_detach (*C function*), 3669
tcpc_set_drp_toggle (*C function*), 3670
tcpc_set_low_power_mode (*C function*), 3671
tcpc_set_roles (*C function*), 3666
tcpc_set_rx_enable (*C function*), 3667
tcpc_set_snk_ctrl (*C function*), 3670
tcpc_set_src_ctrl (*C function*), 3670
tcpc_set_vconn (*C function*), 3666
tcpc_set_vconn_cb (*C function*), 3665
tcpc_set_vconn_discharge_cb (*C function*), 3666
tcpc_sop_prime_enable (*C function*), 3671
tcpc_status_reg (*C enum*), 3663
tcpc_status_reg.TCPC_CC_STATUS (*C enumerator*), 3663
tcpc_status_reg.TCPC_EXTENDED_ALERT_STATUS (*C enumerator*), 3664
tcpc_status_reg.TCPC_EXTENDED_STATUS (*C enumerator*), 3664
tcpc_status_reg.TCPC_FAULT_STATUS (*C enumerator*), 3664
tcpc_status_reg.TCPC_POWER_STATUS (*C enumerator*), 3663
tcpc_status_reg.TCPC_VENDOR_DEFINED_STATUS (*C enumerator*), 3664
tcpc_transmit_data (*C function*), 3668
tcpc_vconn_control_cb_t (*C type*), 3662
tcpc_vconn_discharge (*C function*), 3666
tcpc_vconn_discharge_cb_t (*C type*), 3662
TFTP_BLOCK_SIZE (*C macro*), 2883
tftp_callback_t (*C type*), 2883
tftp_data_param (*C struct*), 2884
tftp_data_param.data_ptr (*C var*), 2885
tftp_data_param.len (*C var*), 2885
tftp_error_param (*C struct*), 2885
tftp_error_param.code (*C var*), 2885
tftp_error_param.msg (*C var*), 2885
tftp_evt (*C struct*), 2885
tftp_evt_param (*C union*), 2885
tftp_evt_param.data (*C var*), 2885
tftp_evt_param.error (*C var*), 2885

tftp_evt_type (*C enum*), 2883
tftp_evt_type.TFTP_EVT_DATA (*C enumerator*), 2883
tftp_evt_type.TFTP_EVT_ERROR (*C enumerator*), 2883
tftp_evt.param (*C var*), 2885
tftp_evt.type (*C var*), 2885
tftp_get (*C function*), 2884
TFTP_HEADER_SIZE (*C macro*), 2883
tftp_put (*C function*), 2884
tftpc (*C struct*), 2885
TFTPC_BUFFER_OVERFLOW (*C macro*), 2882
TFTPC_DUPLICATE_DATA (*C macro*), 2882
TFTPC_MAX_BUF_SIZE (*C macro*), 2883
TFTPC_REMOTE_ERROR (*C macro*), 2882
TFTPC_RETRIES_EXHAUSTED (*C macro*), 2882
TFTPC_SUCCESS (*C macro*), 2882
TFTPC_UNKNOWN_FAILURE (*C macro*), 2882
tftpc.callback (*C var*), 2886
tftpc.server (*C var*), 2886
tftpc.tftp_buf (*C var*), 2886
tgpio_pin_config_ext_timestamp (*C function*), 3693
tgpio_pin_disable (*C function*), 3693
tgpio_pin_periodic_output (*C function*), 3693
tgpio_pin_polarity (*C enum*), 3692
tgpio_pin_polarity.TGPIO_FALLING_EDGE (*C enumerator*), 3692
tgpio_pin_polarity.TGPIO_RISING_EDGE (*C enumerator*), 3692
tgpio_pin_polarity.TGPIO_TOGGLE_EDGE (*C enumerator*), 3692
tgpio_pin_read_ts_ec (*C function*), 3693
tgpio_port_get_cycles_per_second (*C function*), 3693
tgpio_port_get_time (*C function*), 3692
thread_analyzer_cb (*C type*), 728
thread_analyzer_info (*C struct*), 728
thread_analyzer_info.name (*C var*), 729
thread_analyzer_info.stack_size (*C var*), 729
thread_analyzer_info.stack_used (*C var*), 729
thread_analyzer_print (*C function*), 728
thread_analyzer_run (*C function*), 728
thread_info_enabled (*runners.core.ZephyrBinaryRunner property*), 201
timeutil_sync_config (*C struct*), 676
timeutil_sync_config.local_Hz (*C var*), 676
timeutil_sync_config.ref_Hz (*C var*), 676
timeutil_sync_estimate_skew (*C function*), 674
timeutil_sync_instant (*C struct*), 676
timeutil_sync_instant.local (*C var*), 676
timeutil_sync_instant.ref (*C var*), 676
timeutil_sync_local_from_ref (*C function*), 675
timeutil_sync_ref_from_local (*C function*), 674
timeutil_sync_skew_to_ppb (*C function*), 675
timeutil_sync_state (*C struct*), 676
timeutil_sync_state_set_skew (*C function*), 674
timeutil_sync_state_update (*C function*), 673
timeutil_sync_state.base (*C var*), 677
timeutil_sync_state.cfg (*C var*), 677
timeutil_sync_state.latest (*C var*), 677
timeutil_sync_state.skew (*C var*), 677
timeutil_timegm (*C function*), 673
timeutil_timegm64 (*C function*), 672
timing_counter_get (*C function*), 652
timing_cycles_get (*C function*), 652

timing_cycles_to_ns (*C function*), 652
timing_cycles_to_ns_avg (*C function*), 652
timing_freq_get (*C function*), 652
timing_freq_get_mhz (*C function*), 652
timing_init (*C function*), 652
timing_start (*C function*), 652
timing_stop (*C function*), 652
TLS_ALPN_LIST (*C macro*), 2482
TLS_CERT_NOCOPY (*C macro*), 2482
TLS_CERT_NOCOPY_NONE (*C macro*), 2484
TLS_CERT_NOCOPY_OPTIONAL (*C macro*), 2484
TLS_CIPHERSUITE_LIST (*C macro*), 2481
TLS_CIPHERSUITE_USED (*C macro*), 2481
tls_credential_add (*C function*), 2506
tls_credential_delete (*C function*), 2506
tls_credential_get (*C function*), 2506
tls_credential_type (*C enum*), 2505
tls_credential_type.TLS_CREDENTIAL_CA_CERTIFICATE (*C enumerator*), 2505
tls_credential_type.TLS_CREDENTIAL_NONE (*C enumerator*), 2505
tls_credential_type.TLS_CREDENTIAL_PRIVATE_KEY (*C enumerator*), 2505
tls_credential_type.TLS_CREDENTIAL_PSK (*C enumerator*), 2505
tls_credential_type.TLS_CREDENTIAL_PSK_ID (*C enumerator*), 2505
tls_credential_type.TLS_CREDENTIAL_SERVER_CERTIFICATE (*C enumerator*), 2505
TLS_DTLS_CID (*C macro*), 2483
TLS_DTLS_CID_DISABLED (*C macro*), 2484
TLS_DTLS_CID_ENABLED (*C macro*), 2484
TLS_DTLS_CID_STATUS (*C macro*), 2483
TLS_DTLS_CID_STATUS_BIDIRECTIONAL (*C macro*), 2485
TLS_DTLS_CID_STATUS_DISABLED (*C macro*), 2484
TLS_DTLS_CID_STATUS_DOWNLINK (*C macro*), 2484
TLS_DTLS_CID_STATUS_UPLINK (*C macro*), 2484
TLS_DTLS_CID_SUPPORTED (*C macro*), 2484
TLS_DTLS_CID_VALUE (*C macro*), 2483
TLS_DTLS_HANDSHAKE_ON_CONNECT (*C macro*), 2483
TLS_DTLS_HANDSHAKE_TIMEOUT_MAX (*C macro*), 2482
TLS_DTLS_HANDSHAKE_TIMEOUT_MIN (*C macro*), 2482
TLS_DTLS_PEER_CID_VALUE (*C macro*), 2483
TLS_DTLS_ROLE (*C macro*), 2482
TLS_DTLS_ROLE_CLIENT (*C macro*), 2484
TLS_DTLS_ROLE_SERVER (*C macro*), 2484
TLS_HOSTNAME (*C macro*), 2481
TLS_NATIVE (*C macro*), 2482
TLS_PEER_VERIFY (*C macro*), 2482
TLS_PEER_VERIFY_NONE (*C macro*), 2484
TLS_PEER_VERIFY_OPTIONAL (*C macro*), 2484
TLS_PEER_VERIFY_REQUIRED (*C macro*), 2484
TLS_SEC_TAG_LIST (*C macro*), 2481
TLS_SESSION_CACHE (*C macro*), 2483
TLS_SESSION_CACHE_DISABLED (*C macro*), 2484
TLS_SESSION_CACHE_ENABLED (*C macro*), 2484
TLS_SESSION_CACHE_PURGE (*C macro*), 2483
tool_opt_help() (*runners.core.ZephyrBinaryRunner class method*), 201
{TOOLCHAIN}_TOOLCHAIN_PATH, 22
TOOLCHAIN_VER, 289
TYPE_SECTION_COUNT (*C macro*), 703
TYPE_SECTION_END (*C macro*), 702
TYPE_SECTION_END_EXTERN (*C macro*), 702
TYPE_SECTION_FOREACH (*C macro*), 702

TYPE_SECTION_GET (*C macro*), 703
TYPE_SECTION_ITERABLE (*C macro*), 701
TYPE_SECTION_START (*C macro*), 702
TYPE_SECTION_START_EXTERN (*C macro*), 702

U

u8_to_dec (*C function*), 698
UAC2_ENTITY_ID (*C macro*), 3065
uac2_ops (*C struct*), 3066
uac2_ops.buf_release_cb (*C var*), 3067
uac2_ops.data_recv_cb (*C var*), 3067
uac2_ops.feedback_cb (*C var*), 3067
uac2_ops.get_recv_buf (*C var*), 3066
uac2_ops.sof_cb (*C var*), 3066
uac2_ops.terminal_update_cb (*C var*), 3066
uart_callback_set (*C function*), 3647
uart_callback_t (*C type*), 3646
uart_config (*C struct*), 3638
uart_config_data_bits (*C enum*), 3636
uart_config_data_bits.UART_CFG_DATA_BITS_5 (*C enumerator*), 3636
uart_config_data_bits.UART_CFG_DATA_BITS_6 (*C enumerator*), 3636
uart_config_data_bits.UART_CFG_DATA_BITS_7 (*C enumerator*), 3636
uart_config_data_bits.UART_CFG_DATA_BITS_8 (*C enumerator*), 3636
uart_config_data_bits.UART_CFG_DATA_BITS_9 (*C enumerator*), 3636
uart_config_flow_control (*C enum*), 3636
uart_config_flow_control.UART_CFG_FLOW_CTRL_DTR_DSR (*C enumerator*), 3637
uart_config_flow_control.UART_CFG_FLOW_CTRL_NONE (*C enumerator*), 3637
uart_config_flow_control.UART_CFG_FLOW_CTRL_RS485 (*C enumerator*), 3637
uart_config_flow_control.UART_CFG_FLOW_CTRL_RTS_CTS (*C enumerator*), 3637
uart_config_get (*C function*), 3637
uart_config_parity (*C enum*), 3635
uart_config_parity.UART_CFG_PARITY_EVEN (*C enumerator*), 3635
uart_config_parity.UART_CFG_PARITY_MARK (*C enumerator*), 3636
uart_config_parity.UART_CFG_PARITY_NONE (*C enumerator*), 3635
uart_config_parity.UART_CFG_PARITY_ODD (*C enumerator*), 3635
uart_config_parity.UART_CFG_PARITY_SPACE (*C enumerator*), 3636
uart_config_stop_bits (*C enum*), 3636
uart_config_stop_bits.UART_CFG_STOP_BITS_0_5 (*C enumerator*), 3636
uart_config_stop_bits.UART_CFG_STOP_BITS_1 (*C enumerator*), 3636
uart_config_stop_bits.UART_CFG_STOP_BITS_1_5 (*C enumerator*), 3636
uart_config_stop_bits.UART_CFG_STOP_BITS_2 (*C enumerator*), 3636
uart_config.baudrate (*C var*), 3639
uart_config.data_bits (*C var*), 3639
uart_config.flow_ctrl (*C var*), 3639
uart_config.parity (*C var*), 3639
uart_config.stop_bits (*C var*), 3639
uart_configure (*C function*), 3637
uart_drv_cmd (*C function*), 3638
uart_err_check (*C function*), 3637
uart_event (*C struct*), 3652
uart_event_rx (*C struct*), 3651
uart_event_rx_buf (*C struct*), 3651
uart_event_rx_buf.buf (*C var*), 3652
uart_event_rx_stop (*C struct*), 3652
uart_event_rx_stop.data (*C var*), 3652
uart_event_rx_stop.reason (*C var*), 3652
uart_event_rx.buf (*C var*), 3651
uart_event_rx.len (*C var*), 3651

uart_event_rx.offset (*C var*), 3651
uart_event_tx (*C struct*), 3651
uart_event_tx.buf (*C var*), 3651
uart_event_tx.len (*C var*), 3651
uart_event_type (*C enum*), 3646
uart_event_type.UART_RX_BUF_RELEASED (*C enumerator*), 3647
uart_event_type.UART_RX_BUF_REQUEST (*C enumerator*), 3647
uart_event_type.UART_RX_DISABLED (*C enumerator*), 3647
uart_event_type.UART_RX_RDY (*C enumerator*), 3647
uart_event_type.UART_RX_STOPPED (*C enumerator*), 3647
uart_event_type.UART_TX_ABORTED (*C enumerator*), 3647
uart_event_type.UART_TX_DONE (*C enumerator*), 3647
uart_event.type (*C var*), 3652
uart_event.uart_event_data (*C union*), 3652
uart_event.uart_event_data.rx (*C var*), 3652
uart_event.uart_event_data.rx_buf (*C var*), 3652
uart_event.uart_event_data.rx_stop (*C var*), 3652
uart_event.uart_event_data.tx (*C var*), 3652
uart_fifo_fill (*C function*), 3641
uart_fifo_fill_u16 (*C function*), 3641
uart_fifo_read (*C function*), 3642
uart_fifo_read_u16 (*C function*), 3642
uart_irq_callback_set (*C function*), 3645
uart_irq_callback_user_data_set (*C function*), 3645
uart_irq_callback_user_data_t (*C type*), 3641
uart_irq_config_func_t (*C type*), 3641
uart_irq_err_disable (*C function*), 3644
uart_irq_err_enable (*C function*), 3644
uart_irq_is_pending (*C function*), 3644
uart_irq_rx_disable (*C function*), 3643
uart_irq_rx_enable (*C function*), 3643
uart_irq_rx_ready (*C function*), 3643
uart_irq_tx_complete (*C function*), 3643
uart_irq_tx_disable (*C function*), 3643
uart_irq_tx_enable (*C function*), 3642
uart_irq_tx_ready (*C function*), 3643
uart_irq_update (*C function*), 3644
uart_line_ctrl (*C enum*), 3634
uart_line_ctrl_get (*C function*), 3638
uart_line_ctrl_set (*C function*), 3638
uart_line_ctrl.UART_LINE_CTRL_BAUD_RATE (*C enumerator*), 3634
uart_line_ctrl.UART_LINE_CTRL_DCD (*C enumerator*), 3635
uart_line_ctrl.UART_LINE_CTRL_DSR (*C enumerator*), 3635
uart_line_ctrl.UART_LINE_CTRL_DTR (*C enumerator*), 3634
uart_line_ctrl.UART_LINE_CTRL_RTS (*C enumerator*), 3634
uart_poll_in (*C function*), 3639
uart_poll_in_u16 (*C function*), 3639
uart_poll_out (*C function*), 3640
uart_poll_out_u16 (*C function*), 3640
uart_rx_buf_rsp (*C function*), 3650
uart_rx_buf_rsp_u16 (*C function*), 3650
uart_rx_disable (*C function*), 3651
uart_rx_enable (*C function*), 3649
uart_rx_enable_u16 (*C function*), 3649
uart_rx_stop_reason (*C enum*), 3635
uart_rx_stop_reason.UART_BREAK (*C enumerator*), 3635
uart_rx_stop_reason.UART_ERROR_COLLISION (*C enumerator*), 3635
uart_rx_stop_reason.UART_ERROR_FRAMING (*C enumerator*), 3635

uart_rx_stop_reason.UART_ERROR_NOISE (*C enumerator*), 3635
uart_rx_stop_reason.UART_ERROR_OVERRUN (*C enumerator*), 3635
uart_rx_stop_reason.UART_ERROR_PARITY (*C enumerator*), 3635
uart_tx (*C function*), 3648
uart_tx_abort (*C function*), 3649
uart_tx_u16 (*C function*), 3648
UCIFI_OBJECT_BATTERY_ID (*C macro*), 2822
udc_caps (*C function*), 3041
udc_device_speed (*C function*), 3041
udc_disable (*C function*), 3041
udc_enable (*C function*), 3040
udc_ep_buf_alloc (*C function*), 3045
udc_ep_buf_free (*C function*), 3045
udc_ep_buf_set_zlp (*C function*), 3045
udc_ep_clear_halt (*C function*), 3044
udc_ep_dequeue (*C function*), 3044
udc_ep_disable (*C function*), 3043
udc_ep_enable (*C function*), 3043
udc_ep_enqueue (*C function*), 3044
udc_ep_set_halt (*C function*), 3043
udc_ep_try_config (*C function*), 3042
udc_get_buf_info (*C function*), 3045
udc_host_wakeup (*C function*), 3042
udc_init (*C function*), 3040
udc_is_enabled (*C function*), 3040
udc_is_initialized (*C function*), 3040
udc_is_suspended (*C function*), 3040
udc_set_address (*C function*), 3041
udc_shutdown (*C function*), 3041
udc_test_mode (*C function*), 3042
uf2_file (*runners.core.RunnerConfig attribute*), 197
uhc_bus_reset (*C function*), 3070
uhc_bus_resume (*C function*), 3071
uhc_bus_suspend (*C function*), 3071
uhc_caps (*C function*), 3074
uhc_control_stage (*C enum*), 3069
uhc_control_stage.UHC_CONTROL_STAGE_DATA (*C enumerator*), 3069
uhc_control_stage.UHC_CONTROL_STAGE_SETUP (*C enumerator*), 3069
uhc_control_stage.UHC_CONTROL_STAGE_STATUS (*C enumerator*), 3069
uhc_data (*C struct*), 3076
uhc_data.bulk_xfers (*C var*), 3077
uhc_data.caps (*C var*), 3076
uhc_data.ctrl_xfers (*C var*), 3076
uhc_data.event_cb (*C var*), 3077
uhc_data.mutex (*C var*), 3076
uhc_data.priv (*C var*), 3077
uhc_data.status (*C var*), 3077
uhc_device_caps (*C struct*), 3076
uhc_device_caps.hs (*C var*), 3076
uhc_disable (*C function*), 3074
uhc_enable (*C function*), 3074
uhc_ep_dequeue (*C function*), 3073
uhc_ep_enqueue (*C function*), 3073
uhc_event (*C struct*), 3075
uhc_event_cb_t (*C type*), 3069
uhc_event_type (*C enum*), 3069
uhc_event_type.UHC_EVT_DEV_CONNECTED_FS (*C enumerator*), 3069
uhc_event_type.UHC_EVT_DEV_CONNECTED_HS (*C enumerator*), 3069

uhc_event_type.UHC_EVT_DEV_CONNECTED_LS (*C enumerator*), 3069
uhc_event_type.UHC_EVT_DEV_REMOVED (*C enumerator*), 3069
uhc_event_type.UHC_EVT_EP_REQUEST (*C enumerator*), 3070
uhc_event_type.UHC_EVT_ERROR (*C enumerator*), 3070
uhc_event_type.UHC_EVT_RESETED (*C enumerator*), 3070
uhc_event_type.UHC_EVT_RESUMED (*C enumerator*), 3070
uhc_event_type.UHC_EVT_RWUP (*C enumerator*), 3070
uhc_event_type.UHC_EVT_SUSPENDED (*C enumerator*), 3070
uhc_event.dev (*C var*), 3076
uhc_event.node (*C var*), 3076
uhc_event.status (*C var*), 3076
uhc_event.type (*C var*), 3076
uhc_event.xfer (*C var*), 3076
uhc_init (*C function*), 3073
uhc_is_enabled (*C function*), 3070
uhc_is_initialized (*C function*), 3070
uhc_shutdown (*C function*), 3074
uhc_sof_enable (*C function*), 3070
UHC_STATUS_ENABLED (*C macro*), 3069
UHC_STATUS_INITIALIZED (*C macro*), 3069
uhc_transfer (*C struct*), 3074
uhc_transfer.addr (*C var*), 3075
uhc_transfer.attrib (*C var*), 3075
uhc_transfer.buf (*C var*), 3075
uhc_transfer.cb (*C var*), 3075
uhc_transfer.ep (*C var*), 3075
uhc_transfer.err (*C var*), 3075
uhc_transfer.mps (*C var*), 3075
uhc_transfer.node (*C var*), 3075
uhc_transfer.queued (*C var*), 3075
uhc_transfer.setup_pkt (*C var*), 3075
uhc_transfer.stage (*C var*), 3075
uhc_transfer.timeout (*C var*), 3075
uhc_transfer.udev (*C var*), 3075
uhc_xfer_alloc (*C function*), 3071
uhc_xfer_alloc_with_buf (*C function*), 3072
uhc_xfer_buf_add (*C function*), 3072
uhc_xfer_buf_alloc (*C function*), 3072
uhc_xfer_buf_free (*C function*), 3073
uhc_xfer_free (*C function*), 3072
UINT_TO_POINTER (*C macro*), 679
usb_bos_capability_lpm (*C struct*), 3035
usb_bos_capability_msos (*C struct*), 3035
usb_bos_capability_types (*C enum*), 3035
usb_bos_capability_types.USB_BOS_CAPABILITY_EXTENSION (*C enumerator*), 3035
usb_bos_capability_types.USB_BOS_CAPABILITY_PLATFORM (*C enumerator*), 3035
usb_bos_capability_webusb (*C struct*), 3035
usb_bos_descriptor (*C struct*), 3035
usb_bos_platform_descriptor (*C struct*), 3035
usb_bos_register_cap (*C function*), 3031
USB_BSTRING_LENGTH (*C macro*), 3046
usb_cancel_transfer (*C function*), 3031
usb_cancel_transfers (*C function*), 3031
usb_cfg_data (*C struct*), 3032
usb_cfg_data.cb_usb_status (*C var*), 3033
usb_cfg_data.endpoint (*C var*), 3033
usb_cfg_data.interface (*C var*), 3033
usb_cfg_data.interface_config (*C var*), 3033

usb_cfg_data.interface_descriptor (*C var*), 3033
usb_cfg_data.num_endpoints (*C var*), 3033
usb_cfg_data.usb_device_description (*C var*), 3033
usb_dc_attach (*C function*), 3021
usb_dc_detach (*C function*), 3022
usb_dc_ep_callback (*C type*), 3019
usb_dc_ep_cb_status_code (*C enum*), 3020
usb_dc_ep_cb_status_code.USB_DC_EP_DATA_IN (*C enumerator*), 3020
usb_dc_ep_cb_status_code.USB_DC_EP_DATA_OUT (*C enumerator*), 3020
usb_dc_ep_cb_status_code.USB_DC_EP_SETUP (*C enumerator*), 3020
usb_dc_ep_cfg_data (*C struct*), 3026
usb_dc_ep_cfg_data.ep_addr (*C var*), 3026
usb_dc_ep_cfg_data.ep_mps (*C var*), 3026
usb_dc_ep_cfg_data.ep_type (*C var*), 3026
usb_dc_ep_check_cap (*C function*), 3022
usb_dc_ep_clear_stall (*C function*), 3023
usb_dc_ep_configure (*C function*), 3022
usb_dc_ep_disable (*C function*), 3023
usb_dc_ep_enable (*C function*), 3023
usb_dc_ep_flush (*C function*), 3024
usb_dc_ep_halt (*C function*), 3023
usb_dc_ep_is_stalled (*C function*), 3023
usb_dc_ep_mps (*C function*), 3025
usb_dc_ep_read (*C function*), 3024
usb_dc_ep_read_continue (*C function*), 3025
usb_dc_ep_read_wait (*C function*), 3025
usb_dc_ep_set_callback (*C function*), 3024
usb_dc_ep_set_stall (*C function*), 3023
usb_dc_ep_synchronozation_type (*C enum*), 3021
usb_dc_ep_synchronozation_type.USB_DC_EP_ADAPTIVE (*C enumerator*), 3021
usb_dc_ep_synchronozation_type.USB_DC_EP_ASYNCHRONOUS (*C enumerator*), 3021
usb_dc_ep_synchronozation_type.USB_DC_EP_NO_SYNCHRONIZATION (*C enumerator*), 3021
usb_dc_ep_synchronozation_type.USB_DC_EP_SYNCHRONOUS (*C enumerator*), 3021
usb_dc_ep_transfer_type (*C enum*), 3021
usb_dc_ep_transfer_type.USB_DC_EP_BULK (*C enumerator*), 3021
usb_dc_ep_transfer_type.USB_DC_EP_CONTROL (*C enumerator*), 3021
usb_dc_ep_transfer_type.USB_DC_EP_INTERRUPT (*C enumerator*), 3021
usb_dc_ep_transfer_type.USB_DC_EP_ISOCHRONOUS (*C enumerator*), 3021
usb_dc_ep_write (*C function*), 3024
usb_dc_reset (*C function*), 3022
usb_dc_set_address (*C function*), 3022
usb_dc_set_status_callback (*C function*), 3022
usb_dc_status_callback (*C type*), 3019
usb_dc_status_code (*C enum*), 3019
usb_dc_status_code.USB_DC_CLEAR_HALT (*C enumerator*), 3020
usb_dc_status_code.USB_DC_CONFIGURED (*C enumerator*), 3020
usb_dc_status_code.USB_DC_CONNECTED (*C enumerator*), 3020
usb_dc_status_code.USB_DC_DISCONNECTED (*C enumerator*), 3020
usb_dc_status_code.USB_DC_ERROR (*C enumerator*), 3020
usb_dc_status_code.USB_DC_INTERFACE (*C enumerator*), 3020
usb_dc_status_code.USB_DC_RESET (*C enumerator*), 3020
usb_dc_status_code.USB_DC_RESUME (*C enumerator*), 3020
usb_dc_status_code.USB_DC_SET_HALT (*C enumerator*), 3020
usb_dc_status_code.USB_DC_SOF (*C enumerator*), 3020
usb_dc_status_code.USB_DC_SUSPEND (*C enumerator*), 3020
usb_dc_status_code.USB_DC_UNKNOWN (*C enumerator*), 3020
usb_dc_wakeup_request (*C function*), 3025
usb_deconfig (*C function*), 3028

USB_DESC_HID (*C macro*), 3097
USB_DESC_HID_PHYSICAL (*C macro*), 3097
USB_DESC_HID_REPORT (*C macro*), 3097
USB_DEVICE_BOS_DESC_DEFINE_CAP (*C macro*), 3027
usb_disable (*C function*), 3028
usb_enable (*C function*), 3028
usb_ep_callback (*C type*), 3027
usb_ep_cfg_data (*C struct*), 3032
usb_ep_cfg_data.ep_addr (*C var*), 3032
usb_ep_cfg_data.ep_cb (*C var*), 3032
usb_ep_clear_stall (*C function*), 3029
usb_ep_read_continue (*C function*), 3030
usb_ep_read_wait (*C function*), 3030
usb_ep_set_stall (*C function*), 3029
usb_get_remote_wakeup_status (*C function*), 3031
USB_HID_GET_IDLE (*C macro*), 3097
USB_HID_GET_PROTOCOL (*C macro*), 3097
USB_HID_GET_REPORT (*C macro*), 3097
usb_hid_init (*C function*), 3035
usb_hid_register_device (*C function*), 3034
USB_HID_SET_IDLE (*C macro*), 3097
usb_hid_set_proto_code (*C function*), 3034
USB_HID_SET_PROTOCOL (*C macro*), 3097
USB_HID_SET_REPORT (*C macro*), 3097
USB_HID_VERSION (*C macro*), 3097
usb_interface_cfg_data (*C struct*), 3032
usb_interface_cfg_data.class_handler (*C var*), 3032
usb_interface_cfg_data.custom_handler (*C var*), 3032
usb_interface_cfg_data.vendor_handler (*C var*), 3032
usb_interface_config (*C type*), 3028
usb_read (*C function*), 3029
usb_request_handler (*C type*), 3027
usb_set_config (*C function*), 3028
USB_STRING_DESCRIPTOR_LENGTH (*C macro*), 3046
USB_TRANS_NO_ZLP (*C macro*), 3027
USB_TRANS_READ (*C macro*), 3027
USB_TRANS_WRITE (*C macro*), 3027
usb_transfer (*C function*), 3030
usb_transfer_callback (*C type*), 3028
usb_transfer_ep_callback (*C function*), 3030
usb_transfer_is_busy (*C function*), 3031
usb_transfer_sync (*C function*), 3031
usb_wakeup_request (*C function*), 3031
usb_write (*C function*), 3028
usbc_bypass_next_sleep (*C function*), 3090
usbc_get_dpm_data (*C function*), 3090
usbc_policy_check_t (*C enum*), 3089
usbc_policy_check_t.CHECK_DATA_ROLE_SWAP_TO_DFP (*C enumerator*), 3089
usbc_policy_check_t.CHECK_DATA_ROLE_SWAP_TO_UFP (*C enumerator*), 3089
usbc_policy_check_t.CHECK_POWER_ROLE_SWAP (*C enumerator*), 3089
usbc_policy_check_t.CHECK_SNK_AT_DEFAULT_LEVEL (*C enumerator*), 3089
usbc_policy_check_t.CHECK_SRC_PS_AT_DEFAULT_LEVEL (*C enumerator*), 3089
usbc_policy_check_t.CHECK_VCONN_CONTROL (*C enumerator*), 3089
usbc_policy_notify_t (*C enum*), 3087
usbc_policy_notify_t.DATA_ROLE_IS_DFP (*C enumerator*), 3088
usbc_policy_notify_t.DATA_ROLE_IS_UFP (*C enumerator*), 3088
usbc_policy_notify_t.HARD_RESET_RECEIVED (*C enumerator*), 3088
usbc_policy_notify_t.MSG_ACCEPT_RECEIVED (*C enumerator*), 3088

usbc_policy_notify_t.MSG_DISCARDED (*C enumerator*), 3088
usbc_policy_notify_t.MSG_NOT_SUPPORTED_RECEIVED (*C enumerator*), 3088
usbc_policy_notify_t.MSG_REJECTED_RECEIVED (*C enumerator*), 3088
usbc_policy_notify_t.NOT_PD_CONNECTED (*C enumerator*), 3088
usbc_policy_notify_t.PD_CONNECTED (*C enumerator*), 3088
usbc_policy_notify_t.PORT_PARTNER_NOT_RESPONSIVE (*C enumerator*), 3088
usbc_policy_notify_t.POWER_CHANGE_0A0 (*C enumerator*), 3088
usbc_policy_notify_t.POWER_CHANGE_1A5 (*C enumerator*), 3088
usbc_policy_notify_t.POWER_CHANGE_3A0 (*C enumerator*), 3089
usbc_policy_notify_t.POWER_CHANGE_DEF (*C enumerator*), 3088
usbc_policy_notify_t.PROTOCOL_ERROR (*C enumerator*), 3088
usbc_policy_notify_t.SENDER_RESPONSE_TIMEOUT (*C enumerator*), 3089
usbc_policy_notify_t.SNK_TRANSITION_TO_DEFAULT (*C enumerator*), 3088
usbc_policy_notify_t.SOURCE_CAPABILITIES_RECEIVED (*C enumerator*), 3089
usbc_policy_notify_t.TRANSITION_PS (*C enumerator*), 3088
usbc_policy_request_t (*C enum*), 3087
usbc_policy_request_t.REQUEST_GET_SNK_CAPS (*C enumerator*), 3087
usbc_policy_request_t.REQUEST_NOP (*C enumerator*), 3087
usbc_policy_request_t.REQUEST_PE_DR_SWAP (*C enumerator*), 3087
usbc_policy_request_t.REQUEST_PE_GET_SRC_CAPS (*C enumerator*), 3087
usbc_policy_request_t.REQUEST_PE_GOTO_MIN (*C enumerator*), 3087
usbc_policy_request_t.REQUEST_PE_HARD_RESET_SEND (*C enumerator*), 3087
usbc_policy_request_t.REQUEST_PE_SOFT_RESET_SEND (*C enumerator*), 3087
usbc_policy_request_t.REQUEST_TC_DISABLED (*C enumerator*), 3087
usbc_policy_request_t.REQUEST_TC_END (*C enumerator*), 3087
usbc_policy_request_t.REQUEST_TC_ERROR_RECOVERY (*C enumerator*), 3087
usbc_policy_wait_t (*C enum*), 3089
usbc_policy_wait_t.WAIT_DATA_ROLE_SWAP (*C enumerator*), 3089
usbc_policy_wait_t.WAIT_POWER_ROLE_SWAP (*C enumerator*), 3089
usbc_policy_wait_t.WAIT_SINK_REQUEST (*C enumerator*), 3089
usbc_policy_wait_t.WAIT_VCONN_SWAP (*C enumerator*), 3089
usbc_request (*C function*), 3090
usbc_set_dpm_data (*C function*), 3090
usbc_set_policy_cb_change_src_caps (*C function*), 3093
usbc_set_policy_cb_check (*C function*), 3091
usbc_set_policy_cb_check_sink_request (*C function*), 3092
usbc_set_policy_cb_get_rdo (*C function*), 3092
usbc_set_policy_cb_get_snk_cap (*C function*), 3091
usbc_set_policy_cb_get_src_caps (*C function*), 3092
usbc_set_policy_cb_get_src_rp (*C function*), 3092
usbc_set_policy_cb_is_ps_ready (*C function*), 3092
usbc_set_policy_cb_is_snk_at_default (*C function*), 3092
usbc_set_policy_cb_notify (*C function*), 3091
usbc_set_policy_cb_present_contract_is_valid (*C function*), 3093
usbc_set_policy_cb_set_port_partner_snk_cap (*C function*), 3093
usbc_set_policy_cb_set_src_cap (*C function*), 3091
usbc_set_policy_cb_src_en (*C function*), 3092
usbc_set_policy_cb_wait_notify (*C function*), 3091
usbc_set_vconn_control_cb (*C function*), 3091
usbc_set_vconn_discharge_cb (*C function*), 3091
usbc_snk_req_reply_t (*C enum*), 3090
usbc_snk_req_reply_t.SNK_REQUEST_REJECT (*C enumerator*), 3090
usbc_snk_req_reply_t.SNK_REQUEST_VALID (*C enumerator*), 3090
usbc_snk_req_reply_t.SNK_REQUEST_WAIT (*C enumerator*), 3090
usbc_start (*C function*), 3090
usbc_suspend (*C function*), 3090
usbc_vbus_check_level (*C function*), 3653
usbc_vbus_discharge (*C function*), 3654

usbc_vbus_driver_api (*C struct*), 3654
usbc_vbus_enable (*C function*), 3654
usbc_vbus_measure (*C function*), 3653
usbd_add_configuration (*C function*), 3050
usbd_add_descriptor (*C function*), 3050
usbd_bos_desc_data (*C struct*), 3057
usbd_bos_desc_data.utype (*C var*), 3057
usbd_bus_speed (*C function*), 3054
usbd_can_detect_vbus (*C function*), 3056
usbd_caps_speed (*C function*), 3054
USBD_CCTX_REGISTERED (*C macro*), 3046
usbd_cctx_vendor_req (*C struct*), 3060
usbd_cctx_vendor_req.len (*C var*), 3060
usbd_cctx_vendor_req.reqs (*C var*), 3060
usbd_ch9_data (*C struct*), 3058
usbd_ch9_data.alternate (*C var*), 3058
usbd_ch9_data.configuration (*C var*), 3058
usbd_ch9_data.ctrl_type (*C var*), 3058
usbd_ch9_data.ep_halt (*C var*), 3058
usbd_ch9_data.post_status (*C var*), 3058
usbd_ch9_data.setup (*C var*), 3058
usbd_ch9_data.state (*C var*), 3058
usbd_ch9_state (*C enum*), 3049
usbd_ch9_state.USBD_STATE_ADDRESS (*C enumerator*), 3049
usbd_ch9_state.USBD_STATE_CONFIGURED (*C enumerator*), 3049
usbd_ch9_state.USBD_STATE_DEFAULT (*C enumerator*), 3049
usbd_class_api (*C struct*), 3060
usbd_class_api.control_to_dev (*C var*), 3060
usbd_class_api.control_to_host (*C var*), 3060
usbd_class_api.disable (*C var*), 3060
usbd_class_api.enable (*C var*), 3060
usbd_class_api.feature_halt (*C var*), 3060
usbd_class_api.get_desc (*C var*), 3061
usbd_class_api.init (*C var*), 3061
usbd_class_api.request (*C var*), 3060
usbd_class_api.resumed (*C var*), 3060
usbd_class_api.shutdown (*C var*), 3061
usbd_class_api.sof (*C var*), 3060
usbd_class_api.suspended (*C var*), 3060
usbd_class_api.update (*C var*), 3060
usbd_class_data (*C struct*), 3061
usbd_class_data.api (*C var*), 3061
usbd_class_data.name (*C var*), 3061
usbd_class_data.priv (*C var*), 3061
usbd_class_data.uds_ctx (*C var*), 3061
usbd_class_data.v_reqs (*C var*), 3061
usbd_class_get_ctx (*C function*), 3049
usbd_class_get_private (*C function*), 3049
usbd_config_attrib_rwup (*C function*), 3055
usbd_config_attrib_self (*C function*), 3056
usbd_config_maxpower (*C function*), 3056
usbd_config_node (*C struct*), 3057
usbd_config_node.class_list (*C var*), 3058
usbd_config_node.desc (*C var*), 3058
usbd_config_node.node (*C var*), 3058
USBD_CONFIGURATION_DEFINE (*C macro*), 3046
usbd_context (*C struct*), 3059
usbd_context.ch9_data (*C var*), 3059

usbd_context.descriptors (*C var*), 3059
usbd_context.dev (*C var*), 3059
usbd_context.fs_configs (*C var*), 3059
usbd_context.fs_desc (*C var*), 3059
usbd_context.hs_configs (*C var*), 3059
usbd_context.hs_desc (*C var*), 3059
usbd_context.msg_cb (*C var*), 3059
usbd_context.mutex (*C var*), 3059
usbd_context.name (*C var*), 3059
usbd_context.status (*C var*), 3059
USBD_DEFINE_CLASS (*C macro*), 3048
USBD_DEFINE_MSC_LUN (*C macro*), 3068
USBD_DESC_BOS_DEFINE (*C macro*), 3048
USBD_DESC_LANG_DEFINE (*C macro*), 3047
USBD_DESC_MANUFACTURER_DEFINE (*C macro*), 3047
usbd_desc_node (*C struct*), 3057
usbd_desc_node.bDescriptorType (*C var*), 3057
usbd_desc_node.bLength (*C var*), 3057
usbd_desc_node.node (*C var*), 3057
usbd_desc_node.ptr (*C var*), 3057
USBD_DESC_PRODUCT_DEFINE (*C macro*), 3047
USBD_DESC_SERIAL_NUMBER_DEFINE (*C macro*), 3048
USBD_DESC_STRING_DEFINE (*C macro*), 3047
USBD_DEVICE_DEFINE (*C macro*), 3046
usbd_device_set_bcd (*C function*), 3055
usbd_device_set_code_triple (*C function*), 3055
usbd_device_set_pid (*C function*), 3055
usbd_device_set_vid (*C function*), 3055
usbd_disable (*C function*), 3052
usbd_enable (*C function*), 3052
usbd_ep_buf_alloc (*C function*), 3053
usbd_ep_buf_free (*C function*), 3054
usbd_ep_clear_halt (*C function*), 3053
usbd_ep_ctrl_enqueue (*C function*), 3053
usbd_ep_dequeue (*C function*), 3054
usbd_ep_enqueue (*C function*), 3053
usbd_ep_is_halted (*C function*), 3053
usbd_ep_set_halt (*C function*), 3053
usbd_init (*C function*), 3052
usbd_is_suspended (*C function*), 3054
usbd_msg (*C struct*), 3062
usbd_msg_cb_t (*C type*), 3048
usbd_msg_register_cb (*C function*), 3052
usbd_msg_type (*C enum*), 3061
usbd_msg_type_string (*C function*), 3062
usbd_msg_type.USBD_MSG_CDC_ACM_CONTROL_LINE_STATE (*C enumerator*), 3062
usbd_msg_type.USBD_MSG_CDC_ACM_LINE_CODING (*C enumerator*), 3062
usbd_msg_type.USBD_MSG_MAX_NUMBER (*C enumerator*), 3062
usbd_msg_type.USBD_MSG_RESET (*C enumerator*), 3062
usbd_msg_type.USBD_MSG_RESUME (*C enumerator*), 3061
usbd_msg_type.USBD_MSG_STACK_ERROR (*C enumerator*), 3062
usbd_msg_type.USBD_MSG_SUSPEND (*C enumerator*), 3062
usbd_msg_type.USBD_MSG_UDC_ERROR (*C enumerator*), 3062
usbd_msg_type.USBD_MSG_VBUS_READY (*C enumerator*), 3061
usbd_msg_type.USBD_MSG_VBUS_REMOVED (*C enumerator*), 3061
usbd_msg.type (*C var*), 3062
USBD_NUMOF_INTERFACES_MAX (*C macro*), 3046
usbd_register_all_classes (*C function*), 3051

usbd_register_class (*C function*), 3050
usbd_remove_descriptor (*C function*), 3050
usbd_shutdown (*C function*), 3052
usbd_speed (*C enum*), 3049
usbd_speed.USBD_SPEED_FS (*C enumerator*), 3049
usbd_speed.USBD_SPEED_HS (*C enumerator*), 3049
usbd_speed.USBD_SPEED_SS (*C enumerator*), 3049
usbd_status (*C struct*), 3058
usbd_status.enabled (*C var*), 3058
usbd_status.initialized (*C var*), 3058
usbd_status.rwup (*C var*), 3059
usbd_status.speed (*C var*), 3059
usbd_status.suspended (*C var*), 3059
usbd_str_desc_data (*C struct*), 3056
usbd_str_desc_data.ascii7 (*C var*), 3057
usbd_str_desc_data.idx (*C var*), 3057
usbd_str_desc_data.use_hwinfo (*C var*), 3057
usbd_str_desc_data.utype (*C var*), 3057
usbd_str_desc_get_idx (*C function*), 3050
usbd_uac2_send (*C function*), 3066
usbd_uac2_set_ops (*C function*), 3066
usbd_unregister_all_classes (*C function*), 3051
usbd_unregister_class (*C function*), 3051
USBD_VENDOR_REQ (*C macro*), 3048
usbd_wakeup_request (*C function*), 3054
usbpd_cc_pin (*C enum*), 3683
usbpd_cc_pin.USBPD_CC_PIN_1 (*C enumerator*), 3683
usbpd_cc_pin.USBPD_CC_PIN_2 (*C enumerator*), 3683
USEC_PER_MSEC (*C macro*), 476
USEC_PER_SEC (*C macro*), 476
utf8_lcpy (*C function*), 698
utf8_trunc (*C function*), 698
UTIL_AND (*C macro*), 691
UTIL_DEC (*C macro*), 691
UTIL_INC (*C macro*), 691
UTIL_OR (*C macro*), 691
UTIL_X2 (*C macro*), 691

V

variant, **3937**
VENDOR_REQ_DEFINE (*C macro*), 3048
vfprintfcb (*C function*), 870
video_api_dequeue_t (*C type*), 3695
video_api_enqueue_t (*C type*), 3695
video_api_flush_t (*C type*), 3695
video_api_get_caps_t (*C type*), 3696
video_api_get_ctrl_t (*C type*), 3696
video_api_get_format_t (*C type*), 3695
video_api_set_ctrl_t (*C type*), 3695
video_api_set_format_t (*C type*), 3695
video_api_set_signal_t (*C type*), 3696
video_api_stream_start_t (*C type*), 3695
video_api_stream_stop_t (*C type*), 3695
video_buffer (*C struct*), 3701
video_buffer_aligned_alloc (*C function*), 3700
video_buffer_alloc (*C function*), 3700
video_buffer_release (*C function*), 3700
video_buffer.buffer (*C var*), 3702

video_buffer.bytesused (*C var*), 3702
video_buffer.driver_data (*C var*), 3702
video_buffer.size (*C var*), 3702
video_buffer.timestamp (*C var*), 3702
video_caps (*C struct*), 3701
video_caps.format_caps (*C var*), 3701
video_caps.min_vbuf_count (*C var*), 3701
VIDEO_CID_CAMERA_BRIGHTNESS (*C macro*), 3703
VIDEO_CID_CAMERA_COLORBAR (*C macro*), 3703
VIDEO_CID_CAMERA_CONTRAST (*C macro*), 3703
VIDEO_CID_CAMERA_EXPOSURE (*C macro*), 3703
VIDEO_CID_CAMERA_GAIN (*C macro*), 3703
VIDEO_CID_CAMERA_QUALITY (*C macro*), 3703
VIDEO_CID_CAMERA_SATURATION (*C macro*), 3703
VIDEO_CID_CAMERA_WHITE_BAL (*C macro*), 3703
VIDEO_CID_CAMERA_ZOOM (*C macro*), 3703
VIDEO_CID_HFLIP (*C macro*), 3702
VIDEO_CID_VFLIP (*C macro*), 3702
VIDEO_CTRL_CLASS_CAMERA (*C macro*), 3702
VIDEO_CTRL_CLASS_GENERIC (*C macro*), 3702
VIDEO_CTRL_CLASS_JPEG (*C macro*), 3702
VIDEO_CTRL_CLASS_MPEG (*C macro*), 3702
VIDEO_CTRL_CLASS_VENDOR (*C macro*), 3702
video_dequeue (*C function*), 3697
video_driver_api (*C struct*), 3702
video_endpoint_id (*C enum*), 3696
video_endpoint_id.VIDEO_EP_ANY (*C enumerator*), 3696
video_endpoint_id.VIDEO_EP_IN (*C enumerator*), 3696
video_endpoint_id.VIDEO_EP_NONE (*C enumerator*), 3696
video_endpoint_id.VIDEO_EP_OUT (*C enumerator*), 3696
video_enqueue (*C function*), 3697
video_flush (*C function*), 3698
video_format (*C struct*), 3700
video_format_cap (*C struct*), 3700
video_format_cap.height_max (*C var*), 3701
video_format_cap.height_min (*C var*), 3701
video_format_cap.height_step (*C var*), 3701
video_format_cap.pixelformat (*C var*), 3701
video_format_cap.width_max (*C var*), 3701
video_format_cap.width_min (*C var*), 3701
video_format_cap.width_step (*C var*), 3701
video_format.height (*C var*), 3700
video_format.pitch (*C var*), 3700
video_format.pixelformat (*C var*), 3700
video_format.width (*C var*), 3700
video_fourcc (*C macro*), 3695
video_get_caps (*C function*), 3698
video_get_ctrl (*C function*), 3699
video_get_format (*C function*), 3697
video_set_ctrl (*C function*), 3699
video_set_format (*C function*), 3697
video_set_signal (*C function*), 3699
video_signal_result (*C enum*), 3696
video_signal_result.VIDEO_BUF_ABORTED (*C enumerator*), 3696
video_signal_result.VIDEO_BUF_DONE (*C enumerator*), 3696
video_signal_result.VIDEO_BUF_ERROR (*C enumerator*), 3696
video_stream_start (*C function*), 3698
video_stream_stop (*C function*), 3698

vprintfcb (*C function*), 871
vsnprintfcb (*C function*), 872

W

W1_CMD_MATCH_ROM (*C macro*), 3150
W1_CMD_OVERDRIVE_MATCH_ROM (*C macro*), 3151
W1_CMD_OVERDRIVE_SKIP_ROM (*C macro*), 3151
W1_CMD_READ_ROM (*C macro*), 3150
W1_CMD_RESUME (*C macro*), 3150
W1_CMD_SEARCH_ALARM (*C macro*), 3151
W1_CMD_SEARCH_ROM (*C macro*), 3150
W1_CMD_SKIP_ROM (*C macro*), 3150
w1_configure (*C function*), 3150
w1_crc8 (*C function*), 3155
W1_CRC8_POLYNOMIAL (*C macro*), 3151
W1_CRC8_SEED (*C macro*), 3151
w1_crc16 (*C function*), 3155
W1_CRC16_POLYNOMIAL (*C macro*), 3151
W1_CRC16_SEED (*C macro*), 3151
w1_get_slave_count (*C function*), 3150
w1_lock_bus (*C function*), 3157
w1_match_rom (*C function*), 3152
w1_read_bit (*C function*), 3148
w1_read_block (*C function*), 3149
w1_read_byte (*C function*), 3149
w1_read_rom (*C function*), 3152
w1_reset_bus (*C function*), 3148
w1_reset_select (*C function*), 3153
w1_resume_command (*C function*), 3152
w1_rom (*C struct*), 3155
W1_ROM_INIT_ZERO (*C macro*), 3151
w1_rom_to_uint64 (*C function*), 3155
w1_rom.crc (*C var*), 3156
w1_rom.family (*C var*), 3156
w1_rom.serial (*C var*), 3156
w1_search_alarm (*C function*), 3154
W1_SEARCH_ALL_FAMILIES (*C macro*), 3151
w1_search_bus (*C function*), 3153
w1_search_callback_t (*C type*), 3151
w1_search_rom (*C function*), 3154
w1_settings_type (*C enum*), 3156
w1_settings_type.W1_SETINGS_TYPE_COUNT (*C enumerator*), 3157
w1_settings_type.W1_SETTING_SPEED (*C enumerator*), 3157
w1_settings_type.W1_SETTING_STRONG_PULLUP (*C enumerator*), 3157
w1_skip_rom (*C function*), 3152
w1_slave_config (*C struct*), 3156
w1_slave_config.overdrive (*C var*), 3156
w1_slave_config.rom (*C var*), 3156
w1_uint64_to_rom (*C function*), 3155
w1_unlock_bus (*C function*), 3157
w1_write_bit (*C function*), 3148
w1_write_block (*C function*), 3149
w1_write_byte (*C function*), 3149
w1_write_read (*C function*), 3153
WAIT_FOR (*C macro*), 685
wait_for_prompt() (*twister_harness.Shell method*), 269
WB_DN (*C macro*), 682
WB_UP (*C macro*), 682

wdt_callback_t (*C type*), 3704
wdt_disable (*C function*), 3704
wdt_feed (*C function*), 3705
WDT_FLAG_RESET_CPU_CORE (*C macro*), 3704
WDT_FLAG_RESET_NONE (*C macro*), 3704
WDT_FLAG_RESET_SOC (*C macro*), 3704
wdt_install_timeout (*C function*), 3705
WDT_OPT_PAUSE_HALTED_BY_DBG (*C macro*), 3703
WDT_OPT_PAUSE_IN_SLEEP (*C macro*), 3703
wdt_setup (*C function*), 3704
wdt_timeout_cfg (*C struct*), 3706
wdt_timeout_cfg.callback (*C var*), 3706
wdt_timeout_cfg.flags (*C var*), 3706
wdt_timeout_cfg.next (*C var*), 3706
wdt_timeout_cfg.window (*C var*), 3706
wdt_window (*C struct*), 3706
wdt_window.max (*C var*), 3706
wdt_window.min (*C var*), 3706
websocket_connect (*C function*), 2573
websocket_connect_cb_t (*C type*), 2572
websocket_disconnect (*C function*), 2574
WEBSOCKET_FLAG_BINARY (*C macro*), 2572
WEBSOCKET_FLAG_CLOSE (*C macro*), 2572
WEBSOCKET_FLAG_FINAL (*C macro*), 2572
WEBSOCKET_FLAG_PING (*C macro*), 2572
WEBSOCKET_FLAG_PONG (*C macro*), 2572
WEBSOCKET_FLAG_TEXT (*C macro*), 2572
websocket_opcode (*C enum*), 2573
websocket_opcode.WEBSOCKET_OPCODE_CLOSE (*C enumerator*), 2573
websocket_opcode.WEBSOCKET_OPCODE_CONTINUE (*C enumerator*), 2573
websocket_opcode.WEBSOCKET_OPCODE_DATA_BINARY (*C enumerator*), 2573
websocket_opcode.WEBSOCKET_OPCODE_DATA_TEXT (*C enumerator*), 2573
websocket_opcode.WEBSOCKET_OPCODE_PING (*C enumerator*), 2573
websocket_opcode.WEBSOCKET_OPCODE_PONG (*C enumerator*), 2573
websocket_recv_msg (*C function*), 2574
websocket_register (*C function*), 2575
websocket_request (*C struct*), 2575
websocket_request.cb (*C var*), 2575
websocket_request.host (*C var*), 2575
websocket_request.http_cb (*C var*), 2576
websocket_request.optional_headers (*C var*), 2575
websocket_request.optional_headers_cb (*C var*), 2575
websocket_request.tmp_buf (*C var*), 2576
websocket_request.tmp_buf_len (*C var*), 2576
websocket_request.url (*C var*), 2575
websocket_send_msg (*C function*), 2574
websocket_unregister (*C function*), 2575
west, **3937**
west installation, **3937**
west manifest, **3937**
west manifest repository, **3937**
west project, **3937**
west workspace, **3937**
wifi_ap_config_param (*C enum*), 2725
wifi_ap_config_params (*C struct*), 2742
wifi_ap_config_params.max_inactivity (*C var*), 2742
wifi_ap_config_params.max_num_sta (*C var*), 2742
wifi_ap_config_params.type (*C var*), 2742

wifi_ap_config_param.WIFI_AP_CONFIG_PARAM_MAX_INACTIVITY (*C enumerator*), 2725
wifi_ap_config_param.WIFI_AP_CONFIG_PARAM_MAX_NUM_STA (*C enumerator*), 2725
wifi_ap_sta_info (*C struct*), 2740
wifi_ap_sta_info.link_mode (*C var*), 2741
wifi_ap_sta_info.mac (*C var*), 2741
wifi_ap_sta_info.mac_length (*C var*), 2741
wifi_ap_sta_info.twt_capable (*C var*), 2741
wifi_ap_status (*C enum*), 2729
wifi_ap_status.WIFI_STATUS_AP_AUTH_TYPE_NOT_SUPPORTED (*C enumerator*), 2729
wifi_ap_status.WIFI_STATUS_AP_CHANNEL_NOT_ALLOWED (*C enumerator*), 2729
wifi_ap_status.WIFI_STATUS_AP_CHANNEL_NOT_SUPPORTED (*C enumerator*), 2729
wifi_ap_status.WIFI_STATUS_AP_FAIL (*C enumerator*), 2729
wifi_ap_status.WIFI_STATUS_AP_OP_NOT_PERMITTED (*C enumerator*), 2729
wifi_ap_status.WIFI_STATUS_AP_OP_NOT_SUPPORTED (*C enumerator*), 2729
wifi_ap_status.WIFI_STATUS_AP_SSID_NOT_ALLOWED (*C enumerator*), 2729
wifi_ap_status.WIFI_STATUS_AP_SUCCESS (*C enumerator*), 2729
wifi_band_channel (*C struct*), 2732
wifi_band_channel.band (*C var*), 2732
wifi_band_channel.channel (*C var*), 2732
wifi_band_txt (*C function*), 2730
WIFI_CHANNEL_ANY (*C macro*), 2715
wifi_channel_info (*C struct*), 2741
wifi_channel_info.channel (*C var*), 2742
wifi_channel_info.if_index (*C var*), 2742
wifi_channel_info.oper (*C var*), 2742
WIFI_CHANNEL_MAX (*C macro*), 2715
WIFI_CHANNEL_MIN (*C macro*), 2714
wifi_config_ps_param_fail_reason (*C enum*), 2725
wifi_config_ps_param_fail_reason.WIFI_PS_PARAM_FAIL_CMD_EXEC_FAIL (*C enumerator*), 2725
wifi_config_ps_param_fail_reason.WIFI_PS_PARAM_FAIL_DEVICE_CONNECTED (*C enumerator*), 2725
wifi_config_ps_param_fail_reason.WIFI_PS_PARAM_FAIL_DEVICE_NOT_CONNECTED (*C enumerator*), 2725
wifi_config_ps_param_fail_reason.WIFI_PS_PARAM_FAIL_OPERATION_NOT_SUPPORTED (*C enumerator*), 2725
wifi_config_ps_param_fail_reason.WIFI_PS_PARAM_FAIL_UNABLE_TO_GET_IFACE_STATUS (*C enumerator*), 2725
wifi_config_ps_param_fail_reason.WIFI_PS_PARAM_FAIL_UNSPECIFIED (*C enumerator*), 2725
wifi_config_ps_param_fail_reason.WIFI_PS_PARAM_LISTEN_INTERVAL_RANGE_INVALID (*C enumerator*), 2725
wifi_conn_status (*C enum*), 2727
wifi_conn_status.WIFI_STATUS_CONN_AP_NOT_FOUND (*C enumerator*), 2728
wifi_conn_status.WIFI_STATUS_CONN_FAIL (*C enumerator*), 2728
wifi_conn_status.WIFI_STATUS_CONN_LAST_STATUS (*C enumerator*), 2728
wifi_conn_status.WIFI_STATUS_CONN_SUCCESS (*C enumerator*), 2728
wifi_conn_status.WIFI_STATUS_CONN_TIMEOUT (*C enumerator*), 2728
wifi_conn_status.WIFI_STATUS_CONN_WRONG_PASSWORD (*C enumerator*), 2728
wifi_conn_status.WIFI_STATUS_DISCONN_FIRST_STATUS (*C enumerator*), 2728
wifi_connect_req_params (*C struct*), 2734
wifi_connect_req_params.band (*C var*), 2735
wifi_connect_req_params.bssid (*C var*), 2735
wifi_connect_req_params.channel (*C var*), 2735
wifi_connect_req_params.mfp (*C var*), 2735
wifi_connect_req_params.psk (*C var*), 2734
wifi_connect_req_params.psk_length (*C var*), 2734
wifi_connect_req_params.sae_password (*C var*), 2734
wifi_connect_req_params.sae_password_length (*C var*), 2734
wifi_connect_req_params.security (*C var*), 2735

wifi_connect_req_params.ssid (*C var*), 2734
wifi_connect_req_params.ssid_length (*C var*), 2734
wifi_connect_req_params.timeout (*C var*), 2735
WIFI_COUNTRY_CODE_LEN (*C macro*), 2714
wifi_disconn_reason (*C enum*), 2728
wifi_disconn_reason.WIFI_REASON_DISCONN_AP_LEAVING (*C enumerator*), 2728
wifi_disconn_reason.WIFI_REASON_DISCONN_INACTIVITY (*C enumerator*), 2728
wifi_disconn_reason.WIFI_REASON_DISCONN_SUCCESS (*C enumerator*), 2728
wifi_disconn_reason.WIFI_REASON_DISCONN_UNSPECIFIED (*C enumerator*), 2728
wifi_disconn_reason.WIFI_REASON_DISCONN_USER_REQUEST (*C enumerator*), 2728
wifi_filter (*C enum*), 2721
wifi_filter_info (*C struct*), 2741
wifi_filter_info.buffer_size (*C var*), 2741
wifi_filter_info.filter (*C var*), 2741
wifi_filter_info.if_index (*C var*), 2741
wifi_filter_info.oper (*C var*), 2741
wifi_filter.WIFI_PACKET_FILTER_ALL (*C enumerator*), 2722
wifi_filter.WIFI_PACKET_FILTER_CTRL (*C enumerator*), 2722
wifi_filter.WIFI_PACKET_FILTER_DATA (*C enumerator*), 2722
wifi_filter.WIFI_PACKET_FILTER_MGMT (*C enumerator*), 2722
wifi_frequency_bands (*C enum*), 2718
wifi_frequency_bands.__WIFI_FREQ_BAND_AFTER_LAST (*C enumerator*), 2718
wifi_frequency_bands.WIFI_FREQ_BAND_2_4_GHZ (*C enumerator*), 2718
wifi_frequency_bands.WIFI_FREQ_BAND_5_GHZ (*C enumerator*), 2718
wifi_frequency_bands.WIFI_FREQ_BAND_6_GHZ (*C enumerator*), 2718
wifi_frequency_bands.WIFI_FREQ_BAND_MAX (*C enumerator*), 2718
wifi_frequency_bands.WIFI_FREQ_BAND_UNKNOWN (*C enumerator*), 2718
wifi_iface_mode (*C enum*), 2719
wifi_iface_mode.WIFI_MODE_AP (*C enumerator*), 2719
wifi_iface_mode.WIFI_MODE_IBSS (*C enumerator*), 2719
wifi_iface_mode.WIFI_MODE_INFRA (*C enumerator*), 2719
wifi_iface_mode.WIFI_MODE_MESH (*C enumerator*), 2720
wifi_iface_mode.WIFI_MODE_P2P_GO (*C enumerator*), 2720
wifi_iface_mode.WIFI_MODE_P2P_GROUP_FORMATION (*C enumerator*), 2720
wifi_iface_state (*C enum*), 2718
wifi_iface_state.WIFI_STATE_4WAY_HANDSHAKE (*C enumerator*), 2719
wifi_iface_state.WIFI_STATE_ASSOCIATED (*C enumerator*), 2719
wifi_iface_state.WIFI_STATE_ASSOCIATING (*C enumerator*), 2719
wifi_iface_state.WIFI_STATE_AUTHENTICATING (*C enumerator*), 2719
wifi_iface_state.WIFI_STATE_COMPLETED (*C enumerator*), 2719
wifi_iface_state.WIFI_STATE_DISCONNECTED (*C enumerator*), 2719
wifi_iface_state.WIFI_STATE_GROUP_HANDSHAKE (*C enumerator*), 2719
wifi_iface_state.WIFI_STATE_INACTIVE (*C enumerator*), 2719
wifi_iface_state.WIFI_STATE_INTERFACE_DISABLED (*C enumerator*), 2719
wifi_iface_state.WIFI_STATE_SCANNING (*C enumerator*), 2719
wifi_iface_status (*C struct*), 2735
wifi_iface_status.band (*C var*), 2736
wifi_iface_status.beacon_interval (*C var*), 2736
wifi_iface_status.bssid (*C var*), 2736
wifi_iface_status.channel (*C var*), 2736
wifi_iface_status.dtim_period (*C var*), 2736
wifi_iface_status.iface_mode (*C var*), 2736
wifi_iface_status.link_mode (*C var*), 2736
wifi_iface_status.mfp (*C var*), 2736
wifi_iface_status.rssi (*C var*), 2736
wifi_iface_status.security (*C var*), 2736
wifi_iface_status.ssid (*C var*), 2736
wifi_iface_status.ssid_len (*C var*), 2735

wifi_iface_status.state (*C var*), 2735
wifi_iface_status.twt_capable (*C var*), 2736
WIFI_INTERFACE_INDEX_MAX (*C macro*), 2715
WIFI_INTERFACE_INDEX_MIN (*C macro*), 2715
wifi_link_mode (*C enum*), 2720
wifi_link_mode_txt (*C function*), 2730
wifi_link_mode.WIFI_0 (*C enumerator*), 2720
wifi_link_mode.WIFI_1 (*C enumerator*), 2720
wifi_link_mode.WIFI_2 (*C enumerator*), 2720
wifi_link_mode.WIFI_3 (*C enumerator*), 2720
wifi_link_mode.WIFI_4 (*C enumerator*), 2720
wifi_link_mode.WIFI_5 (*C enumerator*), 2720
wifi_link_mode.WIFI_6 (*C enumerator*), 2720
wifi_link_mode.WIFI_6E (*C enumerator*), 2720
wifi_link_mode.WIFI_7 (*C enumerator*), 2720
WIFI_MAC_ADDR_LEN (*C macro*), 2714
wifi_mfp_options (*C enum*), 2718
wifi_mfp_options.WIFI_MFP_DISABLE (*C enumerator*), 2718
wifi_mfp_options.WIFI_MFP_OPTIONAL (*C enumerator*), 2718
wifi_mfp_options.WIFI_MFP_REQUIRED (*C enumerator*), 2718
wifi_mfp_txt (*C function*), 2730
wifi_mgmt_op (*C enum*), 2729
wifi_mgmt_ops (*C struct*), 2742
wifi_mgmt_ops.ap_config_params (*C var*), 2745
wifi_mgmt_ops.ap_disable (*C var*), 2743
wifi_mgmt_ops.ap_enable (*C var*), 2743
wifi_mgmt_ops.ap_sta_disconnect (*C var*), 2743
wifi_mgmt_ops.channel (*C var*), 2744
wifi_mgmt_ops.connect (*C var*), 2742
wifi_mgmt_ops.disconnect (*C var*), 2743
wifi_mgmt_ops.filter (*C var*), 2744
wifi_mgmt_ops.get_power_save_config (*C var*), 2744
wifi_mgmt_ops.get_stats (*C var*), 2743
wifi_mgmt_ops.get_version (*C var*), 2745
wifi_mgmt_ops.iface_status (*C var*), 2743
wifi_mgmt_ops.mode (*C var*), 2744
wifi_mgmt_ops.reg_domain (*C var*), 2744
wifi_mgmt_ops.scan (*C var*), 2742
wifi_mgmt_ops.set_power_save (*C var*), 2743
wifi_mgmt_ops.set_rts_threshold (*C var*), 2745
wifi_mgmt_ops.set_twt (*C var*), 2744
wifi_mgmt_op.WIFI_MGMT_GET (*C enumerator*), 2729
wifi_mgmt_op.WIFI_MGMT_SET (*C enumerator*), 2729
wifi_mgmt_raise_ap_disable_result_event (*C function*), 2731
wifi_mgmt_raise_ap_enable_result_event (*C function*), 2731
wifi_mgmt_raise_ap_sta_connected_event (*C function*), 2732
wifi_mgmt_raise_ap_sta_disconnected_event (*C function*), 2732
wifi_mgmt_raise_connect_result_event (*C function*), 2730
wifi_mgmt_raise_disconnect_complete_event (*C function*), 2731
wifi_mgmt_raise_disconnect_result_event (*C function*), 2730
wifi_mgmt_raise_iface_status_event (*C function*), 2731
wifi_mgmt_raise_raw_scan_result_event (*C function*), 2731
wifi_mgmt_raise_twt_event (*C function*), 2731
wifi_mgmt_raise_twt_sleep_state (*C function*), 2731
wifi_mode_info (*C struct*), 2741
wifi_mode_info.if_index (*C var*), 2741
wifi_mode_info.mode (*C var*), 2741
wifi_mode_info.oper (*C var*), 2741

wifi_mode_txt (*C function*), 2730
wifi_operational_modes (*C enum*), 2721
wifi_operational_modes.WIFI_AP_MODE (*C enumerator*), 2721
wifi_operational_modes.WIFI_MONITOR_MODE (*C enumerator*), 2721
wifi_operational_modes.WIFI_PROMISCUOUS_MODE (*C enumerator*), 2721
wifi_operational_modes.WIFI_SOFTAP_MODE (*C enumerator*), 2721
wifi_operational_modes.WIFI_STA_MODE (*C enumerator*), 2721
wifi_operational_modes.WIFI_TX_INJECTION_MODE (*C enumerator*), 2721
wifi_ps (*C enum*), 2721
wifi_ps_config (*C struct*), 2739
wifi_ps_config.num_twt_flows (*C var*), 2739
wifi_ps_config.ps_params (*C var*), 2739
wifi_ps_config.twt_flows (*C var*), 2739
wifi_ps_get_config_err_code_str (*C function*), 2730
wifi_ps_mode (*C enum*), 2721
wifi_ps_mode_txt (*C function*), 2730
wifi_ps_mode.WIFI_PS_MODE_LEGACY (*C enumerator*), 2721
wifi_ps_mode.WIFI_PS_MODE_WMM (*C enumerator*), 2721
wifi_ps_param_type (*C enum*), 2724
wifi_ps_param_type.WIFI_PS_PARAM_LISTEN_INTERVAL (*C enumerator*), 2724
wifi_ps_param_type.WIFI_PS_PARAM_MODE (*C enumerator*), 2724
wifi_ps_param_type.WIFI_PS_PARAM_STATE (*C enumerator*), 2724
wifi_ps_param_type.WIFI_PS_PARAM_TIMEOUT (*C enumerator*), 2724
wifi_ps_param_type.WIFI_PS_PARAM_WAKEUP_MODE (*C enumerator*), 2724
wifi_ps_params (*C struct*), 2736
wifi_ps_params.enabled (*C var*), 2736
wifi_ps_params.fail_reason (*C var*), 2737
wifi_ps_params.listen_interval (*C var*), 2736
wifi_ps_params.mode (*C var*), 2737
wifi_ps_params.timeout_ms (*C var*), 2737
wifi_ps_params.type (*C var*), 2737
wifi_ps_params.wakeup_mode (*C var*), 2737
wifi_ps_txt (*C function*), 2730
wifi_ps_wakeup_mode (*C enum*), 2724
wifi_ps_wakeup_mode_txt (*C function*), 2730
wifi_ps_wakeup_mode.WIFI_PS_WAKEUP_MODE_DTIM (*C enumerator*), 2725
wifi_ps_wakeup_mode.WIFI_PS_WAKEUP_MODE_LISTEN_INTERVAL (*C enumerator*), 2725
WIFI_PSK_MAX_LEN (*C macro*), 2714
WIFI_PSK_MIN_LEN (*C macro*), 2714
wifi_ps.WIFI_PS_DISABLED (*C enumerator*), 2721
wifi_ps.WIFI_PS_ENABLED (*C enumerator*), 2721
wifi_raw_scan_result (*C struct*), 2740
wifi_raw_scan_result.data (*C var*), 2740
wifi_raw_scan_result.frame_length (*C var*), 2740
wifi_raw_scan_result.frequency (*C var*), 2740
wifi_raw_scan_result.rssi (*C var*), 2740
wifi_reg_chan_info (*C struct*), 2739
wifi_reg_chan_info.center_frequency (*C var*), 2739
wifi_reg_chan_info.dfs (*C var*), 2740
wifi_reg_chan_info.max_power (*C var*), 2739
wifi_reg_chan_info.passive_only (*C var*), 2740
wifi_reg_chan_info.supported (*C var*), 2739
wifi_reg_domain (*C struct*), 2740
wifi_reg_domain.chan_info (*C var*), 2740
wifi_reg_domain.country_code (*C var*), 2740
wifi_reg_domain.force (*C var*), 2740
wifi_reg_domain.num_channels (*C var*), 2740
wifi_reg_domain.oper (*C var*), 2740

WIFI_SAE_PSWD_MAX_LEN (*C macro*), 2714
wifi_scan_params (*C struct*), 2732
wifi_scan_params.band_chan (*C var*), 2733
wifi_scan_params.bands (*C var*), 2733
wifi_scan_params.dwell_time_active (*C var*), 2733
wifi_scan_params.dwell_time_passive (*C var*), 2733
wifi_scan_params.max_bss_cnt (*C var*), 2733
wifi_scan_params.scan_type (*C var*), 2733
wifi_scan_params.ssids (*C var*), 2733
wifi_scan_result (*C struct*), 2733
wifi_scan_result.band (*C var*), 2734
wifi_scan_result.channel (*C var*), 2734
wifi_scan_result.mac (*C var*), 2734
wifi_scan_result.mac_length (*C var*), 2734
wifi_scan_result.mfp (*C var*), 2734
wifi_scan_result.rssi (*C var*), 2734
wifi_scan_result.security (*C var*), 2734
wifi_scan_result.ssid (*C var*), 2733
wifi_scan_result.ssid_length (*C var*), 2734
wifi_scan_type (*C enum*), 2720
wifi_scan_type.WIFI_SCAN_TYPE_ACTIVE (*C enumerator*), 2720
wifi_scan_type.WIFI_SCAN_TYPE_PASSIVE (*C enumerator*), 2721
wifi_security_txt (*C function*), 2730
wifi_security_type (*C enum*), 2717
wifi_security_type.WIFI_SECURITY_TYPE_EAP (*C enumerator*), 2717
wifi_security_type.WIFI_SECURITY_TYPE_NONE (*C enumerator*), 2717
wifi_security_type.WIFI_SECURITY_TYPE_PSK (*C enumerator*), 2717
wifi_security_type.WIFI_SECURITY_TYPE_PSK_SHA256 (*C enumerator*), 2717
wifi_security_type.WIFI_SECURITY_TYPE_SAE (*C enumerator*), 2717
wifi_security_type.WIFI_SECURITY_TYPE_WAPI (*C enumerator*), 2717
wifi_security_type.WIFI_SECURITY_TYPE_WEP (*C enumerator*), 2718
wifi_security_type.WIFI_SECURITY_TYPE_WPA_AUTO_PERSONAL (*C enumerator*), 2718
wifi_security_type.WIFI_SECURITY_TYPE_WPA_PSK (*C enumerator*), 2718
WIFI_SSID_MAX_LEN (*C macro*), 2714
wifi_state_txt (*C function*), 2730
wifi_status (*C struct*), 2735
wifi_status.ap_status (*C var*), 2735
wifi_status.conn_status (*C var*), 2735
wifi_status.disconn_reason (*C var*), 2735
wifi_status.status (*C var*), 2735
wifi_twt_fail_reason (*C enum*), 2723
wifi_twt_fail_reason.WIFI_TWT_FAIL_CMD_EXEC_FAIL (*C enumerator*), 2723
wifi_twt_fail_reason.WIFI_TWT_FAIL_DEVICE_NOT_CONNECTED (*C enumerator*), 2723
wifi_twt_fail_reason.WIFI_TWT_FAIL_FLOW_ALREADY_EXISTS (*C enumerator*), 2724
wifi_twt_fail_reason.WIFI_TWT_FAIL_INVALID_FLOW_ID (*C enumerator*), 2724
wifi_twt_fail_reason.WIFI_TWT_FAIL_IP_NOT_ASSIGNED (*C enumerator*), 2724
wifi_twt_fail_reason.WIFI_TWT_FAIL_OPERATION_IN_PROGRESS (*C enumerator*), 2724
wifi_twt_fail_reason.WIFI_TWT_FAIL_OPERATION_NOT_SUPPORTED (*C enumerator*), 2723
wifi_twt_fail_reason.WIFI_TWT_FAIL_PEER_NOT_HE_CAPAB (*C enumerator*), 2723
wifi_twt_fail_reason.WIFI_TWT_FAIL_PEER_NOT_TWT_CAPAB (*C enumerator*), 2724
wifi_twt_fail_reason.WIFI_TWT_FAIL_UNABLE_TO_GET_IFACE_STATUS (*C enumerator*), 2723
wifi_twt_fail_reason.WIFI_TWT_FAIL_UNSPECIFIED (*C enumerator*), 2723
wifi_twt_flow_info (*C struct*), 2738
wifi_twt_flow_info.announce (*C var*), 2739
wifi_twt_flow_info.dialog_token (*C var*), 2738
wifi_twt_flow_info.flow_id (*C var*), 2738
wifi_twt_flow_info.implicit (*C var*), 2739
wifi_twt_flow_info.negotiation_type (*C var*), 2738

wifi_twt_flow_info.responder (*C var*), 2739
wifi_twt_flow_info.trigger (*C var*), 2739
wifi_twt_flow_info.twt_interval (*C var*), 2738
wifi_twt_flow_info.twt_wake_ahead_duration (*C var*), 2739
wifi_twt_flow_info.twt_wake_interval (*C var*), 2739
wifi_twt_get_err_code_str (*C function*), 2730
wifi_twt_negotiation_type (*C enum*), 2722
wifi_twt_negotiation_type_txt (*C function*), 2730
wifi_twt_negotiation_type.WIFI_TWT_BROADCAST (*C enumerator*), 2722
wifi_twt_negotiation_type.WIFI_TWT_INDIVIDUAL (*C enumerator*), 2722
wifi_twt_negotiation_type.WIFI_TWT_WAKE_TBTT (*C enumerator*), 2722
wifi_twt_operation (*C enum*), 2722
wifi_twt_operation_txt (*C function*), 2730
wifi_twt_operation.WIFI_TWT_SETUP (*C enumerator*), 2722
wifi_twt_operation.WIFI_TWT_TEARDOWN (*C enumerator*), 2722
wifi_twt_params (*C struct*), 2737
wifi_twt_params.announce (*C var*), 2738
wifi_twt_params.dialog_token (*C var*), 2737
wifi_twt_params.fail_reason (*C var*), 2738
wifi_twt_params.flow_id (*C var*), 2737
wifi_twt_params.implicit (*C var*), 2738
wifi_twt_params.negotiation_type (*C var*), 2737
wifi_twt_params.operation (*C var*), 2737
wifi_twt_params.resp_status (*C var*), 2737
wifi_twt_params.responder (*C var*), 2738
wifi_twt_params.setup (*C var*), 2738
wifi_twt_params.setup_cmd (*C var*), 2737
wifi_twt_params.teardown (*C var*), 2738
wifi_twt_params.teardown_all (*C var*), 2738
wifi_twt_params.teardown_status (*C var*), 2737
wifi_twt_params.trigger (*C var*), 2738
wifi_twt_params.twt_interval (*C var*), 2737
wifi_twt_params.twt_wake_ahead_duration (*C var*), 2738
wifi_twt_params.twt_wake_interval (*C var*), 2738
wifi_twt_setup_cmd (*C enum*), 2722
wifi_twt_setup_cmd_txt (*C function*), 2730
wifi_twt_setup_cmd.WIFI_TWT_SETUP_CMD_ACCEPT (*C enumerator*), 2723
wifi_twt_setup_cmd.WIFI_TWT_SETUP_CMD_ALTERNATE (*C enumerator*), 2723
wifi_twt_setup_cmd.WIFI_TWT_SETUP_CMD_DEMAND (*C enumerator*), 2722
wifi_twt_setup_cmd.WIFI_TWT_SETUP_CMD_DICTATE (*C enumerator*), 2723
wifi_twt_setup_cmd.WIFI_TWT_SETUP_CMD_GROUPING (*C enumerator*), 2723
wifi_twt_setup_cmd.WIFI_TWT_SETUP_CMD_REJECT (*C enumerator*), 2723
wifi_twt_setup_cmd.WIFI_TWT_SETUP_CMD_REQUEST (*C enumerator*), 2722
wifi_twt_setup_cmd.WIFI_TWT_SETUP_CMD_SUGGEST (*C enumerator*), 2722
wifi_twt_setup_resp_status (*C enum*), 2723
wifi_twt_setup_resp_status.WIFI_TWT_RESP_NOT_RECEIVED (*C enumerator*), 2723
wifi_twt_setup_resp_status.WIFI_TWT_RESP_RECEIVED (*C enumerator*), 2723
wifi_twt_sleep_state (*C enum*), 2729
wifi_twt_sleep_state.WIFI_TWT_STATE_AWAKE (*C enumerator*), 2730
wifi_twt_sleep_state.WIFI_TWT_STATE_SLEEP (*C enumerator*), 2729
wifi_twt_teardown_status (*C enum*), 2724
wifi_twt_teardown_status.WIFI_TWT_TEARDOWN_FAILED (*C enumerator*), 2724
wifi_twt_teardown_status.WIFI_TWT_TEARDOWN_SUCCESS (*C enumerator*), 2724
WIFI_UTILS_MAX_BAND_STR_LEN (*C macro*), 2714
WIFI_UTILS_MAX_CHAN_STR_LEN (*C macro*), 2714
wifi_utils_parse_scan_bands (*C function*), 2712
wifi_utils_parse_scan_chan (*C function*), 2713
wifi_utils_parse_scan_ssids (*C function*), 2712

wifi_utils_validate_chan (*C function*), 2713
wifi_utils_validate_chan_2g (*C function*), 2713
wifi_utils_validate_chan_5g (*C function*), 2714
wifi_utils_validate_chan_6g (*C function*), 2714
wifi_version (*C struct*), 2732
wifi_version.drv_version (*C var*), 2732
wifi_version.fw_version (*C var*), 2732
write() (*twister_harness.DeviceAdapter method*), 268
WRITE_BIT (*C macro*), 686

X

XCC_NO_G_FLAG, 289
XIP, **3937**
XTENSA_CORE, 289
XTENSA_TOOLCHAIN_PATH, 289
XTOOLS_TOOLCHAIN_PATH, 292

Z

zassert (*C macro*), 227
zassert_between_inclusive (*C macro*), 228
zassert_equal (*C macro*), 228
zassert_equal_ptr (*C macro*), 228
zassert_false (*C macro*), 227
zassert_is_null (*C macro*), 227
zassert_mem_equal (*C macro*), 229
zassert_mem_equal__ (*C macro*), 229
zassert_not_equal (*C macro*), 228
zassert_not_null (*C macro*), 228
zassert_not_ok (*C macro*), 227
zassert_ok (*C macro*), 227
zassert_str_equal (*C macro*), 229
zassert_true (*C macro*), 227
zassert_unreachable (*C macro*), 227
zassert_within (*C macro*), 228
zassume (*C macro*), 227
zassume_between_inclusive (*C macro*), 234
zassume_equal (*C macro*), 233
zassume_equal_ptr (*C macro*), 233
zassume_false (*C macro*), 232
zassume_is_null (*C macro*), 233
zassume_mem_equal (*C macro*), 234
zassume_mem_equal__ (*C macro*), 234
zassume_not_equal (*C macro*), 233
zassume_not_null (*C macro*), 233
zassume_not_ok (*C macro*), 232
zassume_ok (*C macro*), 232
zassume_str_equal (*C macro*), 234
zassume_true (*C macro*), 232
zassume_within (*C macro*), 234
zbus_chan_add_obs (*C function*), 1281
ZBUS_CHAN_ADD_OBS (*C macro*), 1275
ZBUS_CHAN_ADD_OBS_WITH_MASK (*C macro*), 1274
zbus_chan_claim (*C function*), 1278
zbus_chan_const_msg (*C function*), 1280
ZBUS_CHAN_DECLARE (*C macro*), 1275
ZBUS_CHAN_DEFINE (*C macro*), 1275
zbus_chan_finish (*C function*), 1279
zbus_chan_msg (*C function*), 1279

zbus_chan_msg_size (*C function*), 1280
zbus_chan_name (*C function*), 1279
zbus_chan_notify (*C function*), 1279
zbus_chan_pub (*C function*), 1277
zbus_chan_read (*C function*), 1278
zbus_chan_rm_obs (*C function*), 1281
zbus_chan_set_msg_sub_pool (*C function*), 1280
zbus_chan_user_data (*C function*), 1280
zbus_channel (*C struct*), 1286
zbus_channel_data (*C struct*), 1285
zbus_channel_data.msg_subscriber_pool (*C var*), 1286
zbus_channel_data.observers (*C var*), 1285
zbus_channel_data.observers_end_idx (*C var*), 1285
zbus_channel_data.observers_start_idx (*C var*), 1285
zbus_channel_data.sem (*C var*), 1285
zbus_channel.data (*C var*), 1286
zbus_channel.message (*C var*), 1286
zbus_channel.message_size (*C var*), 1286
zbus_channel.name (*C var*), 1286
zbus_channel.user_data (*C var*), 1286
zbus_channel.validator (*C var*), 1286
zbus_iterate_over_channels (*C function*), 1284
zbus_iterate_over_channels_with_user_data (*C function*), 1284
zbus_iterate_over_observers (*C function*), 1285
zbus_iterate_over_observers_with_user_data (*C function*), 1285
ZBUS_LISTENER_DEFINE (*C macro*), 1276
ZBUS_LISTENER_DEFINE_WITH_ENABLE (*C macro*), 1276
ZBUS_MSG_INIT (*C macro*), 1275
ZBUS_MSG_SUBSCRIBER_DEFINE (*C macro*), 1277
ZBUS_MSG_SUBSCRIBER_DEFINE_WITH_ENABLE (*C macro*), 1276
zbus_obs_attach_to_thread (*C function*), 1283
ZBUS_OBS_DECLARE (*C macro*), 1275
zbus_obs_detach_from_thread (*C function*), 1283
zbus_obs_is_chan_notification_masked (*C function*), 1282
zbus_obs_is_enabled (*C function*), 1282
zbus_obs_name (*C function*), 1283
zbus_obs_set_chan_notification_mask (*C function*), 1282
zbus_obs_set_enable (*C function*), 1281
zbus_observer (*C struct*), 1287
zbus_observer_data (*C struct*), 1286
zbus_observer_data.enabled (*C var*), 1287
zbus_observer_type (*C enum*), 1277
zbus_observer_type.ZBUS_OBSERVER_LISTENER_TYPE (*C enumerator*), 1277
zbus_observer_type.ZBUS_OBSERVER_MSG_SUBSCRIBER_TYPE (*C enumerator*), 1277
zbus_observer_type.ZBUS_OBSERVER_SUBSCRIBER_TYPE (*C enumerator*), 1277
zbus_observer.callback (*C var*), 1287
zbus_observer.data (*C var*), 1287
zbus_observer.message_fifo (*C var*), 1287
zbus_observer.name (*C var*), 1287
zbus_observer.queue (*C var*), 1287
ZBUS_OBSERVERS (*C macro*), 1275
ZBUS_OBSERVERS_EMPTY (*C macro*), 1275
zbus_observer.type (*C var*), 1287
zbus_sub_wait (*C function*), 1283
zbus_sub_wait_msg (*C function*), 1284
ZBUS_SUBSCRIBER_DEFINE (*C macro*), 1276
ZBUS_SUBSCRIBER_DEFINE_WITH_ENABLE (*C macro*), 1276
zephyr module, **3937**

ZEPHYR_BASE, 27, 138140, 181, 739, 1613
ZEPHYR_BOARD_ALIASES, 23
ZEPHYR_SCA_VARIANT, 281
ZEPHYR_SDK_INSTALL_DIR, 15, 20, 22, 285
ZEPHYR_TOOLCHAIN_VARIANT, 15, 22, 28, 242, 285, 288293
ZephyrBinaryRunner (*class in runners.core*), 198
ZERO_OR_COMPILE_ERROR (*C macro*), 680
zexpect (*C macro*), 227
zexpect_between_inclusive (*C macro*), 231
zexpect_equal (*C macro*), 230
zexpect_equal_ptr (*C macro*), 231
zexpect_false (*C macro*), 230
zexpect_is_null (*C macro*), 230
zexpect_mem_equal (*C macro*), 231
zexpect_not_equal (*C macro*), 231
zexpect_not_null (*C macro*), 230
zexpect_not_ok (*C macro*), 230
zexpect_ok (*C macro*), 230
zexpect_str_equal (*C macro*), 232
zexpect_true (*C macro*), 230
zexpect_within (*C macro*), 231
zsock_accept (*C function*), 2499
zsock_addrinfo (*C struct*), 2501
zsock_addrinfo.ai_addr (*C var*), 2502
zsock_addrinfo.ai_addrlen (*C var*), 2502
zsock_addrinfo.ai_canonname (*C var*), 2502
zsock_addrinfo.ai_eflags (*C var*), 2502
zsock_addrinfo.ai_family (*C var*), 2502
zsock_addrinfo.ai_flags (*C var*), 2502
zsock_addrinfo.ai_next (*C var*), 2501
zsock_addrinfo.ai_protocol (*C var*), 2502
zsock_addrinfo.ai_socktype (*C var*), 2502
zsock_bind (*C function*), 2498
zsock_close (*C function*), 2498
zsock_connect (*C function*), 2498
zsock_fcntl_impl (*C function*), 2499
ZSOCK_FD_CLR (*C function*), 2501
ZSOCK_FD_ISSET (*C function*), 2501
ZSOCK_FD_SET (*C function*), 2501
zsock_fd_set (*C struct*), 2504
zsock_fd_set (*C type*), 2497
ZSOCK_FD_SETSIZE (*C macro*), 2497
ZSOCK_FD_ZERO (*C function*), 2501
zsock_freeaddrinfo (*C function*), 2500
zsock_gai_strerror (*C function*), 2501
zsock_get_context_object (*C function*), 2497
zsock_getaddrinfo (*C function*), 2500
zsock_gethostname (*C function*), 2500
zsock_getnameinfo (*C function*), 2501
zsock_getpeername (*C function*), 2500
zsock_getsockname (*C function*), 2500
zsock_getsockopt (*C function*), 2500
zsock_inet_ntop (*C function*), 2500
zsock_inet_pton (*C function*), 2500
zsock_ioctl_impl (*C function*), 2499
zsock_listen (*C function*), 2499
ZSOCK_MSG_CTRUNC (*C macro*), 2491
ZSOCK_MSG_DONTWAIT (*C macro*), 2491

ZSOCK_MSG_PEEK (*C macro*), 2491
ZSOCK_MSG_TRUNC (*C macro*), 2491
ZSOCK_MSG_WAITALL (*C macro*), 2492
zsock_poll (*C function*), 2500
ZSOCK_POLLERR (*C macro*), 2491
zsock_pollfd (*C struct*), 2503
zsock_pollfd.events (*C var*), 2504
zsock_pollfd.fd (*C var*), 2504
zsock_pollfd.revents (*C var*), 2504
ZSOCK_POLLHUP (*C macro*), 2491
ZSOCK_POLLIN (*C macro*), 2491
ZSOCK_POLLNVAL (*C macro*), 2491
ZSOCK_POLLOUT (*C macro*), 2491
ZSOCK_POLLPRI (*C macro*), 2491
zsock_recv (*C function*), 2499
zsock_recvfrom (*C function*), 2499
zsock_recvmsg (*C function*), 2499
zsock_select (*C function*), 2501
zsock_send (*C function*), 2499
zsock_sendmsg (*C function*), 2499
zsock_sendto (*C function*), 2499
zsock_setsockopt (*C function*), 2500
ZSOCK_SHUT_RD (*C macro*), 2492
ZSOCK_SHUT_RDWR (*C macro*), 2492
ZSOCK_SHUT_WR (*C macro*), 2492
zsock_shutdown (*C function*), 2498
zsock_socket (*C function*), 2498
zsock_socketpair (*C function*), 2498
ZTEST (*C macro*), 220
ztest_arch_api (*C struct*), 226
ZTEST_BMEM (*C macro*), 219
ztest_check_expected_data (*C macro*), 280
ztest_check_expected_value (*C macro*), 280
ztest_copy_return_data (*C macro*), 280
ZTEST_DMEM (*C macro*), 219
ztest_expect_data (*C macro*), 280
ZTEST_EXPECT_FAIL (*C macro*), 218
ZTEST_EXPECT_SKIP (*C macro*), 219
ztest_expect_value (*C macro*), 279
ztest_expected_result (*C enum*), 222
ztest_expected_result_entry (*C struct*), 224
ztest_expected_result_entry.expected_result (*C var*), 225
ztest_expected_result_entry.test_name (*C var*), 225
ztest_expected_result_entry.test_suite_name (*C var*), 225
ztest_expected_result.ZTEST_EXPECTED_RESULT_FAIL (*C enumerator*), 222
ztest_expected_result.ZTEST_EXPECTED_RESULT_SKIP (*C enumerator*), 222
ZTEST_F (*C macro*), 220
ztest_get_return_value (*C macro*), 281
ztest_get_return_value_ptr (*C macro*), 281
ztest_mem_partition (*C var*), 224
ztest_phase (*C enum*), 223
ztest_phase.TEST_PHASE_AFTER (*C enumerator*), 223
ztest_phase.TEST_PHASE_BEFORE (*C enumerator*), 223
ztest_phase.TEST_PHASE_FRAMEWORK (*C enumerator*), 223
ztest_phase.TEST_PHASE_SETUP (*C enumerator*), 223
ztest_phase.TEST_PHASE_TEARDOWN (*C enumerator*), 223
ztest_phase.TEST_PHASE_TEST (*C enumerator*), 223
ztest_result (*C enum*), 222

ztest_result.ZTEST_RESULT_FAIL (*C enumerator*), 222
ztest_result.ZTEST_RESULT_PASS (*C enumerator*), 222
ztest_result.ZTEST_RESULT_PENDING (*C enumerator*), 222
ztest_result.ZTEST_RESULT_SKIP (*C enumerator*), 222
ztest_result.ZTEST_RESULT_SUITE_FAIL (*C enumerator*), 223
ztest_result.ZTEST_RESULT_SUITE_SKIP (*C enumerator*), 223
ztest_return_data (*C macro*), 280
ztest_returns_value (*C macro*), 281
ZTEST_RULE (*C macro*), 220
ztest_rule_cb (*C type*), 222
ztest_run_all (*C function*), 223
ztest_run_test_suite (*C macro*), 221
ztest_run_test_suites (*C function*), 223
ZTEST_SECTION (*C macro*), 220
ztest_simple_1cpu_after (*C function*), 224
ztest_simple_1cpu_before (*C function*), 224
ztest_skip_failed_assumption (*C function*), 224
ZTEST_SUITE (*C macro*), 219
ztest_suite_after_t (*C type*), 221
ztest_suite_before_t (*C type*), 221
ZTEST_SUITE_COUNT (*C macro*), 219
ztest_suite_node (*C struct*), 226
ztest_suite_node.after (*C var*), 226
ztest_suite_node.before (*C var*), 226
ztest_suite_node.name (*C var*), 226
ztest_suite_node.predicate (*C var*), 226
ztest_suite_node.setup (*C var*), 226
ztest_suite_node.stats (*C var*), 226
ztest_suite_node.teardown (*C var*), 226
ztest_suite_predicate_t (*C type*), 221
ztest_suite_setup_t (*C type*), 221
ztest_suite_stats (*C struct*), 225
ztest_suite_stats.fail_count (*C var*), 225
ztest_suite_stats.run_count (*C var*), 225
ztest_suite_stats.skip_count (*C var*), 225
ztest_suite_teardown_t (*C type*), 221
ZTEST_TEST_COUNT (*C macro*), 219
ztest_test_fail (*C function*), 224
ztest_test_pass (*C function*), 224
ztest_test_rule (*C struct*), 226
ztest_test_skip (*C function*), 224
ztest_unit_test (*C struct*), 225
ztest_unit_test_stats (*C struct*), 225
ztest_unit_test_stats.duration_worst_ms (*C var*), 226
ztest_unit_test_stats.fail_count (*C var*), 226
ztest_unit_test_stats.pass_count (*C var*), 226
ztest_unit_test_stats.run_count (*C var*), 225
ztest_unit_test_stats.skip_count (*C var*), 225
ztest_unit_test.stats (*C var*), 225
ZTEST_USER (*C macro*), 220
ZTEST_USER_F (*C macro*), 220
ztest_verify_all_test_suites_ran (*C function*), 223
ztress_abort (*C function*), 237
ztress_context_data (*C struct*), 238
ZTRESS_CONTEXT_INITIALIZER (*C macro*), 236
ztress_exec_count (*C function*), 237
ztress_execute (*C function*), 236
ZTRESS_EXECUTE (*C macro*), 236

ztress_handler (*C type*), 236
ztress_optimized_ticks (*C function*), 237
ztress_preempt_count (*C function*), 237
ztress_report (*C function*), 237
ztress_set_timeout (*C function*), 237
ZTRESS_THREAD (*C macro*), 235
ZTRESS_TIMER (*C macro*), 235